



**NATIONAL TECHNICAL UNIVERSITY OF ATHENS**

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF INFORMATICS AND COMPUTER TECHNOLOGY

**Accelerating Irregular Applications  
via Efficient Synchronization  
and Data Access Techniques**

Doctoral Dissertation

**CHRISTINA GIANNOULA**

Athens, September 2022





NATIONAL TECHNICAL UNIVERSITY OF ATHENS  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING  
DIVISION OF INFORMATICS AND COMPUTER TECHNOLOGY

**Accelerating Irregular Applications  
via Efficient Synchronization  
and Data Access Techniques**


Doctoral Dissertation

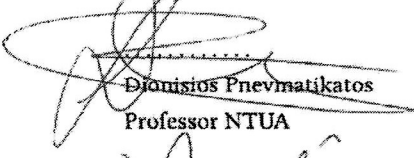
**CHRISTINA GIANNOULA**

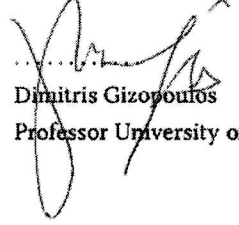
**Advisory Committee:**


Georgios Goumas  
Nectarios Koziris  
Onur Mutlu

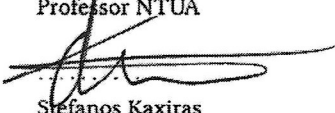
Accepted by the seven-member committee on 27/09/2022.


  
Georgios Goumas  
Associate Professor NTUA


  
Dionisios Pnevmatikatos  
Professor NTUA

  
Dimitris Gizopoulos  
Professor University of Athens

  
Nectarios Koziris  
Professor NTUA

  
Stefanos Kaxiras  
Professor Uppsala University

  
Vasileios Papaefstathiou  
Assistant Professor University of  
Crete

  
Onur Mutlu  
Professor ETH Zurich

Athens, September 2022

.....

**CHRISTINA GIANNOULA**

Doctor of Electrical and Computer Engineer, NTUA

Copyright © CHRISTINA GIANNOULA, 2022

All rights reserved.

Copying, storage and distribution of this work, in whole or part of it, is prohibited for commercial purposes. Reproduction, storage and distribution for the purpose of non-profit, educational or research nature, is allowed provided that the source of origin is mentioned and the copyright message is maintained. Questions concerning the use of this work for commercial purposes should be addressed to the author.

The views and conclusions contained in this document reflect the author and should not be interpreted as representing the official position of the National technical University of Athens.

*To my loving parents, Maria and Christoforos.*  
*Στους αγαπημένους μου γονείς, Μαρία και Χριστόφορο.*



---

# Acknowledgments

---

This doctoral thesis is the culmination of five and half years of hard work throughout my PhD studies. My PhD journey was a significant source of learning and growth for me both professionally and personally. There have been many who have supported me and contributed in different ways. These acknowledgments comprise a brief and humble attempt to thank their invaluable contributions.

First and foremost, I wholeheartedly thank my advisors, Prof. Georgios Goumas, Prof. Nectarios Koziris, and Prof. Onur Mutlu. I am very grateful to Prof. Georgios Goumas for helping me to find and follow a very interesting research direction for me, supporting and advising me with great patience and tolerance, motivating me to work on the fields of high-performance computing and computer architecture and providing me a very comfortable, safe and stimulating environment to grow. I also thank him for being open to my collaborations with researchers from other institutions. I am extremely grateful to Prof. Onur Mutlu for his generous guidance, resources and opportunities which constitute the key to my professional growth, success and research achievements. I thank him for giving me the invaluable opportunity to work with him and his research group, providing rigorous feedback to my paper submissions and talks, teaching me how to think critically, write comprehensively, and perform impactful research. His motivation for top-notch research and his passion for excellence were a constant source of inspiration and have significantly shaped my research mindset and personality. I am grateful to Prof. Nectarios Koziris for giving the opportunity to be a member of his research laboratory, inspiring me with his incredible passion for teaching and working in the field of computer architecture, as well as his continuous support and the encouraging environment he has provided. My advisors' influence and constant encouragement provided real-life lessons and shaped my personality as a researcher, scientist and engineer.

I thank my committee members, Dionisios Pnevmatikatos, Stefanos Kaxiras, Dimitris Gizopoulos, and Vasileios Papaefstathiou for supervising this thesis. Their feedback and suggestions were valuable to improving my doctoral thesis and its constituent works.

I am grateful to Nandita Vijaykumar for being a great mentor and encouraging me to become a strong and independent researcher. During my visit at the SAFARI research group of ETH Zurich, Nandita helped me to stay motivated, taught me how to find the right research problem to work on and how to perform quality research. I also wholeheartedly thank Athena Elafrou, Foteini Strati, Ivan Fernandez and Thomas Lagos for being my closest collaborators and friends throughout PhD studies, the many long hours of brainstorming and our stimulating discussions. I am very grateful for their endless support, valuable feedback, kindness, positivity, and confidence in myself, as well as our invaluable synergy and friendship.

Furthermore, I thank all the CSLab group members for being great colleagues, supporting my research and enabling a productive working environment. I want to especially thank Kostis Nikas, Vasileios Karakostas, Nikela Papadopoulou and Dimitris Siakavaras for providing significant intellectual and technical support in my research contributions and willingly sharing their expertise with me. I am grateful to all the students and mentees with who I worked closely: Foteini Strati, Athanasios Peppas, and Thrasyvoulos-Fivos Iliadis. Their work significantly helped me in completing my PhD thesis.

I am immensely grateful to all the members of the SAFARI research group for creating a rich, stimulating and highly motivating research environment. During my visit at the SAFARI research group, I realized that innovative research can vastly benefit from close collaboration among the group members. The valuable advice of the SAFARI group members, consisting of concrete guidelines and useful methodologies, helped me to work effectively and efficiently and stay focused in tackling my research obstacles. I want to especially thank Juan Gomez-Luna, Lois Orosa, Konstantinos Kanellopoulos and Nika Mansouri-Ghiasi for their rigorous feedback and criticism on my progress and research, their friendship, their technical and intellectual suggestions, as well as for generously sharing their deep knowledge on the field of computer architecture with me.

I gratefully acknowledge financial support from my PhD scholarships. Specifically, it was a great honor for me to receive a PhD Fellowship (October 2017 - March 2020) from the General Secretariat for Research and Technology (GSRT) and the Hellenic Foundation for Research and Innovation (HFRI) and a PhD award (September 2021 - October 2022) funded by the Foundation for Education and European Culture (IPEP).

I am immensely grateful to my friends for their support, companionship and patience. I want to particularly thank Katerina Bogiatzoglou, Konstantina Kada, Vicky Routsis, Orestis Alpos, Stamatios Kourkoutas, Stamatios Anoustis, Artemis Zografou, Marina Gourgioti, Katerina Tsesmeli, Isidora Tourni, Athina Kyriakou, Foteini Strati, and Thomas Lagos for our endless conversations, countless laughs, fun nights out and beautiful trips and excursions that were the best discharge from the hard work that I did during my PhD studies.

Last but not least, I am tremendously blessed and would like to express my profound gratitude to my parents, Maria and Christoforos, and my sister, Chara, for their unconditional love and endless encouragement throughout my PhD journey, as well as their valuable support to pursue my academic dreams. I thank my mother for continuously supporting each and every step of this journey. I thank my father for always believing in myself and for helping me with his optimism to pursue my dreams.



I thank my sister for her valuable support and patience throughout my PhD studies. This dissertation would not be possible without them. I will be forever grateful to my loving family for the dedication, support, patience, love and opportunities they have given me.



---

# Abstract

---

Irregular applications comprise an increasingly important workload domain for many fields, including bioinformatics, chemistry, graph analytics, physics, social sciences and machine learning. Therefore, achieving high performance and energy efficiency in the execution of emerging irregular applications is of vital importance. While there is abundant research on accelerating irregular applications, in this thesis, we identify two critical challenges. First, irregular applications are hard to scale to a high number of parallel threads due to high synchronization overheads. Second, irregular applications have complex memory access patterns and exhibit low operational intensity, and thus they are bottlenecked by expensive data access costs.

This doctoral thesis studies the root causes of inefficiency of irregular applications in modern computing systems, and aims to fundamentally address such inefficiencies, by 1) proposing low-overhead synchronization techniques among parallel threads in cooperation with 2) well-crafted data access policies. Our approach leads to high system performance and energy efficiency on the execution of irregular applications in modern computing platforms, both processor-centric CPU systems and memory-centric Processing-In-Memory (PIM) systems.

We make four major contributions to accelerating irregular applications in different contexts including CPU and Near-Data-Processing (NDP) (or Processing-In-Memory (PIM)) systems. First, we design *ColorTM*, a novel parallel graph coloring algorithm for CPU systems that trades off using synchronization with lower data access costs. *ColorTM* proposes an efficient data management technique co-designed with a speculative synchronization scheme implemented on Hardware Transactional Memory, and significantly outperforms prior state-of-the-art graph coloring algorithms across a wide range of real-world graphs. Second, we propose *SmartPQ*, an adaptive priority queue that achieves high performance under all various contention scenarios in Non-Uniform Memory Access (NUMA) CPU systems. *SmartPQ* tunes itself by dynamically switching between a NUMA-oblivious and a NUMA-aware algorithmic mode, thus providing low data access costs in high contention sce-

narios, and high levels of parallelism in low contention scenarios. Our evaluations show that *SmartPQ* achieves the highest throughput over prior state-of-the-art NUMA-aware and NUMA-oblivious concurrent priority queues under various contention scenarios and even when contention varies during runtime. Third, we introduce *SynCron*, the first practical and lightweight hardware synchronization mechanism tailored for NDP systems. *SynCron* minimizes synchronization overheads in NDP systems by (i) adding low-cost hardware support near memory for synchronization acceleration, (ii) directly buffering the synchronization variables in a specialized cache memory structure, (iii) implementing a hierarchical message-passing communication scheme, and (iv) integrating a hardware-only overflow management scheme to avoid performance degradation when hardware resources for synchronization tracking are exceeded. We demonstrate that *SynCron* outperforms prior state-of-the-art approaches both in performance and energy consumption using a wide range of irregular applications, and has low hardware area and power overheads. Fourth, we design *SparseP*, the first library for high-performance Sparse Matrix Vector Multiplication (SpMV) on real PIM systems. *SparseP* is publicly-available and includes a wide range of data partitioning, load balancing, compression and synchronization techniques to accelerate this irregular kernel in current and future PIM systems. We also extensively characterize the widely used SpMV kernel on a real PIM architecture, and provide recommendations for software, system and hardware designers of future PIM systems.

Overall, we demonstrate that the execution of irregular applications in CPU and NDP/PIM architectures can be significantly accelerated by co-designing lightweight synchronization approaches along with well-crafted data access policies. Specifically, we show that efficient synchronization and data access techniques can provide high amount of parallelism, low-overhead inter-thread communication and low data access and data movement costs in emerging irregular applications, thus significantly improving system performance and system energy. This doctoral thesis also bridges the gap between processor-centric CPU systems and memory-centric PIM systems in the critically-important area of irregular applications. We hope that this dissertation inspires future work in co-designing software algorithms with cutting-edge computing platforms to significantly accelerate emerging irregular applications.

**Keywords:** Irregular Applications, Synchronization, Efficient Data Access Techniques, Multicore Systems, Processing-In-Memory Architectures

---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>27</b>
1.1	Motivation: Excessive Synchronization and High Memory Intensity Degrade the Execution of Irregular Applications . . . . .	28
1.2	Our Approach: Efficient Synchronization and Data Access Techniques for Irregular Applications . . . . .	31
1.2.1	Thesis Statement . . . . .	33
1.3	Overview of Our Research . . . . .	33
1.3.1	<i>ColorTM</i> [1–3]: High-Performance and Balanced Parallel Graph Coloring on Multicore CPU Platforms (Chapter 2) . . . . .	33
1.3.2	<i>SmartPQ</i> [4]: An Adaptive Concurrent Priority Queue for NUMA CPU Architectures (Chapter 3) . . . . .	34
1.3.3	<i>SynCron</i> [5]: Efficient Synchronization Support for NDP Architectures (Chapter 4) . . . . .	35
1.3.4	<i>SparseP</i> [6–11]: Towards Efficient Sparse Matrix Vector Multiplication on Real PIM Architectures (Chapter 5) . . . . .	36
1.4	Contributions . . . . .	37
1.5	Outline . . . . .	39
<b>2</b>	<b><i>ColorTM</i></b>	<b>41</b>
2.1	Overview . . . . .	41
2.2	Prior Graph Coloring Algorithms . . . . .	44
2.2.1	The Greedy Algorithm . . . . .	44
2.2.2	Prior Parallel Graph Coloring Algorithms . . . . .	44
	The SeqSolve Algorithm . . . . .	45

	The IterSolve Algorithm . . . . .	46
	The IterSolveR Algorithm . . . . .	47
2.2.3	Prior Balanced Graph Coloring Algorithms . . . . .	48
	The Color-Centric (CLU) Algorithm . . . . .	49
	The Vertex-Centric (VFF) Algorithm . . . . .	50
	The Recoloring Algorithm . . . . .	51
2.3	<i>ColorTM</i> : Overview . . . . .	53
2.3.1	Speculative Computation and Synchronization . . . . .	53
2.3.2	Eager Coloring Conflict Detection and Resolution . . . . .	54
2.4	<i>ColorTM</i> : Detailed Design . . . . .	55
2.4.1	Speculative Synchronization via HTM . . . . .	55
2.4.2	Critical Adjacent Vertices . . . . .	58
2.4.3	Implementation Details . . . . .	59
2.4.4	Progress and Correctness . . . . .	61
2.4.5	The <i>BalColorTM</i> Algorithm . . . . .	63
2.5	Evaluation Methodology . . . . .	64
2.6	Evaluation . . . . .	64
2.6.1	Analysis of Parallel Graph Coloring Algorithms . . . . .	64
	Analysis of the Coloring Quality . . . . .	65
	Performance Comparison . . . . .	66
	Analysis of <i>ColorTM</i> Execution . . . . .	69
2.6.2	Analysis of Balanced Graph Coloring Algorithms . . . . .	72
	Analysis of Color Balancing Quality . . . . .	72
	Performance Comparison . . . . .	73
	Analysis of <i>BalColorTM</i> Execution . . . . .	77
2.6.3	Analysis of a Real-World Scenario . . . . .	77
2.7	Recommendations . . . . .	81
2.8	Related Work . . . . .	82
2.9	Summary . . . . .	84
<b>3</b>	<b><i>SmartPQ</i></b> . . . . .	<b>85</b>
3.1	Overview . . . . .	85
3.2	NUMA Node Delegation ( <i>Nuddle</i> ) . . . . .	88
3.2.1	Overview . . . . .	88
3.2.2	Implementation Details . . . . .	89
3.3	<i>SmartPQ</i> . . . . .	92
3.3.1	Selecting the Algorithmic Mode . . . . .	93
	The Need for a Machine Learning Approach . . . . .	93
	Decision Tree Classifier . . . . .	94
3.3.2	Implementation Details . . . . .	95

3.4	Experimental Evaluation . . . . .	97
3.4.1	Throughput of <i>Nuddle</i> . . . . .	98
3.4.2	Throughput of <i>SmartPQ</i> . . . . .	100
	Classifier Accuracy . . . . .	100
	Varying the Contention Workload . . . . .	100
3.5	Discussion and Future Work . . . . .	103
3.6	Recommendations . . . . .	103
3.7	Related Work . . . . .	104
3.8	Summary . . . . .	105
<b>4</b>	<b><i>SynCron</i></b>	<b>107</b>
4.1	Overview . . . . .	107
4.2	Background and Motivation . . . . .	110
4.2.1	Baseline Architecture . . . . .	110
4.2.2	The Solution Space for Synchronization . . . . .	111
	Synchronization via Shared Memory . . . . .	111
	Message-passing Synchronization . . . . .	112
4.3	<i>SynCron</i> : Overview . . . . .	113
4.3.1	Overview of <i>SynCron</i> . . . . .	113
4.3.2	<i>SynCron</i> 's Operation . . . . .	114
4.4	<i>SynCron</i> : Detailed Design . . . . .	115
4.4.1	Programming Interface and ISA Extensions . . . . .	115
	Memory Consistency . . . . .	116
	Message Encoding . . . . .	117
4.4.2	Synchronization Engine (SE) . . . . .	118
	Synchronization Processing Unit (SPU) . . . . .	118
	Synchronization Table (ST) . . . . .	118
	Indexing Counters . . . . .	119
	Control Flow in SE . . . . .	119
4.4.3	Overflow Management . . . . .	119
	<i>SynCron</i> 's Synchronization Variable . . . . .	120
	Communication Protocol between SEs . . . . .	120
4.4.4	<i>SynCron</i> Enhancements . . . . .	121
	RMW Operations . . . . .	121
	Lock Fairness . . . . .	121
4.4.5	Comparison with Prior Work . . . . .	122
4.4.6	Use of <i>SynCron</i> in Conventional Systems . . . . .	122
4.5	Methodology . . . . .	123
4.6	Evaluation . . . . .	126
4.6.1	Performance . . . . .	126

	Synchronization Primitives . . . . .	126
	Pointer-Chasing Data Structures . . . . .	126
	Real Applications . . . . .	128
4.6.2	Energy Consumption . . . . .	129
4.6.3	Data Movement . . . . .	129
4.6.4	Non-Uniformity of NDP Systems . . . . .	130
	High Contention . . . . .	130
	Low Contention . . . . .	130
4.6.5	Memory Technologies . . . . .	131
4.6.6	Effect of Data Placement . . . . .	132
4.6.7	<i>SynCron</i> 's Design Choices . . . . .	132
	Hierarchical Design . . . . .	132
	ST Size . . . . .	134
	Overflow Management . . . . .	134
4.6.8	<i>SynCron</i> 's Area and Power Overhead . . . . .	135
4.7	Recommendations . . . . .	136
4.8	Related Work . . . . .	137
4.9	Summary . . . . .	138
<b>5</b>	<b><i>SparseP</i></b>	<b>139</b>
5.1	Overview . . . . .	139
5.2	Background and Motivation . . . . .	143
5.2.1	Sparse Matrix Vector Multiplication (SpMV) . . . . .	143
	Compressed Matrix Storage Formats . . . . .	143
	SpMV in Processor-Centric Systems . . . . .	144
5.2.2	Near-Bank PIM Systems . . . . .	145
	The UPMEM PIM Architecture . . . . .	145
5.3	The <i>SparseP</i> Library . . . . .	146
5.3.1	SpMV Execution on a PIM System . . . . .	146
5.3.2	Overview of Data Partitioning Techniques . . . . .	147
5.3.3	Parallelization Techniques Across PIM Cores . . . . .	148
	1D Partitioning Technique . . . . .	148
	2D Partitioning Technique . . . . .	149
5.3.4	Parallelization Techniques Across Threads within a PIM Core . . . . .	150
	Load Balancing Approaches . . . . .	151
	Synchronization Approaches . . . . .	152
5.3.5	Kernel Implementation . . . . .	152
5.4	Evaluation Methodology . . . . .	154
5.5	Analysis of SpMV Execution on One DPU . . . . .	155
5.5.1	Load Balancing Schemes Across Tasklets of One DPU . . . . .	155



5.5.2	Analysis of Compressed Matrix Formats on One DPU . . . . .	157
5.6	Analysis of SpMV Execution on Multiple DPUs . . . . .	159
5.6.1	Analysis of SpMV Execution Using 1D Partitioning Techniques . . . . .	159
	Analysis of Kernel Time . . . . .	159
	Analysis of End-To-End SpMV Execution . . . . .	163
5.6.2	Analysis of SpMV Execution Using 2D Partitioning Techniques . . . . .	165
	Sensitivity Studies on 2D Partitioning Techniques . . . . .	165
	Analysis of Compressed Formats . . . . .	170
	Comparison of 2D Partitioning Techniques . . . . .	171
5.6.3	Comparison of 1D and 2D Partitioning Techniques . . . . .	172
5.7	Comparison with CPUs and GPUs . . . . .	174
5.7.1	Performance Comparison . . . . .	174
5.7.2	Energy Comparison . . . . .	176
5.7.3	Discussion . . . . .	177
5.8	Key Takeaways and Recommendations . . . . .	177
5.9	Related Work . . . . .	179
5.10	Summary . . . . .	181
<b>6</b>	<b>Conclusions and Future Directions</b>	<b>183</b>
6.1	Future Research Directions . . . . .	185
6.1.1	Accelerating Irregular Applications in Unconventional Systems . . . . .	185
6.1.2	Adaptive Algorithmic, System-Level and Hardware-Based Approaches for Irregular Applications . . . . .	188
6.2	Concluding Remarks . . . . .	190
<b>7</b>	<b>Other Works of the Author</b>	<b>191</b>
<b>8</b>	<b>Appendix A</b>	<b>193</b>
8.1	Extended Results for <i>SparseP</i> . . . . .	193
8.1.1	Synchronization Approaches in Block-Based Compressed Matrix Formats . . . . .	193
8.1.2	Fine-Grained Data Transfers in 2D Partitioning Techniques . . . . .	194
8.1.3	Effect of the Number of Vertical Partitions Using Two Different UPMEM PIM Systems . . . . .	195
8.1.4	Performance of Compressed Matrix Formats Using 2D Partitioning Techniques	195
8.1.5	Analysis of 1D- and 2D-Partitioned Kernels in Two UPMEM PIM Systems . . . . .	197
8.2	Arithmetic Throughput of One DPU for the Multiplication Operation . . . . .	199
8.3	The <i>SparseP</i> Software Package . . . . .	201
8.4	Large Matrix Dataset . . . . .	202



---

# List of Figures

---

1.1	(a) Dense Matrix Vector Multiplication using three parallel threads. (b) Sparse Matrix Vector Multiplication with a coarse-grained parallelization strategy among three parallel threads. (c) Sparse Matrix Vector Multiplication with a fine-grained parallelization strategy among three parallel threads. The colored cells of each matrix represent non-zero elements. . . . .	29
1.2	An example SpMV execution on the first four rows of a sparse $9 \times 9$ matrix with only 10 non-zero elements. The execution steps are performed at a row granularity. The colored cells of the matrix with purple color represent non-zero elements, and the colored cells of the input vector represent the elements of the input vector that are processed/accessed at each execution step. . . . .	30
2.1	The Greedy algorithm. . . . .	44
2.2	The SeqSolve algorithm. . . . .	45
2.3	The IterSolve algorithm. . . . .	46
2.4	The IterSolveR algorithm. . . . .	48
2.5	The CLU algorithm. . . . .	49
2.6	The VFF algorithm. . . . .	50
2.7	The Recoloring algorithm. . . . .	52
2.8	A Naive Approach. . . . .	53
2.9	<i>ColorTM</i> : Overview. . . . .	54
2.10	An example execution scenario in which threads $T1$ and $T2$ attempt to concurrently find colors for the vertices $v$ and $x$ , respectively, using a) HTM and b) fine-grained locking for synchronization. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color. . . . .	56

2.11	An example execution scenario in which threads $T1$ and $T2$ attempt to concurrently update the colors of the vertices $v$ and $u$ respectively, using two different transactions, and the HTM mechanism detects read-write conflicts to their data sets. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color. . . . .	57
2.12	An example execution scenario in which the graph is partitioned across two parallel threads. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color. When the threads $T1$ and $T2$ attempt to color the vertices $v$ and $u$ , respectively, the critical adjacent vertices that need to be validated within the critical section (HTM) are <i>only</i> the vertices $u$ and $v$ , respectively.	59
2.13	The <i>ColorTM</i> algorithm. . . . .	60
2.14	The <i>BalColorTM</i> algorithm. . . . .	62
2.15	Scalability achieved by all parallel graph coloring implementations in large real-world graphs. . . . .	67
2.16	Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads). . . . .	68
2.17	Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads). . . .	69
2.18	Abort ratio exhibited by <i>ColorTM</i> in all large real-world graphs. . . . .	70
2.19	Breakdown of different types of aborts exhibited by <i>ColorTM</i> in real-world graphs. . .	71
2.20	Distribution of color class sizes produced by <i>ColorTM</i> and all our evaluated balanced graph coloring schemes. Note that small color class sizes result in reduced parallelism in the real-world end-application. . . . .	74
2.21	Scalability achieved by all balanced graph coloring implementations in large real-world graphs. . . . .	75
2.22	Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads). . . . .	76
2.23	Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads). . . . .	76
2.24	Abort ratio exhibited by <i>BalColorTM</i> in all large real-world graphs. . . . .	77

2.25	Scalability of the end-to-end Community Detection execution achieved by (i) the Grappolo [12] parallelization approach of the Louvain method (SimpleCD) and (ii) the chromatic scheduling parallelization approach with <i>ColorTM</i> ( <i>ColorTMCD</i> ) and (iii) the chromatic scheduling parallelization approach with both <i>ColorTM</i> and <i>BalColorTM</i> ( <i>BalColorTMCD</i> ) in large real-world graphs. . . . .	79
2.26	Speedup of the actual kernel of the Community Detection execution achieved by (i) SimpleCD (D), (ii) <i>ColorTMCD</i> (C) and (iii) <i>BalColorTMCD</i> (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads). . . . .	80
2.27	Speedup breakdown of the end-to-end Community Detection execution achieved by (i) SimpleCD (D), (ii) <i>ColorTMCD</i> (C) and (iii) <i>BalColorTMCD</i> (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads). . . . .	81
3.1	Throughput achieved by a NUMA-oblivious [13, 14] and a NUMA-aware [15] priority queue, both initialized with 1024 keys. We use 64 threads that perform a mix of <i>insert</i> and <i>deleteMin</i> operations in parallel, and the key range is set to 2048 keys. We use <i>all</i> NUMA nodes of a 4-node NUMA system, the characteristics of which are presented in Section 3.4. . . . .	86
3.2	High-level overview of <i>SmartPQ</i> . <i>SmartPQ</i> dynamically adapts its algorithm to the contention levels of the workload based on the prediction of a simple classifier. . . . .	87
3.3	High-level design of <i>ffwd</i> [15] and <i>Nuddle</i> . <i>Nuddle</i> locates <i>all</i> server threads at the <i>same</i> NUMA node to design a NUMA-aware scheme, and associates each of them to multiple client thread groups. <i>Nuddle</i> uses the communication protocol proposed in <i>ffwd</i> [15]. . . . .	89
3.4	Helper structures of <i>Nuddle</i> . . . . .	90
3.5	Initialization functions of <i>Nuddle</i> . . . . .	91
3.6	Functions used by server threads and client threads to perform operations using <i>Nuddle</i> . . . . .	92
3.7	Throughput achieved by <i>Nuddle</i> (using 8 server threads) and its underlying NUMA-oblivious <i>base algorithm</i> , i.e., <i>alistarh-herlihy</i> [13, 14], when we vary (a) the number of threads that perform operations in the shared data structure, and (b) the key range of the workload. . . . .	93
3.7a	Varying the number of threads. . . . .	93
3.7b	Varying the key range. . . . .	93
3.8	The modified code of <i>Nuddle</i> with the decision-making mechanism to implement <i>SmartPQ</i> . . . . .	96

3.9	Throughput of concurrent priority queue implementations. The columns show different priority queue sizes using the key range of double the elements of each size. The rows show different operation workloads. The vertical line in each plot shows the point after which we oversubscribe software threads to hardware contexts. . . . .	99
3.10	Throughput achieved by <i>SmartPQ</i> , <i>Nuddle</i> and its underlying <i>base algorithm (alistarh_herlihy)</i> , in synthetic benchmarks, in which we vary a) the key range, b) the number of threads that perform operations in the data structure, and c) the percentage of <i>insert/deleteMin</i> operations in the workload. . . . .	101
3.10a	Varying the key range. . . . .	101
3.10b	Varying the number of threads. . . . .	101
3.10c	Varying the operation workload. . . . .	101
3.11	Throughput achieved by <i>SmartPQ</i> , <i>Nuddle</i> and its underlying <i>base algorithm (alistarh_herlihy)</i> , in synthetic benchmarks, in which we vary multiple features in the contention workload. . . . .	102
4.1	High-level organization of an NDP architecture. . . . .	110
4.2	Slowdown of a stack data structure using a coherence-based lock over using an <i>ideal</i> zero-cost lock, when varying (a) the NDP cores within a single NDP unit and (b) the number of NDP units while keeping core count constant at 60. . . . .	112
4.3	High-level overview of <i>SynCron</i> . . . . .	114
4.4	An example execution scenario for a lock requested by <i>all</i> NDP cores. . . . .	115
4.5	Message encoding of <i>SynCron</i> . . . . .	117
4.6	The Synchronization Engine (SE). . . . .	118
4.7	Synchronization Table (ST) entry. . . . .	118
4.8	Control flow in SE. . . . .	119
4.9	Synchronization variable of <i>SynCron (synchronVar)</i> . . . . .	120
4.10	Speedup of different synchronization primitives. . . . .	126
4.11	Throughput of pointer-chasing using data structures. . . . .	127
4.12	Speedup in real applications normalized to <i>Central</i> . . . . .	128
4.13	Scalability of real applications using <i>SynCron</i> . . . . .	128
4.14	Energy breakdown in real applications for C: <i>Central</i> , H: <i>Hier</i> , SC: <i>SynCron</i> and I: <i>Ideal</i> . . . . .	129
4.15	Data movement in real applications for C: <i>Central</i> , H: <i>Hier</i> , SC: <i>SynCron</i> and I: <i>Ideal</i> . . . . .	130
4.16	Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units. . . . .	130
4.17	Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units. All data is normalized to <i>Ideal (lower is better)</i> . . . . .	131
4.18	Speedup with different memory technologies. . . . .	131
4.19	Performance sensitivity to a better graph partitioning and maximum ST occupancy of <i>SynCron</i> . . . . .	132

4.21	Speedup of <i>SynCron</i> normalized to <i>flat</i> , as we vary the transfer latency of the interconnection links used to connect NDP units, under (a) a low-contention and synchronization-intensive scenario using 4 NDP units, and (b) a high-contention scenario using 2 and 4 NDP units. . . . .	133
4.20	Speedup of <i>SynCron</i> normalized to <i>flat</i> with 40 ns link latency between NDP units, under a low-contention and synchronization non-intensive scenario. . . . .	133
4.22	Slowdown with varying ST size (normalized to 64-entry ST). Numbers on top of bars show the percentage of overflowed requests. . . . .	134
4.23	Throughput achieved by BST_FG using different overflow schemes and varying the ST size. The reported numbers show to the percentage of overflowed requests. . . . .	135
5.1	(a) CSR representation of a sparse matrix. (b) CSR-based SpMV implementation. . . .	143
5.2	(a) SpMV with a dense matrix representation, and (b) CSR, (c) COO, (d) BCSR, (e) BCOO formats. . . . .	143
5.3	High-level organization of a near-bank PIM architecture. . . . .	145
5.4	Execution of the SpMV kernel on a real PIM system. . . . .	147
5.5	Data partitioning techniques of the <i>SparseP</i> package. . . . .	147
5.6	Load balancing schemes across PIM cores for the CSR (left) and COO (right) formats with the 1D partitioning technique. The colored cells of the matrix represent non-zero elements. . . . .	149
5.7	Load balancing schemes across PIM cores for the BCSR (left) and BCOO (left) formats with the 1D partitioning technique. The cells of the matrix represent sub-blocks of size 4x4. The colored cells of the matrix represent non-zero sub-blocks, and the number inside a colored cell describes the number of non-zero elements of the corresponding sub-block. . . . .	149
5.8	The 2D partitioning techniques of <i>SparseP</i> package assuming 4 PIM cores and 2 vertical partitions. . . . .	150
5.9	Execution time achieved by various load balancing schemes of each compressed matrix format using 16 tasklets of a single DPU. . . . .	156
5.10	Scalability of all compressed formats for the int8 (top graphs) and fp64 (bottom graphs) data types as the number of tasklets of a single DPU increases. . . . .	158
5.11	Performance comparison of load balancing techniques for each particular compressed format using 2048 DPUs and the int32 data type. . . . .	160
5.12	Performance comparison of load balancing techniques for each data type using 2048 DPUs. . . . .	162
5.13	Throughput of various compressed formats using 2048 DPUs and the int32 data type. . . .	162
5.14	Performance comparison of various compressed formats using 2048 DPUs and the int32 data type. Performance is normalized to that of CSR.nnz. . . . .	162
5.15	Total execution time when using 2048 DPUs and the int32 data type for CR: CSR.nnz, CO: COO.nnz-1f, BR: BCSR.block and BO: BCOO.block kernels. . . . .	163

5.16	End-to-end execution time breakdown achieved by COO . nnz - 1 f when varying (a) the data type using 2048 DPUs (normalized to the experiment for the int8 data type), and (b) the number of DPUs for the int32 data type (normalized to 64 DPUs). . . . .	164
5.17	Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 (left) and 32 (right) vertical partitions. Performance is normalized to that of the RC scheme. . . . .	166
5.18	Execution time breakdown of <i>equally-sized</i> partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs. . . . .	167
5.19	Execution time breakdown of <i>equally-wide</i> partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs. . . . .	167
5.20	Execution time breakdown of <i>variable-sized</i> partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs. . . . .	167
5.21	Execution time breakdown of 2D partitioning schemes using the COO format and 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int8 and fp64 data types. Performance is normalized to the performance of the experiment with 1 vertical partition. . . . .	168
5.22	End-to-end execution time breakdown of the <i>equally-sized</i> 2D partitioning technique for CR: DCSR, CO: DCOO, BR: DBCSR and BO: DBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR. . . . .	170
5.23	End-to-end execution time breakdown of the <i>equally-wide</i> 2D partitioning technique for CR: RBDCSR, CO: RBDCOO, BR: RBDBCSR and BO: RBDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR. . . . .	171
5.24	End-to-end execution time breakdown of the <i>variable-sized</i> 2D partitioning technique for CR: BDCSR, CO: BDCOO, BR: BDBCSR and BO: BDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR. . . . .	171
5.25	Throughput of 2D partitioning techniques using the COO and BCOO formats, 2048 DPUs and the int32 type. . . . .	172
5.26	Performance comparison of 2D partitioning techniques using the COO and BCOO formats, 2048 DPUs and the int32 type. Performance is normalized to that of DCOO. . . . .	172
5.27	Throughput of the best-performing 1D- and 2D-partitioned kernels for the fp32 data type. . . . .	173
5.28	Performance comparison of the best-performing 1D- and 2D-partitioned kernels for the fp32 data type. Performance is normalized to that of COO . nnz - 1 f. . . . .	173



5.29	Performance comparison between the UPMEM PIM system, Intel Xeon CPU and Tesla V100 GPU on SpMV execution. . . . .	175
5.30	Energy comparison between the UPMEM PIM system, Intel Xeon CPU and Tesla V100 GPU on SpMV execution. . . . .	177
8.1	Performance of the BCOO format with various load balancing schemes and synchronization approaches for all the data types and small matrices using 16 tasklets of one DPU. . . . .	194
8.2	Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 vertical partitions. Performance is normalized to that of the RC scheme. . . . .	194
8.3	Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 32 vertical partitions. Performance is normalized to that of the RC scheme. . . . .	194
8.4	Execution time breakdown of DCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems. . . . .	196
8.5	Execution time breakdown of RBDCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems. . . . .	196
8.6	Execution time breakdown of BDCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems. . . . .	196
8.7	End-to-end execution time breakdown of the <i>equally-sized</i> 2D partitioning technique for CR: DCSR, CO: DCOO, BR: DBCSR and BO: DBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR. . . . .	196
8.8	End-to-end execution time breakdown of the <i>equally-wide</i> 2D partitioning technique for CR: RBDCSR, CO: RBDCOO, BR: RBDBCSR and BO: RBDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR. . . . .	197
8.9	End-to-end execution time breakdown of the <i>variable-sized</i> 2D partitioning technique for CR: BDCOO, CO: BDCOO, BR: BDBCSR and BO: BDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR. . . . .	197

8.10	Throughput of 1D- and 2D-partitioned kernels for the fp32 data type using two different UPMEM PIM systems. . . . .	197
8.11	Performance comparison of 1D- and 2D-partitioned kernels for the fp32 data type using two different UPMEM PIM systems. Performance is normalized to that of COO . nnz - 1 f (A). . . . .	198
8.12	Throughput of the MUL operation on one DPU at 350 MHz for all the data types. . .	199
8.13	Throughput of the MUL operation on one DPU at 425 MHz for all the data types. . .	200

---

# List of Tables

---

2.1	Large Real-World Graph Dataset. . . . .	65
2.2	The geometric mean on the number of colors produced across all large real-world graphs (lower is better) for each parallel graph coloring implementation using one core (1 thread), all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads). . . . .	66
2.3	Color balancing quality achieved by <i>ColorTM</i> and all balanced graph coloring implementations in the large real-world graphs. We present the relative standard deviation (in %) on the sizes of the color classes obtained by each scheme (lower is better). In <i>ColorTM</i> and Recoloring, we provide inside the parentheses the number of color classes produced. The CLU, VFF and <i>BalColorTM</i> produce the same number of color classes with the initial coloring scheme. . . . .	73
3.1	The features of the contention workload which are used for classification. . . . .	95
3.2	Features of the contention workload for benchmarks evaluated in Figure 3.10. We use bold font on the features that change in each execution phase. . . . .	101
3.2a	Varying the key range in the workload. . . . .	101
3.2b	Varying the number of threads that perform operations in the data structure. . . . .	101
3.2c	Varying the percentage of <i>insert/deleteMin</i> operations. . . . .	101
3.3	Features of the contention workload for benchmarks evaluated in Figure 3.11. We use bold font on the features that change in each execution phase. . . . .	102
4.1	Throughput of two coherence-based lock algorithms on an Intel Xeon Gold server using the liblock library [16]. . . . .	112
4.2	<i>SynCron</i> 's Programming Interface (i.e., API). . . . .	116

4.3	Message opcodes of <i>SynCron</i> . . . . .	117
4.4	Comparison of <i>SynCron</i> with prior mechanisms. . . . .	123
4.5	Configuration of our simulated system. . . . .	124
4.6	Summary of all workloads used in our evaluation. . . . .	125
4.7	ST occupancy in real applications. . . . .	134
4.8	Comparison of SE with a simple general-purpose in-order core, ARM Cortex A7. . . .	135
5.1	Parallelization techniques across PIM cores of the <i>SparseP</i> library. *: row-granularity, †: block-row-granularity . . . . .	151
5.2	Parallelization schemes across threads of a PIM core. *: row-granularity, †: block-row-granularity . . . . .	153
5.3	Small Matrix Dataset. . . . .	154
5.4	Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern. The highlighted matrices with red color exhibit block pattern [17, 18]. . . . .	155
5.5	Evaluated CPU, GPU, and UPMEM PIM Systems. . . . .	174
8.1	Evaluated UPMEM PIM Systems. . . . .	195
8.2	The <i>SparseP</i> library. *: row-granularity, †: block-row-granularity, ‡: (only for 8-bit integer and small block sizes) . . . . .	201
8.3	Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern. . . . .	202

# CHAPTER 1

---

## Introduction

---

Irregular applications such as graph processing, data analytics, sparse linear algebra and dynamic pointer-chasing constitute an important part of software systems we rely on. These applications lie at the heart of many important workloads including deep neural networks [19–24], bioinformatics [25–27], databases [28, 29], data analytics [29–37], large-scale simulations [38–43], medical imaging [18, 40, 44], economic modeling [18, 40, 44], and scientific applications [18, 40–44]. Therefore, optimizing and accelerating irregular applications is of vital importance, and thus a large corpus of research proposes either software designs [1, 13–15, 18, 44–152] or hardware mechanisms [5, 10, 20, 22, 25–29, 32–37, 39, 129, 153–288] to accelerate the execution of such applications.

In this dissertation, we identify three important characteristics of irregular applications that critically affect their performance. First, irregular applications exhibit *inherent imbalance* as a result of the real-world input data sets given. Specifically, the concrete pieces involved in the underlying data structures and program data of irregular applications are *not* of equal size. For example, the matrices involved in linear algebra kernels are very sparse, i.e., the vast majority of elements are zeros [18, 103, 139, 150, 286, 289–294], and in most real-world matrices the number of non-zero ele-

ments per row shows high disparity and imbalance across the rows of the matrix [295]. Similarly, the real-world graphs involved in graph processing workloads typically have a power-law distribution, i.e., only a *few* vertices have a very *high* adjacency degree, while the vast majority of the remaining vertices of the graph have a very low adjacency degree [10, 109, 155], which causes high disparity and imbalance in the number of edges across vertices. Therefore, *naively* parallelizing such workloads to a large number of threads in modern computing platforms can incur 1) high load imbalance across parallel threads, and 2) high disparity in the amount of computation versus memory accesses executed across parallel threads. Second, irregular applications exhibit *random memory access patterns*, i.e., the memory accesses performed are neither sequential nor strided, and they are *input-driven* dependent. Such complex memory access patterns are very hard to predict. Therefore, irregular applications exhibit complex data dependencies, poor spatial and temporal data locality, and high data movement overheads to transfer data between memory and processors of commodity computing systems. Third, most irregular applications have *low operational intensity*, i.e., the amount of useful arithmetic operations performed by the processors compared to the amount of data necessary to perform these operations is very low. As a result, irregular applications are memory-bound kernels. They can be significantly bottlenecked by the memory subsystem, incurring high latency costs and excessive memory bandwidth consumption to access data through memory.

As such, irregular applications constitute an important and emerging workload domain. However, at the same time, it is very challenging to achieve high performance and energy efficiency in the execution of such workloads in modern computing systems due to the large memory and communication bottlenecks. Overall, irregular applications have several important inherent characteristics that necessitate new approaches both in the software, i.e., re-designing parallel algorithms, and the hardware level, i.e., re-designing key components of modern computing architectures, to achieve high system performance, and cooperatively between the software and the hardware.

## 1.1 Motivation: Excessive Synchronization and High Memory Intensity Degrade the Execution of Irregular Applications

Modern computing systems and state-of-the-art parallel algorithms have two important implications that render the efficient execution of irregular applications a significantly challenging task.

**Implication 1: Excessive Synchronization.** To achieve high system performance in a multi-threaded execution context, load balance among parallel threads is critical. Therefore, software engineers employ a *fine-grained* parallelization strategy among parallel threads in irregular applications due to the inherent imbalance exhibited in the input data sets involved. For example, Figure 1.1 compares a *regular* Dense Matrix Vector Multiplication (DEMv) with an *irregular* Sparse Matrix Vector Multiplication (SpMV). In the DEMv execution, a coarse-grained parallelization strategy (Figure 1.1a), in which the rows of the matrix are equally distributed across parallel threads, can easily achieve high load balance. However, using a coarse-grained parallelization strategy to parallelize the irregular SpMV kernel (Figure 1.1b) results in significantly high load imbalance among parallel threads, due to the high disparity in the number of non-zero elements processed across parallel threads, causing

large performance overheads. Thus, a fine-grained parallelization strategy among parallel threads is necessary, e.g., Figure 1.1c. Unfortunately, this approach however results in excessive and frequent synchronization among parallel threads. In the example of the irregular SpMV kernel, with a fine-grained parallelization strategy, parallel threads that process non-zero elements of the *same* row of the matrix (Figure 1.1c), use synchronization primitives (e.g., locks, mutexes) to ensure atomicity and correctness, when performing write updates on the *same* element of the output vector. Therefore, a large amount of processor cycles is spent on communication and synchronization with significant performance overheads [16, 44, 47, 291, 296].

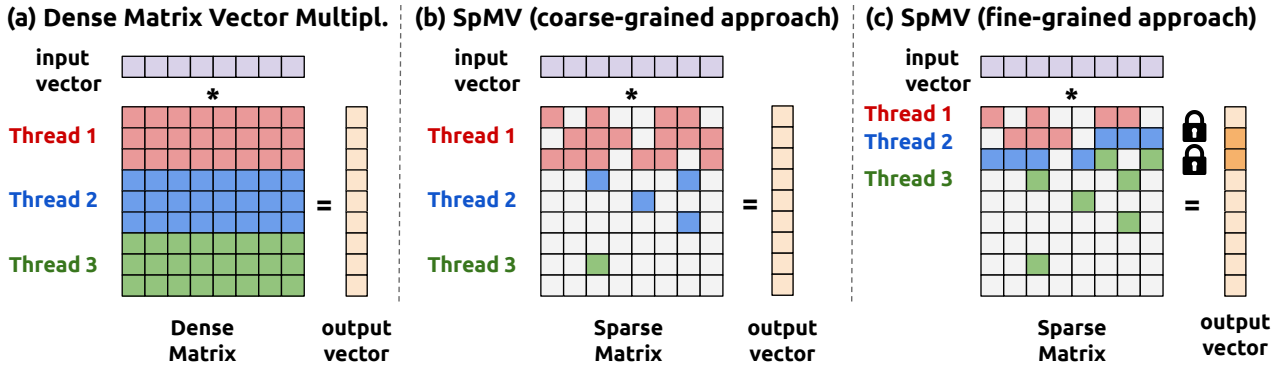


Figure 1.1: (a) Dense Matrix Vector Multiplication using three parallel threads. (b) Sparse Matrix Vector Multiplication with a coarse-grained parallelization strategy among three parallel threads. (c) Sparse Matrix Vector Multiplication with a fine-grained parallelization strategy among three parallel threads. The colored cells of each matrix represent non-zero elements.

At the application level, existing fine-grained parallel algorithms (e.g., [13, 59–68, 77, 78]) typically *lack* well-tuned synchronization implementations [45, 297], and/or do *not* achieve high system performance under all *various* contention scenarios. Recent works [16, 45, 297, 298] demonstrate that (i) naive synchronization schemes used in irregular applications can cause high memory traffic with significant latency access costs, and (ii) the best-performing synchronization scheme varies depending on the levels of contention among parallel threads and the characteristics of the underlying hardware platform. At the architecture level, even though numerous hardware synchronization mechanisms have been proposed [299–317], most of them incur high hardware cost to be implemented in commodity systems, require important cross-stack modifications and/or have narrow programming interfaces, and thus they are hard to adopt.

**Implication 2: High Memory Intensity.** Irregular applications involve random memory access patterns, have low operational intensity and are primarily bottlenecked by the memory subsystem [18, 21, 29, 32–37, 44, 103, 156, 162, 291, 294]. Thus, irregular applications incur high memory intensity with significant data access costs, and a large fraction of the application’s execution time is spent on data accesses and/or waiting for data to be transferred between memory and processors. Things become even worse with the large growth in input data set sizes as well as intermediate data used and generated during runtime. Therefore, irregular applications need to process increasingly large volumes of data (input data sets with tens or hundreds of GBs memory footprints [46, 318]), and need to effectively handle the high data demand.

We demonstrate the aforementioned critical performance implication with an example, i.e., the SpMV kernel execution. The SpMV kernel performs  $\mathcal{O}(NNZ)$  operations on  $\mathcal{O}(N + NNZ)$  amount of data (assuming a square matrix), where  $NNZ$  is the number of non-zero elements of the input matrix and  $N$  is the number of columns of the input matrix (equal to the number of elements of the input vector). However, the real-world matrices involved are very sparse [18, 103, 150, 286, 289–293]. For instance, the matrices that represent Facebook’s and YouTube’s network connectivity contain *only* 0.0003% [286, 289] and 2.31% [286, 290] non-zero elements, respectively. Figure 1.2 presents an example SpMV execution on the first four rows of a sparse  $9 \times 9$  matrix with only 10 non-zero elements, i.e., having  $\sim 0.17$  operational intensity when assuming single precision non-zero elements (i.e., integers). As shown in Figure 1.2, the accesses to the elements of the input vector are random and irregular, and they depend on the sparsity pattern of the matrix that is given as input. The data accesses to the elements of the input vector are very hard to predict, since they are affected by the particular characteristics of the input matrix, and are typically performed using the main memory of commodity systems, which often has high latency and low bandwidth [244, 319]. Thus, SpMV execution is highly limited by the irregular data accesses to the elements of the input vector and the data movement costs of accessing the elements of the input vector, which cause significant performance overheads in the total execution time [18, 103, 111, 120, 132, 139, 286, 292, 294, 320–323].

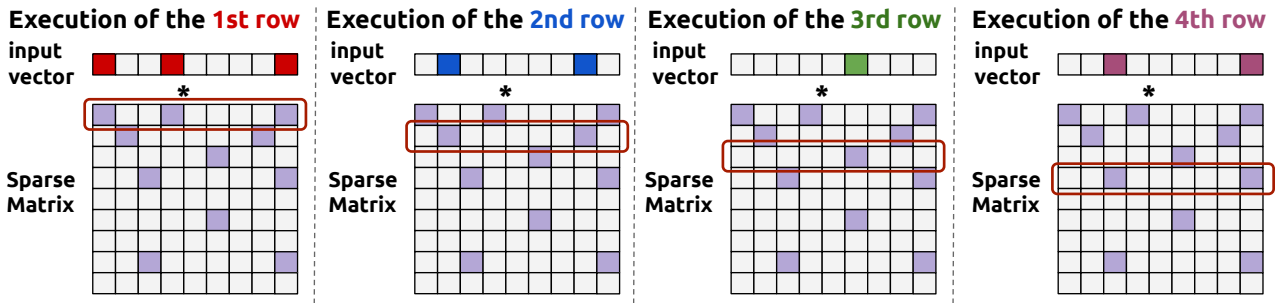


Figure 1.2: An example SpMV execution on the first four rows of a sparse  $9 \times 9$  matrix with only 10 non-zero elements. The execution steps are performed at a row granularity. The colored cells of the matrix with purple color represent non-zero elements, and the colored cells of the input vector represent the elements of the input vector that are processed/accessed at each execution step.

Two recent works [21, 324] explain that the energy overheads of data movement across the memory hierarchy of commodity systems can be significantly higher than that of computation in irregular applications. First, Boroumand et al. [21] show that moving data between memory and processors causes more than 60% of the total system energy efficiency in several irregular Google’s consumer workloads. Second, Boroumand et al. [324] demonstrate that the commercial Google Edge TPU unit [325] spends 50.3% of its total energy on off-chip memory accesses across a wide range of irregular machine learning applications, including convolutional neural networks, transducers and recurrent neural networks. Multiple other works (e.g., [326, 327]) provide analysis of data movement bottlenecks in a variety of irregular workloads. Therefore, we conclude that the high memory intensity of irregular applications causes significant bottlenecks and high overheads both in performance and energy consumption.

At the application level, many parallel applications and software packages do *not* handle data well



(e.g., [13, 49, 53–56, 59, 60, 77, 78]), and/or do *not* adapt their parallelization strategies to the particular characteristics of the input data given. Recent works [328–333] highlight that different pieces of program data have different performance characteristics (latency/bandwidth/parallelism sensitivity), and inherent properties. Consequently, *data-oblivious* policies, i.e., policies that are designed without taking into consideration the properties of the application data they handle, result in lost performance optimization opportunities, which could be achieved by exploiting data properties. Similarly, at the architecture level, existing hardware mechanisms (e.g., [334–337]) are designed without considering modern applications’ memory access patterns and overwhelming data demand, and as such, they cause frequent data movement across the entire system and significant data access costs.

## 1.2 Our Approach: Efficient Synchronization and Data Access Techniques for Irregular Applications

In this dissertation, we study a wide range of irregular applications, including graph processing, data analytics, pointer-chasing and sparse linear algebra, and explore their performance implications on two modern computing platforms: (i) the *processor-centric Non-Uniform Memory Access (NUMA) CPU* architectures, and (ii) the *memory-centric Processing-In-Memory (PIM)* (or Near-Data-Processing (NDP)) architectures. The NUMA CPU architectures constitute the dominant hardware platform in today’s computing systems, and have been significantly optimized over decades to integrate general-purpose cores with high computation capability. The PIM/NDP architectures have been recently commercialized [156–158, 161, 162, 338–341], and represent a promising disruptive paradigm to alleviate the costs of data movement across the memory hierarchy. PIM/NDP architectures equip memory chips with a large number of low-area and low-power cores with relatively low computation capability, and alleviate data movement overheads by performing computation close to where the application data resides. Therefore, PIM/NDP architectures provide high levels of parallelism and very large memory bandwidth.

We posit that, moving forward, both hardware mechanisms and parallel algorithms need to consider the synchronization needs and memory access patterns of irregular applications as the two major priorities to significantly improve system performance and system energy efficiency, when employing hundreds or thousands of parallel threads. In particular, modern software and hardware designs for irregular applications should provide two major types of optimization approaches: (1) efficient synchronization, and (2) efficient data access techniques.

**Efficient Synchronization Techniques.** Modern computing platforms need to support low-cost and practical hardware synchronization mechanisms, while parallel algorithmic designs need to provide fine-grained communication and adaptive synchronization approaches among parallel threads to significantly accelerate the execution of irregular applications. Lightweight synchronization techniques are highly effective at execution of irregular applications, since they improve system performance and energy efficiency by (1) mitigating coherence/communication traffic overheads caused when synchronizing hundreds or thousands of parallel threads, and (2) exposing high levels of fine-grained parallelism thanks to enabling low-cost communication and coordination among parallel

threads. We demonstrate the benefits of efficient synchronization in four different contexts. First, we design a *speculative* synchronization scheme for the widely used graph coloring kernel [48] (Chapter 2), which *speculatively* performs most computations and data accesses *outside* the critical section, and thus effectively minimizes synchronization costs and provides high levels of parallelism on the graph coloring kernel by enabling short critical sections with small memory footprints. Second, we propose an *adaptive* concurrent priority queue (Chapter 3), which dynamically tunes its parallelization strategy between two algorithmic modes (a NUMA-aware and a NUMA-oblivious mode) *without* using barrier synchronization, and thus achieving *negligible* synchronization costs upon transitions. Third, we introduce a low-cost and practical hardware synchronization mechanism tailored for NDP architectures (Chapter 4), which significantly improves system performance and system energy efficiency in a wide variety of irregular parallel applications including pointer chasing, graph processing, and time series analysis. Fourth, we implement three synchronization schemes among parallel threads of a multithreaded PIM core in the SpMV kernel (Chapter 5), and show that the *lock-free* synchronization scheme provides significant performance benefits over the *lock-based* synchronization schemes in a real PIM system, by providing higher amount of parallelism among parallel threads.

**Efficient Data Access Techniques.** Modern computing systems need to eliminate data movement overheads, while parallel algorithmic designs need to provide well-crafted data distribution and data-aware parallelization strategies (by exploiting properties of data), as well as adaptive cache and memory management techniques (by leveraging characteristics of the underlying hardware), in order to minimize data access costs in the execution of irregular applications. *Data-aware* parallel algorithms and *memory-centric* architectures can significantly improve performance and energy efficiency in the execution of irregular applications by (1) reducing data access and communication costs, and (2) better leveraging the available memory bandwidth of the underlying hardware to increase the efficiency of the application execution. We demonstrate the benefits of efficient data access techniques in four different contexts. First, we propose an *eager* coloring conflict policy to detect and resolve coloring inconsistencies arising among parallel threads in the graph coloring kernel [48] (Chapter 2), which effectively reduces data access costs by accessing conflicted vertices *immediately* using the low-cost on-chip caches of multicore CPU platforms. Second, we design (i) a concurrent priority queue (Chapter 3) having a parallelization strategy that is aware of the non-uniform distribution (NUMA-aware) of the underlying data structure in a NUMA CPU architecture, and thus achieves higher performance benefits (by minimizing data access costs) in high-contention scenarios over state-of-the-art concurrent priority queues [13, 77], which are oblivious to the non-uniform distribution (NUMA-oblivious) of the underlying data structure in a NUMA CPU architecture, and we also propose (ii) an *adaptive* concurrent priority queue (Chapter 3), which *dynamically* tunes its parallelization strategy between a NUMA-aware and a NUMA-oblivious algorithmic mode depending on the contention levels of the current workload, and thus provides high throughput in NUMA CPU systems under all various contention scenarios (by reducing data access costs and achieving high amount of parallelism). Third, we add a specialized low-cost cache memory structure inside synchronization accelerators for NDP systems (Chapter 4) to *directly* buffer synchronization variables, and thus minimize latency access costs by avoiding costly memory accesses for synchronization. Fourth, we introduce various data

partitioning techniques to efficiently map the irregular SpMV execution kernel on near-bank PIM systems [157, 338, 340] (Chapter 5), and show that the best-performing SpMV execution on a near-bank PIM system [157, 338] (Chapter 5) is achieved using intelligent *data-aware* algorithmic designs that (i) trade off computation for lower data movement overheads, and (ii) select their parallelization strategy and data partitioning policy based on the particular sparsity pattern of the input matrix, i.e., by exploiting properties of the input data. We also observe that the memory-bound SpMV kernel on a state-of-the-art *memory-centric* PIM system achieves a much higher fraction of the machine's peak performance compared to that on state-of-the-art *processor-centric* CPU and GPU systems (Chapter 5).

### 1.2.1 Thesis Statement

In this dissertation, we propose parallelization techniques and algorithmic designs, along with hardware mechanisms that enable lightweight synchronization and low-cost data accesses in emerging irregular applications running in processor-centric CPU and memory-centric PIM/NDP architectures. Specifically, we propose (1) a novel parallel algorithm that minimizes synchronization and data access costs in the graph coloring kernel execution on CPU systems, (2) an adaptive concurrent priority queue that provides high amount of parallelism and minimizes memory traffic in NUMA CPU systems, (3) an end-to-end hardware mechanism that enables low-cost synchronization in NDP systems, and (4) several efficient algorithmic designs that provide low synchronization and data transfer costs in the SpMV kernel execution on real near-bank PIM systems.

This dissertation, hence, provides substantial evidence for the following thesis statement:

*Low-overhead synchronization approaches in cooperation with well-crafted data access techniques can significantly improve performance and energy efficiency of important data-intensive irregular applications.*

## 1.3 Overview of Our Research

We propose four new approaches to accelerate irregular applications in CPU and PIM/NDP architectures via efficient synchronization and data access techniques, which we briefly describe next. We also put our contributions in the context of relevant prior work and provide detailed discussions of and comparisons to prior work in Chapters 2- 5.

### 1.3.1 *ColorTM* [1–3]: High-Performance and Balanced Parallel Graph Coloring on Multicore CPU Platforms (Chapter 2)

Graph coloring is an important graph processing kernel, and it is widely used in many real-world end-applications including the conflicting job scheduling [48, 342–345], register allocation [346–350] and sparse linear algebra [351–354]. The total runtime of the graph coloring kernel typically adds to the overall parallel overhead of the real-world end-application, and thus a high-performance parallel graph coloring algorithm for modern multicore platforms is necessary. Prior works [49, 53–56] that parallelize the graph coloring kernel are still inefficient (as we demonstrate in Chapter 2), because they

detect and resolve the coloring inconsistencies arisen among parallel threads with a *lazy* approach: they detect and resolve the coloring inconsistencies much later in the runtime compared to the time that the coloring inconsistencies appeared. As a result, prior approaches access the conflicted vertices of the graph multiple times, mainly using the expensive last levels of the memory hierarchy (e.g., main memory) of commodity multicore platforms, thus incurring high data access costs.

To this end, we design *ColorTM* [1–3], a high-performance parallel graph coloring algorithm for multicore platforms. *ColorTM* is designed to provide both low synchronization overheads and low data access costs via two key techniques. First, we introduce an *eager* conflict detection and resolution approach of the coloring inconsistencies arisen among parallel threads: coloring inconsistencies are immediately detected and resolved at the time they appear. This way in *ColorTM*, the conflicted vertices are accessed multiple times, using the low-cost lower levels of the memory hierarchy of multicore platforms, thus achieving low data access costs. Second, we design a *speculative* computation and synchronization scheme, in which parallel threads speculatively perform computations and data accesses *outside* the critical section to enable short critical sections with small memory footprints. This key technique provides high levels of parallelism and low synchronization costs by executing multiple small and short critical sections in parallel. Next, we extend our algorithmic design to propose a *balanced* parallel graph coloring algorithm, named *BalColorTM* [2], in which the vertices of the graph are *almost equally* distributed across the color classes produced. Enabling color classes with almost equal sizes can provide high hardware resource utilization and high load balance among parallel threads in the real-world end-applications of graph coloring.

We evaluate *ColorTM* and *BalColorTM* on a modern multicore platform using a wide variety of large real-world graphs with diverse characteristics. In Chapter 2, we show that *ColorTM* and *BalColorTM* significantly outperform prior state-of-the-art graph coloring algorithms [49, 53–56], while also achieving high coloring quality. We also demonstrate that *ColorTM* and *BalColorTM* can provide significant performance improvements in real-world end-applications, e.g., Community Detection [355]. *ColorTM* and *BalColorTM* are freely and publicly available [2] at [github.com/cgiannoula/ColorTM](https://github.com/cgiannoula/ColorTM) to enable further research on the graph coloring kernel in modern multicore systems.

### 1.3.2 *SmartPQ* [4]: An Adaptive Concurrent Priority Queue for NUMA CPU Architectures (Chapter 3)

Concurrent priority queues lie at the heart of many important applications including graph processing [356–360] and discrete event simulations [361–363]. Prior works [13, 15, 59, 60, 62–65, 77, 78, 86, 364] have designed concurrent priority queues for modern NUMA architectures. These implementations for concurrent priority queues are typically of two types: (i) NUMA-oblivious concurrent priority queues [13, 59, 60, 62–65, 77, 78, 364], in which the parallelization strategy implemented is *oblivious* to the non-uniform memory access costs of the underlying memory subsystem, and (ii) NUMA-aware concurrent priority queues [15, 86], in which the parallelization strategy implemented takes into consideration the non-uniform memory access costs of the underlying memory subsystem. We examine prior state-of-the-art concurrent priority queues [13, 15, 77] on a NUMA CPU system using a wide variety of contention scenarios, and find that none of the prior state-of-the-art concurrent priority

queues performs best across all various contention scenarios. Specifically, NUMA-oblivious concurrent priority queues provide high levels of parallelism, low data access costs, and high performance in *insert*-dominated scenarios, which typically exhibit low-contention, since parallel threads may work on different parts of the underlying data structure. In contrast, NUMA-oblivious concurrent priority queues cause high data movement traffic in the memory subsystem of a NUMA architecture, and incur significant performance slowdowns over the NUMA-aware concurrent priority queues in *deleteMin*-dominated workloads, which exhibit very high contention, since parallel threads highly compete to remove the first few elements of the underlying data structure.

To this end, we propose *SmartPQ* [4], an adaptive concurrent priority queue for NUMA architectures that achieves the highest performance in all different contention scenarios, and even when the contention of the workload varies over time. *SmartPQ* is designed to provide high levels of parallelism and low data access and data movement costs under all various contention scenarios. To achieve this, *SmartPQ* *dynamically* adapts itself over time between a NUMA-oblivious and a NUMA-aware algorithmic mode depending on the contention levels of the workload. *SmartPQ* integrates (i) *NUMA Node Delegation (Nuddle)*, a *generic* framework to wrap *any* arbitrary NUMA-oblivious concurrent data structure, and transform it into its NUMA-aware counterpart, and (ii) a simple decision tree *classifier*, which predicts the best-performing algorithmic mode (between a NUMA-oblivious and a NUMA-aware algorithmic mode) given the current contention levels of the workload. This way *SmartPQ* uses the NUMA-aware *Nuddle* priority queue in *deleteMin*-dominated workloads, and switches to *directly* using the *Nuddle*'s underlying NUMA-oblivious concurrent priority queue in *insert*-dominated scenarios, thus enabling low data access costs in all various contention scenarios.

We evaluate *SmartPQ* in a modern NUMA CPU system using a wide range of contention scenarios, and also using synthetic benchmarks that *dynamically* vary the contention of the workload over time. In Chapter 3, we demonstrate that *SmartPQ* achieves the highest performance over prior state-of-the-art NUMA-oblivious and NUMA-aware concurrent priority queues [13, 15, 77] in *all various* contention scenarios and at *any* point in time with 87.9% success rate.

### 1.3.3 *SynCron* [5]: Efficient Synchronization Support for NDP Architectures (Chapter 4)

NDP architectures [32, 36, 158, 160, 268, 365] alleviate the expensive data movement between processors and memory by performing computation close to where the application data resides. Typical NDP architectures support several NDP units connected to each other, with each unit comprising multiple NDP cores close to memory [21, 32, 34, 35, 366–368]. Therefore, NDP architectures provide high levels of parallelism, low memory access latency, and large aggregate memory bandwidth, thereby being a very good fit to accelerate irregular applications such as genome analysis [25, 27], graph processing [29, 32–37], databases [28, 29], pointer-chasing workloads [76, 215, 216, 369], and sparse neural networks [21–24]. However, to fully leverage the benefits of NDP architectures for these irregular parallel applications, an effective synchronization solution for NDP systems is necessary.

Numerous prior works [299–308, 315–317, 370–380] propose synchronization solutions for *processor-centric* CPU, GPU and Massively Parallel Processing (MPP) systems. However, these synchronization

solutions are not efficient or suitable for the *memory-centric* NDP systems (Chapter 4), which are fundamentally different from commodity *processor-centric* systems. First, synchronization approaches for CPU systems are typically implemented upon the underlying hardware cache coherence protocols, but most NDP systems do *not* support hardware cache coherence (e.g., [32, 34, 35, 159, 367]). Second, synchronization in GPUs and MPPs is implemented in dedicated hardware atomic units, known as *remote atomics*. However, synchronization using remote atomics has been shown to be inefficient, since it causes high global traffic and hotspots [153, 370, 371, 381, 382]. Finally, prior hardware synchronization mechanisms [299–301, 303–305, 307, 308] tailored for commodity processor-centric systems are not suitable for memory-centric NDP systems, because they would either incur high hardware costs to be implemented in large-scale NDP systems (e.g., [301, 303–305]) or cause excessive network traffic across the NDP units of the system with significant performance overheads upon contention (e.g., [299, 300, 307, 308]).

To this end, we design *SynCron* [5], a low-overhead hardware synchronization mechanism tailored for memory-centric NDP architectures. *SynCron* consists of four key techniques. First, we offload synchronization among NDP cores to dedicated low-cost hardware units to avoid the need for complex coherence protocols and expensive atomic operations. Second, we directly buffer the synchronization variables in a specialized cache memory structure to avoid costly memory accesses for synchronization. Third, we coordinate synchronization among NDP cores of several NDP units via a hierarchical message-passing scheme to minimize synchronization traffic across NDP units of the system under high-contention scenarios. Fourth, when applications with frequent synchronization oversubscribe the hardware synchronization resources, we use an efficient and programmer-transparent overflow management scheme to avoid costly fallback solutions and minimize overheads.

In Chapter 4, we demonstrate that *SynCron* achieves the goals of performance, cost, programming ease, and generality by covering a wide range of synchronization primitives. In addition, we show that *SynCron* significantly improves system performance and energy efficiency across a wide range of irregular applications including pointer-chasing, graph applications, and time series analysis, while also has low area and power overheads to be integrated into the compute die of NDP units.

### 1.3.4 *SparseP* [6–11]: Towards Efficient Sparse Matrix Vector Multiplication on Real PIM Architectures (Chapter 5)

The SpMV kernel has been characterized as one of the most thoroughly studied scientific computation kernels [18, 291], and is a fundamental linear algebra kernel for important applications from the scientific computing, machine learning, graph analytics and high performance computing domains. In commodity processor-centric systems like CPU and GPU systems, SpMV is a memory-bandwidth-bound kernel for the majority of real sparse matrices, and is bottlenecked by data movement between memory and processors [17, 18, 42, 44, 103, 111, 120, 146, 161, 162, 272, 291, 294, 320–323, 383, 384]. To alleviate the data movement bottleneck, several manufacturers have already started to commercialize near-bank PIM architectures [157, 161, 162, 338–341, 385–398], after decades of research efforts. *Near-bank* PIM designs tightly couple a PIM core with each DRAM bank, exploiting bank-level parallelism to expose high on-chip memory bandwidth of standard DRAM to processors. Two *real* near-bank PIM

architectures are Samsung’s FIMDRAM [340, 341] and the UPMEM PIM system [157, 161, 162, 399].

Recent works leverage near-bank PIM architectures to provide high performance and energy benefits on bioinformatics [161, 162, 400, 401], skyline computation [402], compression [403] and neural network [161, 162, 340, 385, 387] kernels. A recent study [161, 162] provides PrIM benchmarks [404], which are a collection of 16 kernels for evaluating near-bank PIM architectures, like the UPMEM PIM system. Similarly, a recent work [405] implements several machine learning kernels, i.e., linear regression, logistic regression, decision tree, k-means clustering, on the UPMEM PIM system to understand the potential of a modern general-purpose PIM architecture to accelerate machine learning training. However, there is *no* prior work to efficiently map the SpMV execution kernel on near-bank PIM systems, and thoroughly study the widely used, memory-bound SpMV kernel on a real PIM system.

To this end, we design efficient SpMV algorithms tailored for current and future real PIM systems, which are publicly available in the *SparseP* software package [6–11]. The *SparseP* software package includes 25 SpMV kernels for real PIM systems, which are designed to provide high levels of parallelism, low synchronization costs, low data movement overheads, as well as to effectively leverage the immense memory bandwidth supported in near-bank PIM architectures. Specifically, *SparseP* supports (1) the four most popular compressed matrix formats, (2) a wide range of data types, (3) two types of well-crafted data partitioning techniques of the sparse matrix to DRAM banks of PIM-enabled memory modules, (4) various load balancing schemes across thousands of PIM cores, (5) various load balancing schemes across several threads of a multithreaded PIM core, and (6) three synchronization approaches among threads within multithreaded PIM core.

We conduct an extensive and comprehensive study of *SparseP* kernels on the memory-centric UPMEM PIM system [156, 157, 161, 162], the first publicly-available real-world PIM architecture. In Chapter 5, we analyze the SpMV execution (1) using one single multithreaded PIM core, (2) using thousands of PIM cores, and (3) comparing its performance and energy consumption with that achieved on processor-centric CPU and GPU systems. Based on our rigorous experimental results and observations, we provide programming recommendations for software designers and suggestions for hardware and system designers of future PIM systems. Our *SparseP* software package is freely and publicly available at <https://github.com/CMU-SAFARI/SparseP> to enable further research on the irregular SpMV kernel in current and future PIM systems.

## 1.4 Contributions

This dissertation explores lightweight synchronization approaches in cooperation with efficient data access techniques to accelerate irregular applications both in processor-centric CPU systems and memory-centric NDP/PIM systems. This doctoral thesis aims to bridge the gap between processor-centric CPU systems and memory-centric PIM systems in the critically-important area of irregular applications. Based on our rigorous experimental results and observations, we provide programming recommendations for software designers and suggestions for hardware and system designers of CPU and NDP/PIM systems in Chapters 2– 5. In summary, this dissertation makes the following major contributions:

- We introduce *ColorTM*, a novel algorithmic design to accelerate the widely used irregular graph coloring kernel on modern multicore CPU platforms. We introduce a *speculative* synchronization and computation approach on graph coloring to mitigate synchronization overheads. We propose an *eager* detection and resolution policy of the coloring inconsistencies arising among parallel threads to minimize data access costs. We extend our algorithmic design to present *BalColorTM*, an efficient *balanced* graph coloring kernel, which produces color classes with almost equal sizes. We demonstrate the effectiveness of *ColorTM* and *BalColorTM* at significantly outperforming prior state-of-the-art parallel graph coloring algorithms, and providing high performance benefits on a real-world end-application using a wide variety of large real-world graphs with diverse characteristics. Chapter 2 describes *ColorTM* and *BalColorTM* and their evaluations in detail.
- We propose *SmartPQ*, an adaptive concurrent priority queue for NUMA CPU architectures. We introduce *Nuddle*, a generic technique to obtain efficient NUMA-aware concurrent data structures by wrapping any arbitrary NUMA-oblivious concurrent data structure. We design the adaptive *SmartPQ* that uses the NUMA-aware *Nuddle* concurrent priority queue under high-contention scenarios, and switches to *directly* using the *Nuddle*'s underlying NUMA-oblivious concurrent priority queue under low-contention scenarios. This way *SmartPQ* provides high levels of parallelism, low data access costs, and significant performance benefits in modern NUMA CPU systems under all various contention scenarios, and even when the contention of the workload varies over time. We show the effectiveness of *SmartPQ* at providing significant performance benefits over prior state-of-the-art NUMA-aware and NUMA-oblivious concurrent priority queues under various contention scenarios. Chapter 3 describes *SmartPQ* and its evaluations in detail.
- We introduce *SynCron*, the first end-to-end hardware synchronization mechanism for NDP architectures. *SynCron* provides low-overhead synchronization in the execution of irregular applications on NDP systems, has low hardware costs, supports many synchronization primitives, and implements an easy-to-use high-level synchronization interface. We design low-cost synchronization units that coordinate synchronization across NDP cores, and directly buffer synchronization variables in a specialized cache memory to avoid costly memory accesses to them. We integrate an efficient hierarchical message-passing synchronization scheme, and hardware-only programmer-transparent overflow management to mitigate performance overheads when hardware resources are exceeded. We demonstrate the effectiveness of *SynCron* at significantly improving system performance and system energy efficiency using a wide range of irregular parallel applications, including pointer-chasing, graph processing, and time series analysis, and under various contention scenarios. Chapter 4 describes *SynCron* and its evaluations in detail.
- We propose *SparseP*, the first open-source software package of 25 efficient SpMV kernels tailored for real near-bank PIM architectures. We support several well-crafted data partitioning techniques of the sparse matrix to PIM-enabled memory and various load balancing schemes across PIM cores and across threads of a multithreaded PIM core to trade off computation bal-



ance across PIM cores for lower data transfer costs to PIM-enabled memory. We include three different synchronization approaches among several threads within a multithreaded PIM core to minimize synchronization overheads and achieve high levels of parallelism. We perform the first comprehensive study of the widely used irregular SpMV kernel on the UPMEM PIM architecture, the first real commercial PIM architecture, using various compressed matrix storage formats, many data types, and 26 sparse matrices with diverse sparsity patterns. We demonstrate that the SpMV execution on the memory-centric UPMEM PIM system with 2528 PIM cores achieves a much higher fraction of the machine’s peak performance compared to that on the state-of-the-art processor-centric CPU and GPU systems, and also provides high energy efficiency. Chapter 5 describes *SparseP* and its evaluations in detail.

## 1.5 Outline

This dissertation is organized into 8 chapters. Chapter 1 describes and motivates the thesis statement of this dissertation, and also briefly describes the research contributions of this dissertation. Chapter 2 introduces *ColorTM*, a new algorithmic design to accelerate the irregular graph coloring kernel in modern CPU architectures, and presents its experimental study on a modern multicore platform. Chapter 3 introduces *SmartPQ*, an adaptive concurrent priority queue that achieves high performance in NUMA CPU architectures under all various contention scenarios, and presents its respective evaluations. Chapter 4 introduces *SynCron*, an end-to-end hardware mechanism to support efficient and low-cost synchronization in NDP systems, and presents its evaluations across a wide variety of irregular applications including graph processing, pointer-chasing and time series analysis. Chapter 5 introduces *SparseP*, the first open-source library of 25 algorithms to efficiently execute the irregular SpMV kernel on real PIM architectures, and presents a comprehensive experimental study of these SpMV kernels on the first real commercial PIM architecture. Chapter 6 presents future research directions and concluding remarks of this dissertation. Chapter 7 presents several other research works of the author of this dissertation. Chapter 8 presents additional experimental results and descriptions for the *SparseP* contribution (Chapter 5).



# CHAPTER 2

---

## *ColorTM*

---

### 2.1 Overview

Graph coloring assigns colors to the vertices of a graph such that any two adjacent vertices have different colors. Graph coloring kernel is widely used in many important real-world applications including the conflicting job scheduling [48, 342–345], register allocation [346–350], sparse linear algebra [351–354], machine learning (e.g., to select non-similar samples that form an effective training set), and chromatic scheduling of graph processing applications [406, 407]. For instance, the chromatic scheduling execution is as follows: given the vertex coloring of a graph, chromatic scheduling performs  $N$  steps that are executed *serially*, where  $N$  is the number of colors used to color the graph, and at each step the vertices assigned to the *same* color are processed *in parallel*, i.e., representing independent tasks that are executed concurrently. In addition, it is of vital importance that programmers manage the registers of modern CPUs effectively, and thus compilers [349, 350] optimize the register allocation problem via graph coloring: compilers construct undirected graphs, named register inference graphs (RIGs), with vertices representing the variables used in the source code and

edges between vertices representing variables that are simultaneously active at some point in the program execution, and then compilers leverage the graph coloring kernel to identify independent variables that can be allocated on the same registers, i.e., if there *no* edge in the RIG connecting the associated vertices of the variables.

To achieve high system performance in the aforementioned real-world scenarios, software designers need to improve three key aspects of the graph coloring kernel. First, they need to minimize the number of colors used to color the input graph. For example, in the chromatic scheduling scheme minimizing the number of colors used to color the graph reduces the number of the sequential steps performed in the multithreaded end-application. However, minimizing the number of colors in graph coloring is an NP-hard problem [408], and thus prior works [57, 58, 343, 344, 351, 409–413] introduce ordering heuristics that generate effective graph colorings with a relatively small number of colors. Second, given that the execution time of the graph coloring kernel adds to the overall parallel overhead of the real-world end-application, software engineers need to design high-performance graph coloring algorithms for modern multicore computing systems. Third, an effective graph coloring for a real-world end-application necessitates a balanced distribution of the vertices across the color classes, i.e., the sizes of the color classes to be almost the same. Producing color classes with high skew in their sizes, i.e., high disparity in the number of vertices distributed across color classes, typically causes load imbalance and low resource utilization in real-world end-application. For example, in the register allocation scenario high disparity in the sizes of the color classes results to a large number of registers needed (high financial costs), equal to the size of the largest color class produced, while a *large* portion of the registers remains idle (unused) for a *long* time during the program execution (i.e., in time periods corresponding to many color classes with small sizes), thus causing low hardware resource utilization. Therefore, software designers need to propose *balanced* and *fast* graph coloring algorithms for commodity computing systems. Our **goal** in this work is to improve the last two key aspects of the graph coloring kernel by introducing high-performance and balanced multithreaded graph coloring algorithms for modern multicore platforms.

With a straightforward parallelization of graph coloring, coloring conflicts may arise when two parallel threads assign the same color to adjacent vertices. To deal with this problematic case, recent works [49, 53–56] perform two additional phases: a conflict detection phase, which iterates over the vertices of the graph to detect coloring inconsistencies between adjacent vertices, and a conflict resolution phase, which iterates over the detected conflicted vertices to resolve the coloring inconsistencies via recoloring. Nevertheless, these prior works [49, 53–56] are still inefficient, as we demonstrate in Section 2.6, because they need to traverse the *whole* graph at least two times (one for coloring the vertices and one for detecting coloring conflicts), and also detect and resolve coloring conflicts with a *lazy* approach, i.e., much later in the runtime compared to the time that the coloring conflicts appeared. As a result, prior approaches access the conflicted vertices of the graph multiple times, however mainly using the expensive last levels of the memory hierarchy (e.g., main memory) of commodity multicore platforms, thus incurring high data access costs.

In this work, we present *ColorTM* [2], a high-performance graph coloring algorithm for multicore platforms. *ColorTM* is designed to provide low synchronization and data access costs. Our algorithm

proposes (a) an *eager* conflict detection and resolution approach, i.e., immediately detecting and resolving coloring inconsistencies when they arise, such that to minimize data access costs by accessing conflicted vertices immediately using the low-cost lower levels of the memory hierarchy of multicore platforms, and (b) a speculative computation and synchronization scheme, i.e., leveraging Hardware Transactional Memory (HTM) and speculatively performing computations and data accesses outside the critical section, such that to provide high levels of parallelism and low synchronization costs by executing multiple small and short critical sections in parallel. Specifically, *ColorTM* consists of three steps: for each vertex on the graph, it (i) speculatively finds a candidate legal color by recording the colors of the adjacent vertices, (ii) validates and updates the color of the current vertex by checking the colors of the critical adjacent vertices within an HTM transaction to detect potential coloring conflicts, and (iii) eagerly repeats steps (i) and (ii) for the current vertex multiple times until a valid coloring is found.

However, *ColorTM* does not provide any guarantee on the sizes of the color classes relative to each other. As we demonstrate in our evaluation (Section 2.6), the color classes produced by *ColorTM* for a real-world graphs have high disparity in the number of vertices across them, thus causing load imbalance and low resource utilization in real-world end-applications. Therefore, we extend our algorithmic design to propose a *balanced* graph coloring algorithm, named *BalColorTM* [2]. *BalColorTM* achieves high system performance and produces highly balanced color classes, i.e., having almost the same number of vertices across color classes, targeting to provide high hardware resource utilization and load balance in the real-world end-applications of graph coloring.

We evaluate *ColorTM* and *BalColorTM* on a dual socket Intel Haswell server using a wide variety of large real-world graphs with diverse characteristics. *ColorTM* improves performance by  $12.98\times$  on average using 56 parallel threads compared to state-of-the-art graph coloring algorithms, while providing similar coloring quality. *BalColorTM* outperforms prior state-of-the-art balanced graph coloring algorithms by  $1.78\times$  on average using 56 parallel threads, and provides the best color balancing quality over prior schemes (See Section 2.6). Finally, we study the effectiveness of our proposed *ColorTM* and *BalColorTM* in parallelizing a widely used real-world end-application, i.e., Community Detection [355], and demonstrate that our proposed algorithmic designs can provide significant performance improvements in real-world scenarios. *ColorTM* and *BalColorTM* are publicly available [2] at [github.com/cgiannoula/ColorTM](https://github.com/cgiannoula/ColorTM).

The main **contributions** of this work are:

- We design high-performance and balanced graph coloring algorithms, named *ColorTM* and *BalColorTM*, for modern multicore platforms.
- We leverage HTM to efficiently detect coloring inconsistencies between adjacent vertices (processed by different parallel threads) with low synchronization costs. We propose an eager conflict resolution approach to efficiently resolve coloring inconsistencies in multithreaded executions by minimizing data access costs.
- We evaluate *ColorTM* and *BalColorTM* using a wide variety of large real-world graphs and demonstrate that they provide significant performance improvements over prior state-of-the-

art graph coloring algorithms. Our proposed algorithmic designs significantly improve performance in multithreaded executions of real-world end-applications.

## 2.2 Prior Graph Coloring Algorithms

In this section, we describe prior state-of-the-art graph coloring algorithms [49, 53–56]. Section 2.2.1 presents the sequential graph coloring algorithm. Section 2.2.2 describes prior parallel (no guarantee on the sizes of color classes) graph coloring algorithms proposed in the literature, while Section 2.2.3 presents prior balanced (color classes are highly balanced) graph coloring algorithms proposed in the literature.

### 2.2.1 The Greedy Algorithm

Figure 2.1 presents the sequential graph coloring algorithm, called *Greedy* [48]. Consider an undirected graph  $G = (V, E)$ , and the neighborhood  $N(v)$  of a vertex  $v \in V$  defined as  $N(v) = \{u \in V : (v, u) \in E\}$ . For each vertex  $v$  of the graph, Greedy records the colors of  $v$ 's adjacent vertices in a forbidden set of colors, and assigns the minimum legal color to the vertex  $v$  based on the forbidden set of colors.

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 for each  $v \in V$  do
4   forbidColors =  $\emptyset$ 
5   for each  $u \in N(v)$  do
6     forbidColors = forbidColors  $\cup \{u.color\}$ 
7    $v.color = \text{minLegalColor}(\text{forbidColors})$ 

```

Figure 2.1: The Greedy algorithm.

The Greedy approach produces at most  $\Delta + 1$  colors [48], where  $\Delta$  is the degree of the graph  $G$ . The degree of the graph is defined as  $\Delta = \max_{v \in V} \{deg(v)\}$ , where  $deg(v)$  is the degree of a vertex  $v$ , which is the number of its adjacent vertices  $deg(v) = |N(v)|$ . However, finding the minimum number of colors to color a graph  $G$  is an NP-hard problem [414]. In this work, we have experimented with the *first-fit* ordering heuristic [48], in which the vertices of the graph are processed and colored in the order they appear in the input graph representation  $G$ , since this heuristic can provide high coloring quality based on prior works [48, 57, 415]. We leave the exploration of other ordering heuristics for future work.

### 2.2.2 Prior Parallel Graph Coloring Algorithms

To parallelize the graph coloring problem, the vertices of the graph are distributed among parallel threads. However, due to crossing edges, the coloring subproblems assigned to parallel threads are not independent, and the parallel algorithm may terminate with an invalid coloring. Specifically, a race condition arises when two parallel threads assign the same color to adjacent vertices. The algorithm

implies that when a parallel thread updates the color of a vertex, the forbidden set of colors of the adjacent vertices has not been changed. Thus, the nature of this algorithm imposes that the reads to the colors of the adjacent vertices of a vertex  $v$  have to be executed *atomically* with the write-update to the color of the vertex  $v$ .

### The SeqSolve Algorithm

Figure 2.2 presents the parallel graph coloring algorithm proposed by Gebremedhin et al. [53], henceforth referred to as SeqSolve. This algorithm consists of three steps: (i) multiple parallel threads iterate over the whole graph and speculatively color the vertices of the graph with *no* synchronization (lines 3-6), (ii) multiple parallel threads iterate over the whole graph and detect coloring inconsistencies that appeared in the (i) step (lines 7-13), and (iii) only *one* single thread resolves the detected coloring inconsistencies by re-coloring the conflicted vertices (lines 14-16). Since the (iii) step is executed by only a single thread, no coloring inconsistencies appear after this step. Note that when a coloring conflict arises between two adjacent vertices, only *one* of the involved adjacent vertices needs to be re-colored, e.g., using a simple order heuristic among the vertices (line 11).

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 // Speculative Coloring - Step (i)
4 for each  $v \in V$  do in parallel
5   Assign the minimum legal color to the vertex  $v$ 
6 --barrier--
7 // Detection of Coloring Inconsistencies - Step (ii)
8  $R = \emptyset$  // Set of Conflicted Vertices
9 for each  $v \in V$  do in parallel
10   for each  $u \in N(v)$  do
11     if  $((v.\text{color} == u.\text{color}) \ \&\& \ (v < u))$ 
12        $R = R \cup v$ 
13 --barrier--
14 // Sequential Resolution of Coloring Conflicts - Step (iii)
15 for each  $v \in R$  do
16   Assign the minimum legal color to the vertex  $v$ 

```

Figure 2.2: The SeqSolve algorithm.

In the SeqSolve algorithm, we make three key observations. First, if the number of coloring conflicts arised in a multithreaded execution is low, the algorithm might scale well [53]. However, as the number of parallel threads increases and the graph becomes denser, i.e., the vertices of the graph have a large number of adjacent vertices, many more coloring conflicts arise in multithreaded executions. In such scenarios, a large number of coloring inconsistencies is resolved sequentially, i.e., by only *one* single thread, thus achieving limited parallelism. Second, we note SeqSolve traverses the whole graph at least two times, i.e., step (i) and step (ii). Assuming large real-world graphs that do not typically fit in the Last Level Cache (LLC) of contemporary multicore platforms, the whole graph is traversed twice using the main memory, thus incurring high data access costs. Third, we observe that SeqSolve detects and resolves the coloring conflicts *lazily*, i.e., much later in the runtime compared to the time that the coloring conflicts appears. Specifically, a coloring inconsistency in a vertex  $v$

might appear in step (i). However, SeqSolve detects the coloring inconsistency in vertex  $v$  in step (ii), i.e., after *first* coloring *all* the remaining vertices of the graph. Similarly, SeqSolve resolves the coloring inconsistency of the vertex  $v$  in step (iii), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining vertices of the graph (step (ii)). As a result, many conflicted vertices are accessed multiple times in the runtime, however with a lazy approach, i.e., accessing them through the expensive last levels of the memory hierarchy of commodity platforms, thus incurring high data access costs.

### The IterSolve Algorithm

Figure 2.3 presents the parallel graph coloring algorithm proposed by Boman et al. [54,55], henceforth referred to as IterSolve. This algorithm consists of two repeated steps: (i) multiple parallel threads iterate over the uncolored vertices of the graph and speculatively color the uncolored vertices of the graph with *no* synchronization (lines 5-8), (ii) multiple parallel threads iterate over the recently-colored vertices of the graph and detect coloring inconsistencies appeared in the (i) step (lines 9-15). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph.

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3  $U = V$ 
4 while  $U \neq \emptyset$ 
5   // Speculative Coloring - Step (i)
6   for each  $v \in U$  do in parallel
7     Assign the minimum legal color to the vertex  $v$ 
8   --barrier--
9   // Detection of Coloring Inconsistencies - Step (ii)
10   $R = \emptyset$  // Set of Conflicted Vertices
11  for each  $v \in U$  do in parallel
12    for each  $u \in N(v)$  do
13      if  $((v.\text{color} == u.\text{color}) \ \&\& \ (v < u))$ 
14         $R = R \cup v$ 
15  --barrier--
16   $U = R$ 

```

Figure 2.3: The IterSolve algorithm.

In the IterSolve algorithm, we make four key observations. First, the programmer needs to explicitly define forward progress in the source code, so that the IterSolve algorithm terminates. Specifically, to ensure forward progress when a coloring inconsistency appears between two adjacent vertices, the programmer needs to *explicitly* define only *one* of them to be re-colored (line 13), e.g., based on the vertices' ids. Otherwise, the two adjacent vertices may *always* obtain the same color, if they are always being processed by different threads. Second, similarly to SeqSolve, IterSolve traverses the whole graph at least two times (steps (i) and (ii)), i.e., in the first iteration of the while loop in line 4, where the set  $U$  is equal to the set  $V$  (line 3). In the first iteration of the while loop, the whole large real-world graph is accessed through the main memory twice, thus incurring high data access costs. Third, similarly to SeqSolve, IterSolve detects and resolves the coloring conflicts *lazily*. Specifically,



a coloring inconsistency in a vertex  $v$  might appear in step (i) (line 7), it is detected in step (ii) (line 13), i.e., after *first* coloring *all* the remaining uncolored vertices of the graph. Moreover, IterSolve resolves the coloring inconsistency of a vertex  $v$  in step (i) (with re-coloring), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining recently-colored vertices of the graph (step (ii)). Thus, IterSolve incurs high data access costs on the many conflicted vertices, which are accessed multiple times in the runtime with *lazy* approach, through the last levels of the memory hierarchy of commodity platforms. Fourth, the iterative process of resolving coloring conflicts may introduce new conflicts, and thus, IterSolve might need additional iterations to fix them. This scenario may happen when adjacent vertices are assigned to the *same* thread and incur coloring inconsistencies, they will be assigned and processed by *different* parallel threads in the next iteration. The authors of the original IterSolve papers [54, 55] empirically observe that a few iterations of IterSolve are needed to produce a valid coloring. However, the authors used *synthetic* and *not* real-world graphs in their evaluation. In addition, the more iterations are needed, the more *lazy* traversals on the conflicted vertices of the graph are performed, which can significantly degrade performance.

### The IterSolver Algorithm

Figure 2.4 presents the parallel graph coloring algorithm proposed by Rokos et al. [56], henceforth referred to as IterSolverR. Rokos et al. observed that the IterSolve algorithm (Figure 2.3) can be improved by merging the steps (i) and (ii) into a single *detect-and-re-color* step, thus eliminating one of the two barrier synchronizations of IterSolve (lines 8 and 15 in Figure 2.3). When a coloring inconsistency on a vertex  $v$  is found, the vertex  $v$  can be immediately re-colored (line 18 in Figure 2.4). However, the new re-coloring on the vertex  $v$  may *again* introduce a coloring inconsistency in multithreaded executions, since re-colorings are performed *concurrently* by multiple parallel threads (line 11). Therefore, the vertex  $v$  is marked as *recently-re-colored* vertex (line 19), and needs to be re-validated in the next iteration of IterSolverR. Overall, IterSolverR (Figure 2.4) first speculatively colors all the vertices of the graph and marks them as *recently-colored* vertices (lines 3- 6). Then, it executes one single repeated step (lines 8-21): multiple parallel threads iterate over the recently-colored vertices of the graph, and detect if coloring inconsistencies have appeared, which in that case are immediately resolved via re-coloring. This step is repeated until there are no *recently-re-colored* vertices: in one single iteration of this step, there are *no* coloring inconsistencies detected in any adjacent vertices of the graph.

In the IterSolverR algorithm, even though one barrier synchronization is eliminated compared to IterSolve, we observe that IterSolverR still traverses the whole graph at least twice: (i) in Step 0 (lines 4-5), and (ii) in the first iteration of the while loop in line 8, where the set  $U$  is equal to the set  $V$  (line 7), including all the vertices of the graph. Thus, IterSolverR traverses the large real-world graph twice through the main memory, incurring high data access costs. In addition, we find that similarly to SeqSolve and IterSolve, the IterSolverR algorithm also detects the coloring inconsistencies *lazily*. Specifically, a coloring inconsistency on a vertex  $v$  might appear in the re-coloring process of lines 17-19, since the re-coloring process is *concurrently* executed on multiple conflicted vertices by multiple parallel threads. However, re-coloring inconsistencies of lines 17-19 are detected in the next

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 // Speculative Coloring (Step 0)
4 for each  $v \in V$  do in parallel
5     Assign the minimum legal color to the vertex  $v$ 
6 --barrier--
7  $U = V$  // Mark all Vertices as Recently-Colored
8 while  $U \neq \emptyset$ 
9      $R = \emptyset$  // Set of Recently-Colored Vertices
10    // Conflict Detection and Resolution (Step (i))
11    for each  $v \in U$  do in parallel
12        bool conflict-detected = false
13        for each  $u \in N(v)$  do
14            if  $((v.\text{color} == u.\text{color}) \ \&\& \ (v < u))$ 
15                conflict-detected = true
16            break
17        if (conflict-detected == true)
18            Assign the minimum legal color to vertex  $v$ 
19             $R = R \cup v$  // Mark vertex  $v$  as Recently-Colored
20 --barrier--
21  $U = R$ 

```

Figure 2.4: The IterSolveR algorithm.

iteration of the step (i) in lines 13-16, i.e., after *first* processing *all* the remaining vertices of the set  $U$  (line 11). Therefore, as we demonstrate in our evaluation (Section 2.6.1), IterSolveR is still inefficient, incurring high data access costs on multiple conflicted vertices which are accessed multiple times in the runtime with a *lazy* approach.

### 2.2.3 Prior Balanced Graph Coloring Algorithms

To provide a balanced coloring on a graph in which the color classes produced include almost the same number of vertices, a two-step process is typically used: (i) obtain an initial graph coloring using a balanced-oblivious algorithm (e.g., Section 2.2.2), and (ii) obtain a balanced graph coloring using a balanced-aware (henceforth referred to as balanced for simplicity) graph coloring algorithm, as we describe next. Specifically, given a graph  $G = (V, E)$ , we can assume that the number of colors produced by the initial coloring step (i) is  $C$ . A strictly balanced graph coloring results in the size of *each* color class being  $b = V/C$ .<sup>1</sup> Therefore, we refer to the color classes whose sizes are greater than  $b$  as *over-full* classes, and those whose sizes are less than  $b$  as *under-full* classes. Balanced graph coloring algorithms leverage the quantity of  $b$ , which can be extracted by first executing an initial balanced-oblivious graph coloring on the graph, in order to provide balanced color classes on a graph.

<sup>1</sup>Please note that in our work we make the following assumption: in a real-world end-application, the vertices of the graph represent sub-tasks that have almost equal load/weights of computation. If the vertices of the input graph have different load/weights of computation, a pre-processing step needs to be applied in the original graph: vertices with large computation weights/load are split into multiple smaller vertices, each of them has one weight/load unit of computation.

## The Color-Centric (CLU) Algorithm

Figure 2.5 presents the color-centric balanced graph coloring algorithm proposed by Lu et al. [49], henceforth referred to as CLU. In this scheme, vertices belonging in the *same* color class are processed concurrently, and a *subset* of vertices from each over-full color class is moved to under-full color classes in order to achieve high color balance. Only vertices belonging in over-full color classes are considered for re-coloring, while graph coloring balance is achieved *without* increasing the number of color classes produced by the initial graph coloring.

```

1 Input: Graph  $G=(V,E)$ 
2 Obtain an initial coloring on  $G$ 
3 Let  $C$  be the number of colors produced
4 Let  $b=V/C$  be the perfect balance
5 Let  $Q$  be the set of vertices of the over-full color classes
6 for each  $c \in Q$  do // Process the Over-Full Color Classes
7   Let  $R(c)$  be the set of vertices with color  $c$ 
8   for each  $v \in R(c)$  do in parallel
9     if (the size of the color class  $c \leq b$ )
10      continue // Color Class is Balanced
11   Let  $k$  be the index of the minimum under-full color class that is
      permissible to vertex  $v$ 
12   if ( $k$  exists) // Re-Coloring
13      $v.color = k$ 
14     Atomically decrease the size of the color class  $c$ 
15     Atomically increase the size of the color class  $k$ 
16 --barrier--

```

Figure 2.5: The CLU algorithm.

The CLU algorithm (Figure 2.5) processes the over-full color classes sequentially (lines 6 and 16), while vertices belonging at the same over-full color class are processed concurrently (line 8). CLU iterates over each vertex  $v$  of an over-full color class, and finds the minimum color of an under-full color class that is permissible to be assigned at the vertex  $v$  (line 11). If such a color exists, the vertex  $v$  is re-colored with a color of an under-full color class (lines 12-15). The CLU algorithm iterates over the vertices of each over-full color class until that particular over-full class becomes balanced at a certain point in the execution, i.e., until when its size becomes smaller or equal to  $b$  (lines 9-10). Then, the vertices belonging on that color class are no longer considered for re-coloring (line 10). Thus, this algorithm terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging in an over-full color class.

In the CLU algorithm, we make two key observations. First, parallel threads always process vertices of the *same* color, thus no coloring inconsistencies are produced: since vertices had the same color in the initial coloring, they are *not* adjacent vertices, and thus they can be re-colored with the same color of an under-full color class without violating correctness. This way CLU requires *only* one iteration over the vertices of all the over-full color classes. Second, the parallel performance of CLU depends on the number of the over-full color classes produced in the initial coloring. CLU requires  $F$  steps, where  $F$  is the number of over-full color classes produced in the initial coloring. At *each* of these steps, i.e., for each over-full color class on the initial coloring, CLU introduces a barrier syn-

chronization among parallel threads (line 16). This way it increases the synchronization costs, which might significantly degrade scalability in multithreaded executions.

### The Vertex-Centric (VFF) Algorithm

Figure 2.6 presents the vertex-centric balanced graph coloring algorithm proposed by Lu et al. [49], henceforth referred to as VFF. The VFF algorithm is the balanced graph coloring counterpart of the IterSolve algorithm (Figure 2.3). In this scheme, vertices from *different* color classes are processed concurrently by parallel threads. Thus, in contrast to CLU, VFF introduces coloring inconsistencies. However, similarly to CLU, in VFF only vertices belonging in over-full color classes are considered for re-coloring, i.e., to be moved to under-full color classes, while graph coloring balance is also achieved *without* increasing the number of color classes produced by the initial graph coloring.

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 Obtain an initial coloring on  $G$ 
4 Let  $C$  be the number of colors produced
5 Let  $b = V/C$  be the perfect balance
6 Let  $Q$  be the set of vertices of the over-full color classes
7 while  $Q \neq \emptyset$  // Process the Over-Full Color Classes
8   // Speculative Re-Coloring - Step (i)
9   for each  $v \in Q$  do in parallel
10     Let  $c$  be the current color of the vertex  $v$ 
11     if  $((c \neq -1) \ \&\& \ (\text{the size of the color class } c \leq b))$ 
12       continue // Color Class is Balanced
13     Let  $k$  be the index of the minimum under-full color class that is
        permissible to vertex  $v$ 
14     if  $(k \text{ exists})$  // Re-Coloring
15        $v.\text{color} = k$ 
16       Atomically decrease the size of the color class  $c$ 
17       Atomically increase the size of the color class  $k$ 
18   --barrier--
19   // Detection of Coloring Inconsistencies - Step (ii)
20    $R = \emptyset$  // Conflicted Vertices of Over-Full Color Classes
21   for each  $v \in Q$  do in parallel
22     for each  $u \in N(v)$  do
23       if  $((v.\text{color} == u.\text{color}) \ \&\& \ (v < u))$ 
24          $R = R \cup v$ 
25        $v.\text{color} = -1$ 
26   --barrier--
27    $Q = R$ 

```

Figure 2.6: The VFF algorithm.

Similarly to IterSolve, VFF (Figure 2.6) consists of two repeated steps: (i) multiple parallel threads iterate over vertices of over-full color classes and speculatively re-color them with permissible colors of under-full color classes, if possible (lines 8-18), and (ii) multiple parallel threads iterate over the recently re-colored vertices and detect coloring inconsistencies that appeared in the (i) step (lines 19-26). Similarly to CLU, VFF iterates over the vertices of an over-full color class until that particular over-full class becomes balanced at a certain point in the execution, i.e., until when its size becomes smaller or equal to  $b$  (lines 11-12). Then, the vertices belonging on that particular color class are no

longer considered for re-coloring (line 12). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph, and the algorithm terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging in an over-full color class.

Since VFF is the balanced graph coloring counterpart of IterSolve, we report similar key observations for them. First, VFF detects and resolves the coloring conflicts *lazily*. Specifically, a coloring inconsistency in a vertex  $v$  might appear in step (i), while it is detected in step (ii), i.e., after *first* iterating over *all* the remaining vertices of over-full color classes. Moreover, VFF resolves the coloring inconsistency in a vertex  $v$  in step (i) (re-coloring), i.e., after *first* detecting if coloring inconsistencies exist in *all* the remaining recently re-colored vertices (in step (ii) of the previous iteration). Thus, VFF incurs high data access costs due to accessing multiple conflicted vertices in the runtime through the last levels of the memory hierarchy of commodity platforms. Second, the iterative process of resolving coloring conflicts may introduce new conflicts, and thus, VFF might need additional iterations to fix them. This scenario may happen when adjacent vertices are assigned to the *same* thread and incur coloring inconsistencies, they will be assigned and processed by *different* parallel threads in the next iteration. Note that the more iterations are needed, the more *lazy* traversals on the conflicted vertices of the graph are performed, which might significantly degrade performance.

### The Recoloring Algorithm

Figure 2.7 presents the re-coloring balanced graph coloring algorithm proposed by Lu et al. [49], henceforth referred to as Recoloring. Recoloring is similar to the VFF (Figure 2.6) and IterSolve (Figure 2.3) schemes. The key idea of this algorithm is that after performing an initial graph coloring with  $C$  colors, *all* the vertices of the graph are re-colored, having an additional condition on the color selection in order to achieve better vertex balance across color classes compared to that produced by the initial graph coloring. Specifically, Recoloring leverages the perfect balance  $b = V/C$  known from the initial graph coloring, and keeps track the sizes of the color classes during the execution in order to improve vertex balance across color classes as follows: each vertex is re-colored using the minimum permissible color  $k$  such that the size of the color class  $k$  is less than  $b$ .

Similarly to IterSolve and VFF, Recoloring (Figure 2.7) consists of two repeated steps: (i) multiple parallel threads iterate over all the vertices of the graph and speculatively re-color them with a new permissible color  $k$ , that satisfies the condition that the size of the color class  $k$  is less than  $b$  (lines 12-17), and (ii) multiple parallel threads iterate over the recently re-colored vertices and detect coloring inconsistencies that appeared in the (i) step (lines 18-25). The steps (i) and (ii) are iteratively repeated until there are *no* coloring inconsistencies in any adjacent vertices of the graph. In contrast to VFF and CLU, Recoloring does not guarantee that the graph color balance achieved uses the same number of colors with the initial graph coloring. To avoid producing a large number of color classes, the Recoloring scheme [49] (Figure 2.7) re-colors the vertices of the graph with the following order: assuming that the vertices of the graph are ordered such that the vertices of the same color class are

listed consecutively (line 6), Recoloring iterates over the vertex sets of the color classes in the reverse order compared to that produced in the initial graph coloring, i.e., starting from the vertices assigned to the color class with the largest index (See line 8). The rationale behind this heuristic is that the vertices that are "difficult" to color, i.e., in the initial graph coloring they are assigned to a color class with large index, will be processed *early*, thus aiming to produce a small number of color classes. For more details, we refer the reader to [49].

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 Obtain an initial coloring on  $G$ 
4 Let  $C$  be the number of colors produced
5 Let  $b = V/C$  be the perfect balance
6 Let  $K(j)$  be the set of vertices  $u$  with color  $j$ 
7 //  $K(j) = \{u \in V, u.color = j\}$ 
8 Construct the order set  $W = \{K(C), K(C-1), \dots, K(1), K(0)\}$ 
9 Initialize the sizes of the  $C$  color classes to 0
10  $Q = W$ 
11 while  $Q \neq \emptyset$  // Re-Color the Whole Graph
12   // Speculative Coloring - Step (i)
13   for each  $v \in Q$  do in parallel
14     Let  $k$  be the minimum color that is permissible to the vertex  $v$  such that
       the size of the color class  $k$  is less than  $b$  // Balanced Color
       Classes
15      $v.color = k$ 
16     Atomically increase the size of the color class  $k$ 
17   --barrier--
18   // Detection of Coloring Inconsistencies - Step (ii)
19    $R = \emptyset$  // Set of Conflicted Vertices
20   for each  $v \in Q$  do in parallel
21     for each  $u \in N(v)$  do
22       if  $((v.color == u.color) \ \&\& \ (v < u))$ 
23         Atomically decrease the size of the color class  $v.color$ 
24        $R = R \cup v$ 
25   --barrier--
26    $Q = R$ 

```

Figure 2.7: The Recoloring algorithm.

In Recoloring, we make three key observations. First, Recoloring traverses the whole graph, i.e., it re-colors *all* the vertices of the graph, while CLU and VFF re-color only a *subset* of the vertices of over-full color classes. As a result, Recoloring performs a much larger number of computations and memory accesses compared to VFF and CLU. Second, similarly to IterSolve and VFF, Recoloring detects and resolves coloring inconsistencies with a *lazy* approach, thus incurring high data access costs. Recoloring may also introduce new conflicts, thus resulting in additional iterations to fix them. Third, even though Recoloring employs a different vertex ordering heuristic to re-color vertices compared to that used in the initial graph coloring (vertices are colored with the order they appear in the input graph), there is *no* guarantee on the number of color classes that will be produced. As we demonstrate in our evaluation (Section 2.6.2), Recoloring might significantly increase the number of color classes produced compared to that produced in the initial graph coloring.

## 2.3 *ColorTM*: Overview

Our proposed algorithmic design is a high-performance graph coloring algorithm for multicore platforms. *ColorTM* provides low synchronization and data access costs by relying on two key techniques, that we describe in detail next.

### 2.3.1 Speculative Computation and Synchronization

As already discussed, the graph coloring kernel implies that the reads to the colors of the adjacent vertices of a vertex  $v$  have to be executed atomically with the write-update to the color of the vertex  $v$ . Figure 2.8 presents a straightforward parallelization scheme of the graph coloring problem. A naive parallelization approach would be to distributed the vertices of the graph across parallel threads, and for each vertex to include within a critical section the whole block of code that computes and assigns a permissible color to that vertex. However, this approach results in large critical sections with large data access footprints and long duration, and significantly limits the amount of parallelism and the scalability to a large number of threads.

```

1 Input: Graph  $G=(V,E)$ 
2 for each  $v \in V$  do in parallel
3   // Atomic Coloring Step (i)
4   begin_critical_section
5   Compute and assign the minimum legal color to the vertex  $v$ 
6   end_critical_section
```

Figure 2.8: A Naive Approach.

We observe that it is not necessary to include inside the critical section (i) the computations performed to find a permissible color for a vertex  $v$ , and (ii) the accesses to *all* the adjacent vertices of the vertex  $v$ . Figure 2.9 presents an overview of *ColorTM*. For each vertex  $v$ , we design *ColorTM* to implement a speculative computation scheme through two sub-steps: (i) speculatively compute a permissible color  $k$  for the vertex  $v$  (line 5) without using synchronization and track the set of critical adjacent vertices (line 6), i.e., a subset of  $v$ 's adjacent vertices that can cause coloring inconsistencies with the vertex  $v$  (See Section 2.4.2 for more details), and (ii) execute a critical section (using synchronization) that validates the speculative color  $k$  computed in step (i) over the colors of the critical adjacent vertices (lines 8-9) and assigns the color  $k$  to the vertex  $v$ , if the validation succeeds (lines 10-14). With the proposed speculative computation scheme, we provide small critical sections, i.e., having small data access footprints and short duration, thus achieving high amount of parallelism and high scalability to a large number of threads.

In addition, we leverage Hardware Transactional Memory (HTM) to implement synchronization on critical sections (lines 7, 12, and 14 of Figure 2.9). HTM enables a speculative synchronization mechanism: multiple critical sections of parallel threads are executed concurrently with an optimistic approach that they will not cause any data inconsistency, even though their data access sets might overlap. In contrast, fine-grained locking with software-based locks (e.g., provided by the pthread



```

1 Input: Graph  $G=(V,E)$ 
2 for each  $v \in V$  do in parallel
3   RETRY:
4   // Speculative Computation
5   Compute a speculative minimum color  $k$  that is permissible to the vertex  $v$ 
6   Keep track the critical adjacent vertices of the vertex  $v$ 
7   begin_critical_section
8   // Validate Coloring
9   Compare  $k$  with the colors of the critical adjacent vertices
10  if (no coloring conflict)
11     $v.\text{color} = k$ 
12    end_critical_section
13  else
14    end_critical_section
15    goto RETRY // Eager Resolution

```

Figure 2.9: *ColorTM*: Overview.

library) constitutes a more conservative synchronization approach: multiple critical sections of parallel threads are executed concurrently, *only* if their data access sets do *not* overlap. Therefore, HTM can enable a higher number of critical sections to be executed in parallel compared to that enabled with the fine-grained locking scheme. We provide more details in Section 2.4.1. With the speculative synchronization approach of HTM, *ColorTM* further minimizes synchronization costs and provides high amount of parallelism.

### 2.3.2 Eager Coloring Conflict Detection and Resolution

We design *ColorTM* to detect and resolve coloring inconsistencies *eagerly*, i.e., immediately detecting and resolving coloring inconsistencies at the time that the coloring conflicts appear. This way, the conflicted vertices are accessed multiple times, however within a short time during runtime. Therefore, application data corresponding to conflicted vertices can remain and be located in the first levels of the memory hierarchy of commodity platforms (i.e., in the low-cost cache memories), thus enabling *ColorTM* to improve performance by achieving low data access costs.

In Figure 2.9, parallel threads concurrently compute speculative colors for multiple vertices of the graph (lines 4-6), and at that time coloring inconsistencies may appear. Then, parallel threads *immediately* detect possible coloring conflict inconsistencies for the current vertices using synchronization (lines 7-14). This way, parallel threads detect conflicts by accessing application data with low access latencies, since the data accessed in lines 7-14 has just been accessed within a short time, i.e., in lines 4-6. Next, if coloring conflicts arise (line 13), parallel threads *immediately* resolve the coloring conflicts by directly retrying to find new colors for the current vertices (**goto** **RETRY** inline 15) (without proceeding to process new vertices). This way, parallel threads resolve conflict inconsistencies by accessing application data with low access latencies, since the data accessed in lines 4-6 after the execution of **goto** **RETRY** has just been accessed within a short time, i.e., in lines 7-14 of the previous iteration.

In *ColorTM*, we highlight two important key design choices. First, *ColorTM* executes only *one* single parallel step (line 2). In contrast to prior state-of-the-art parallel graph coloring algorithms [49,



53–56], *ColorTM* *completely* avoids barrier synchronization among parallel threads: multiple parallel threads repeatedly iterate over each vertex of the graph until a valid coloring is found. By completely avoiding barrier synchronization, *ColorTM* can provide high scalability. Second, *ColorTM* does not perform re-colorings to vertices: once a vertex is assigned a permissible color, it will *not* be re-colored again during the runtime. This way, colored vertices will *not* introduce coloring inconsistencies with vertices that will be processed next. Prior *lazy* iterative graph coloring schemes including IterSolve, IterSolveR, VFF and Recoloring do *not* use data synchronization when they assign permissible colors to vertices. This way, many vertices are re-colored multiple times with different colors during runtime, and thus new additional coloring inconsistencies might be introduced due to re-colorings. Instead, *ColorTM* employs HTM synchronization (lines 7, 12 and 14 of Figure 2.9) when it assigns permissible colors to vertices (line 11 of Figure 2.9). This way, vertices are assigned only *one* final color during the runtime, thus avoiding introducing new coloring inconsistencies due to re-colorings.

## 2.4 *ColorTM*: Detailed Design

*ColorTM* [1] is a high-performance graph coloring algorithm that leverages HTM to implement synchronization among parallel threads, and performs speculative computations outside the critical section in order to minimize the memory footprint and computations executed inside the critical section. In the section, we describe the detailed design and correctness of *ColorTM*. We also extend our proposed design to introduce a new balanced graph coloring algorithm, named *BalColorTM*, which evenly distributes the vertices of the graph across color classes.

### 2.4.1 Speculative Synchronization via HTM

*ColorTM* leverages HTM to implement synchronization among parallel threads instead of using fine-grained locking. As already discussed, HTM is a more optimistic synchronization approach and can provide higher levels of parallelism compared to the fine-grained locking scheme. Specifically, multiple critical sections with *overlapped* data access sets can be executed in parallel with HTM, while they need to be executed sequentially with fine-grained locking.

Figure 2.10 provides an example of the aforementioned scenario in graph coloring. Consider the scenario where thread  $T1$  attempts to assign a color to the vertex  $v$ , and thread  $T2$  attempts to assign a color to the vertex  $x$ . Thread  $T1$  needs to *atomically* read the colors of the adjacent vertices of the vertex  $v$ , i.e.,  $u, r, z$  vertices, and write the corresponding color to the vertex  $v$ . Similarly, Thread  $T2$  needs to *atomically* read the colors of the adjacent vertices of the vertex  $x$ , i.e.,  $u$  vertex, and write the corresponding color to the vertex  $x$ . With HTM (Figure 2.10a),  $T1$ 's and  $T2$ 's transactions can be executed and committed concurrently: neither the write-set of  $T1$ 's transaction does *not* conflict with the read-set of  $T2$ 's transaction, nor the write-set of  $T2$ 's transaction does *not* conflict with the read-set of  $T1$ 's transaction. Therefore, even though  $T1$ 's and  $T2$ 's critical sections have *overlapped* data access sets, i.e., both of them include the color of the vertex  $u$  in their read-sets, they can be executed concurrently with HTM. In contrast, with fine-grained locking,  $T1$ 's and  $T2$ 's critical sections are

executed *sequentially* (Figure 2.10b): threads  $T1$  and  $T2$  compete to acquire the *same* lock, i.e., the lock associated with the vertex  $u$ , in order to execute their critical sections. Thus, only *one* of threads  $T1$  and  $T2$  will acquire the lock, and will proceed. Given that  $T1$ 's and  $T2$ 's critical sections have *overlapped* data access sets, i.e., both of them include the color of the vertex  $u$  in their read-sets, they will be executed sequentially when using the fine-grained locking scheme for synchronization. As a result, we conclude that in graph coloring HTM can provide higher levels of parallelism compared to fine-grained locking.

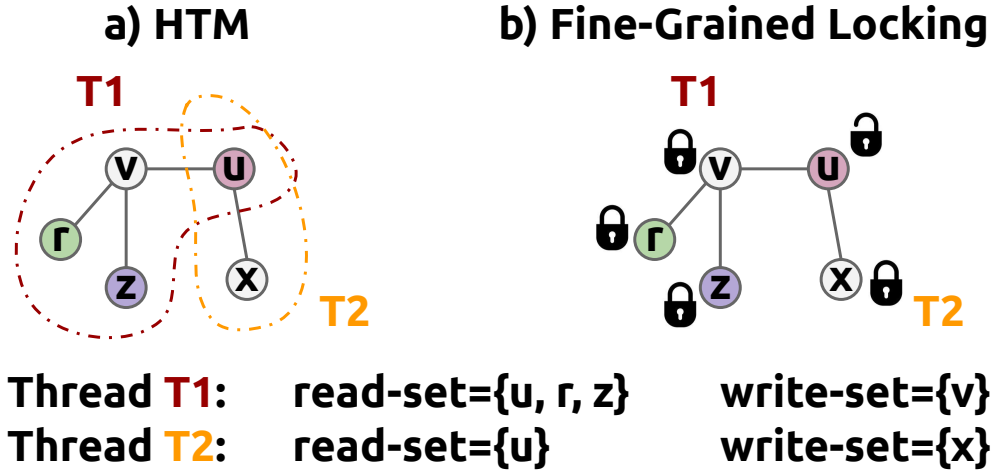


Figure 2.10: An example execution scenario in which threads  $T1$  and  $T2$  attempt to concurrently find colors for the vertices  $v$  and  $x$ , respectively, using a) HTM and b) fine-grained locking for synchronization. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color.

To this end, *ColorTM* employs HTM to deal with race conditions that arise when parallel threads concurrently process adjacent vertices. HTM can detect and resolve coloring inconsistencies among parallel threads as follows:

- **HTM can detect coloring conflicts:** HTM detects coloring conflicts that arise due to crossing edges. For a vertex  $v$  to be colored, we enclose within the transaction (i) the memory location that stores the color of the current vertex  $v$  (the transaction's write-set), and (ii) the memory locations that store the colors of the critical adjacent vertices of the vertex  $v$  (the transaction's read-set). When parallel threads attempt to concurrently update-write the colors of adjacent vertices using different transactions, the HTM mechanism detects read-write conflicts across the running transactions: a running transaction attempts to write the read-set of another running transaction. Figure 2.11 provides an example scenario on how HTM detects coloring inconsistencies among two parallel threads. When the thread  $T1$  attempts to color the vertex  $v$  using HTM, the corresponding running transaction includes the memory location of the color of the vertex  $v$  in its write-set, and the memory locations of the colors of the  $v$ 's adjacent vertices, i.e.,  $u$ ,  $r$  and  $z$  vertices, in its read-set. Similarly, when the thread  $T2$  attempts to color the vertex  $u$  using HTM, the corresponding running transaction includes the memory location of the color of the vertex  $u$  in its write-set, and the memory locations of the colors of the  $u$ 's adjacent vertices, i.e.,  $v$  and  $x$  vertices, in its read-set. When  $T1$ 's and  $T2$ 's transactions are executed concurrently, HTM detects a read-write

conflict either on the color of the vertex  $v$  or the color of the vertex  $u$ : either  $T1$ 's transaction attempts to write the read-set of  $T2$ 's transaction or  $T2$ 's transaction attempts to write the read-set of  $T1$ 's transaction. Therefore, one of the two running transactions will be aborted by the HTM mechanism, and the other one will be committed.

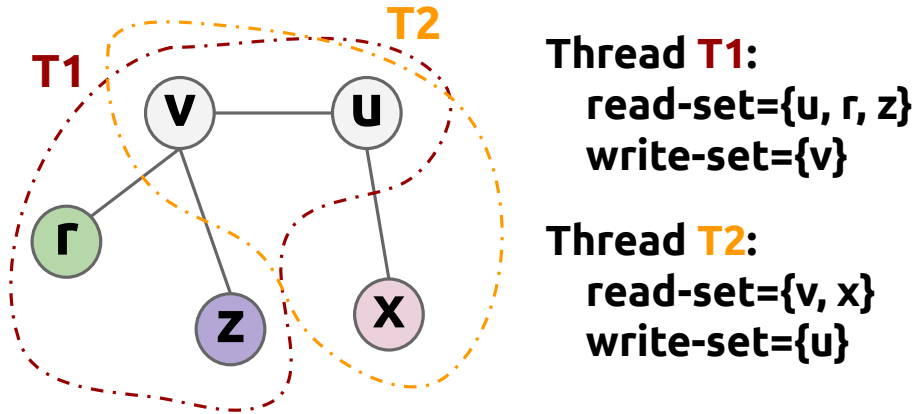


Figure 2.11: An example execution scenario in which threads  $T1$  and  $T2$  attempt to concurrently update the colors of the vertices  $v$  and  $u$  respectively, using two different transactions, and the HTM mechanism detects read-write conflicts to their data sets. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color.

– **HTM can resolve coloring conflicts:** In case of  $n$  conflicting running transactions (read-write conflicts explained in Figure 2.11), the HTM mechanism aborts  $n - 1$  running transactions and commits only one of them. In prior graph coloring schemes such as SeqSolve (line 11 of Figure 2.2), IterSolve (line 13 of Figure 2.3), VFF (line 23 of Figure 2.6) and Recoloring (line 22 of Figure 2.7), the programmer explicitly defines a coloring conflict resolution policy among conflicted vertices to guarantee forward progress, i.e., the programmer explicitly defines which of the conflicted vertices will be re-colored next. In contrast, in *ColorTM* when coloring conflicts arise among multiple running transactions, the programmer does *not* need to explicitly define a conflict resolution policy: the HTM mechanism itself commits one of the multiple conflicted transactions and aborts the remaining running transactions. Thus, the conflict resolution policy implemented in the underlying hardware mechanism of HTM determines which vertices will continue to be processed for coloring.

However, currently available HTM systems [336, 337, 416, 417] are best-effort HTM implementations that do *not* guarantee forward progress: a transaction may always fail to commit and thus, a non-transactional execution path (*fallback path*) needs to be implemented. The most common fallback path is to implement a coarse-grained locking solution: each transaction can be retried up to a predefined number of times (pre-determined threshold), and if it exceeds this threshold, it falls back to the acquisition of global lock, which allows only one single thread to execute its critical section. To implement this, the global lock is added to the transactions' read sets: inside the transaction the thread always reads the value of the global lock variable. During the multithreaded execution, when the transaction of a parallel thread exceeds the predefined threshold of retries, the parallel thread acquires the global lock by writing to the value of the global lock variable, and then the concurrent

running transactions of the remaining threads are aborted (read-write conflict) and wait until the global lock is released.

### 2.4.2 Critical Adjacent Vertices

*ColorTM* implements a speculative computation approach to achieve high performance. Specifically, for each vertex  $v$ , all necessary computations to find a permissible color  $k$  are performed outside the critical section (line 5 in Figure 2.9) such that avoid unnecessary computations inside the critical sections. Within the critical section, *ColorTM* only validates the speculative color  $k$  (line 9 in Figure 2.9) by comparing it with the colors of the adjacent vertices of vertex  $v$ . However, the speculative color  $k$  for a vertex  $v$  does *not* need to be validated with the colors of *all* the adjacent vertices of vertex  $v$ : we observe that some adjacent vertices can be omitted from the validation process of the critical section, because they do not cause *any* coloring inconsistency with the vertex  $v$ . Specifically, we can omit from the validation step performed within the critical section the following adjacent vertices of vertex  $v$ :

1. **The adjacent vertices that are assigned to be processed by the same thread with the vertex  $v$ .** Given that the vertices of the graph are distributed across multiple threads, coloring conflicts cannot arise between adjacent vertices that are assigned to the *same* parallel thread. Therefore, we omit from the validation step of the critical section the adjacent vertices assigned to the same thread as the current vertex  $v$ .
2. **The adjacent vertices that have already obtained a color.** As already explained, *ColorTM* does *not* perform re-colorings to the vertices of the graph: once a vertex is assigned a permissible color within the critical section (using synchronization), it will *not* be re-colored again during runtime. Multiple parallel threads repeatedly iterate over a vertex until a valid coloring is found, which is assigned to it using data synchronization, and then proceed to the remaining vertices. Therefore, in *ColorTM* coloring conflicts do *not* arise between adjacent vertices that have already obtained a color: the colors assigned to adjacent vertices are taken into consideration in the computations performed outside the critical section (line 5 in Figure 2.9) to find a speculative color for the current vertex, and will not be modified when the critical section is executed (lines 7-15 in Figure 2.9), since *ColorTM* does *not* perform re-colorings. Therefore, adjacent vertices of a vertex  $v$  that have already obtained a color when the speculative coloring computation step (line 5 in Figure 2.9) is executed, do not cause any coloring inconsistency when critical section is executed (lines 7-15 in Figure 2.9). Hence, we can safely omit from the validation step of the critical section the adjacent vertices that have already been assigned a color.

Figure 2.12 presents an example execution scenario of a graph partitioned across two parallel threads  $T1$  and  $T2$ . In Figure 2.12, the white vertices represent uncolored vertices and the colorful vertices represent vertices that have already obtained a color during runtime. In this scenario, threads  $T1$  and  $T2$  attempt to color the vertices  $v$  and  $u$ , respectively. According to our described optimizations, the adjacent vertices that need to be validated inside the critical sections (via HTM) of the vertices  $v$  and  $u$  are *only* the vertices  $u$  and  $v$ , respectively.

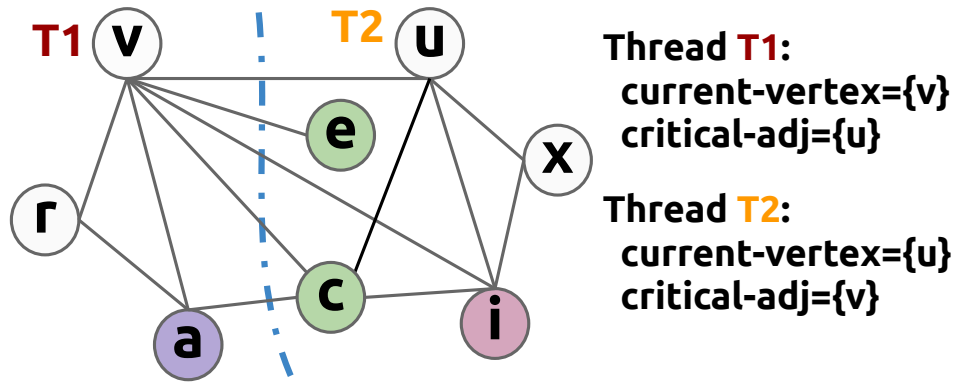


Figure 2.12: An example execution scenario in which the graph is partitioned across two parallel threads. The white circles represent uncolored vertices, and the colorful circles represent vertices that have already obtained a color. When the threads  $T1$  and  $T2$  attempt to color the vertices  $v$  and  $u$ , respectively, the critical adjacent vertices that need to be validated within the critical section (HTM) are *only* the vertices  $u$  and  $v$ , respectively.

Overall, for the current vertex  $v$  to be colored, the necessary adjacent vertices that need to be validated inside the critical section, referred to as *critical* adjacent vertices, are the *uncolored* adjacent vertices assigned to different parallel threads compared to the thread to which the vertex  $v$  is assigned to. By accessing inside the critical section *only* a few data needed to ensure correctness, *ColorTM* provides short critical sections and small transaction footprints, and achieves high levels of parallelism and low synchronization costs, i.e., low abort ratio in hardware transactions of HTM (See Figure 2.18). Note that having large transactions footprints in HTM transactions can cause three important problems: (i) if the transaction read- and write-sets are large, the available hardware buffers of HTM may be oversubscribed (hardware overflow), and in that case the HTM mechanism will abort the running transactions due to capacity aborts, (ii) if the duration of a running transaction is long (e.g., due to expensive data accesses), the running transactions may be aborted due to a time interrupt (when the duration of a transaction exceeds the time scheduling quantum, the OS scheduler schedules out the software thread from the hardware thread and the transaction is aborted), and (iii) the longer the transactions last and the larger their data sets are, the greater the probability that running transactions are aborted due to (read-write) data conflicts among them.

### 2.4.3 Implementation Details

Figure 2.13 presents *ColorTM* in detail. *ColorTM* distributes the vertices of the graph across multiple threads, which color the vertices of the graph through one single parallel step (lines 4-29): multiple parallel threads repeatedly iterate over each vertex of the graph until a valid coloring on each vertex is performed.

For each vertex  $v$ , there are two sub-steps. In the first sub-step (lines 6-13), the parallel thread keeps track (i) the forbidden set of colors assigned to the adjacent vertices of the vertex  $v$  (line 10), and (ii) the critical adjacent vertices of the vertex  $v$  (lines 11-12), which are the uncolored adjacent vertices assigned to different parallel threads (line 11), and then computes a speculative color  $k$  that is permissible for the vertex  $v$  using the `compute_speculative_color()` function (line 13). In

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 Let  $tid$  be the unique id of each parallel thread
4 for each  $v \in V$  do in parallel
5   RETRY:
6   // Speculative Computation
7    $R = \emptyset$  // Track Forbidden Colors
8    $C = \emptyset$  // Track Critical Adjacent Vertices
9   for each  $u \in N(v)$  do
10     $R = R \cup u.color$ 
11    if  $((hasColor(u) == false) \ \&\& \ (get\_threadID(u) \neq tid))$ 
12       $C = C \cup u$  // Critical Adjacent Vertices Are the Uncolored Vertices Assigned to
        Another Thread
13     $k = compute\_speculative\_color(R)$  // Compute a Speculative Color  $k$  for the Vertex
         $v$ 
14    // Validate Coloring
15    if  $(C == \emptyset)$  // Skip the Validation Step, If There Are No Critical Adjacent Vertices
16       $v.color = k$ 
17    else
18      begin\_transaction
19      bool valid = true
20      for each  $u \in C$  do // Validate the Colors of the Critical Adjacent Vertices Over the
        Speculative Color
21        if  $(u.color == k)$ 
22          valid = false
23        break
24      if  $(valid == true)$  // If the Validation Succeeded, Assign the Speculative Color to
        the Vertex  $v$ 
25         $v.color = k$ 
26      end\_transaction
27      else // If the Validation Failed, Immediately Retry to Find a New Color for the
        Vertex  $v$ 
28        end\_transaction
29        goto RETRY // Eager Resolution

```

Figure 2.13: The *ColorTM* algorithm.

the second sub-step (lines 14-29), the parallel thread validates and assigns (if allowed) the speculative color  $k$  to the vertex  $v$  using data synchronization via HTM (lines 18-29). Specifically, the colors of the critical adjacent vertices are compared to the speculative color  $k$  within a hardware transaction (lines 20-23) to ensure that the color  $k$  is still permissible to be assigned to the vertex  $v$ . If the validation succeeds (line 24), the color  $k$  is assigned to the vertex  $v$  within the same transaction (line 25) to ensure correctness: recall that the reads on the colors of the critical adjacent vertices need to be executed *atomically* with the write-update on the color of the vertex  $v$ . Instead, if the validation step fails due to a coloring inconsistency appearing during runtime (line 27), the parallel thread *repeatedly* and *eagerly* retries to find a new permissible color for the current vertex  $v$  (line 29). Note that if there are *no* critical adjacent vertices to be validated (line 15), the speculative color  $k$  is directly assigned to the vertex  $v$  *without* using synchronization (line 16).

Note that in the second sub-step (lines 14-29), *ColorTM* does *not* check if the colors of the critical adjacent vertices have not been modified since the first sub-step (lines 6-13). Instead, the validation of the second sub-step *only* checks that the colors of the critical adjacent vertices are different from the

speculative color  $k$  computed in the first sub-step (line 13). In the meantime, different parallel threads may have just assigned new colors to critical adjacent vertices, which however are different from the color  $k$ , and thus causing *no* coloring inconsistencies. In that scenario, the validation of the second sub-step succeeds. This way, *ColorTM* provides high levels of parallelism: multiple parallel threads that have just assigned *different* colors than the color  $k$  to critical adjacent vertices of the vertex  $v$  will *not* cause any validation failure in the critical section of the vertex  $v$ , and the corresponding running transaction will be safely committed.

#### 2.4.4 Progress and Correctness

We clarify in detail how *ColorTM* resolves the race conditions that may arise during runtime. There are two race conditions that may cause coloring inconsistencies in multithreaded executions. First, while a parallel thread computes a speculative color  $k$  for the vertex  $v$  (lines 9-13 of Figure 2.13), different parallel threads may have just assigned the color  $k$  to one or more adjacent vertices of the vertex  $v$ . In that scenario, the validation step of lines 20-23 of Figure 2.13 fails (line 22, 27), since the speculative color  $k$  has been assigned to at least one critical adjacent vertex (line 21). Then, the corresponding parallel thread will retry to find a new permissible color for the vertex  $v$  (line 29). Second, a race condition arises when  $n$  parallel threads (assuming  $n \geq 1$ ) attempt to write-update the same color  $k$  to  $n$  adjacent vertices (fully connected adjacent vertices) within  $n$  different running transactions. In that scenario, the HTM mechanism detects read-write data conflicts on running transactions, because one (or more) running transaction attempts to write to the read-sets of another running transactions. Recall that the colors of the critical adjacent vertices are included in the read-set of each running transaction (lines 21 of Figure 2.13). Then, the HTM mechanism aborts  $n - 1$  running transactions, and commits only *one* of them. When the aborted  $n - 1$  transactions retry (each transaction can retry up to a predefined number of times), the validation step of lines 20-23 fails (lines 27 of Figure 2.13), since at that time the  $n - 1$  parallel threads observe that there is one critical adjacent vertex that has just been assigned to the color  $k$  (the committed transaction). Afterwards, since the validation failed, the  $n - 1$  parallel threads will retry to find new permissible colors for their current vertices (lines 27-29 of Figure 2.13).

Finally, we clarify that *ColorTM* provides forward progress and eventually terminates: each parallel thread retries to find a new permissible color for a current vertex  $v$  (line 29 of Figure 2.13) up to a limited number of retries. Specifically, a parallel thread retries to find a new color for a vertex  $v$ , when the validation step of lines 20-23 of Figure 2.13 fails. However, for each vertex  $v$  the validation step can fail up to a bounded number of times: the validation step fails when one (or more) critical adjacent vertex has been assigned to the same color  $k'$  with the speculative color  $k$  computed for the vertex  $v$ . Therefore, in the worst case, the validation step might fail up to  $\deg(v)$  times, where  $\deg(v)$  is the adjacency degree of the vertex  $v$ . When all  $v$ 's adjacent vertices have obtained a color, there are *no* critical adjacent vertices to be validated (line 15 of Figure 2.13), and thus, the speculative color  $k$  is directly assigned to the vertex  $v$  (line 16 of Figure 2.13), and the validation step is omitted. As a result, each parallel thread retries to find a color for each vertex  $v$  of the graph at most  $\deg(v)$

times. However, in our evaluation, we find that the validation step fails only for *a few* times: across all our evaluated large real-world graphs (Table 2.1) and using a large number of parallel threads (up to 56 threads) the validation step failures are less than 0.01%. Overall, we conclude that *ColorTM* *correctly* handles all the race conditions that may arise in multithreaded executions of the graph coloring kernel, and *effectively* terminates with a valid coloring.

```

1 Input: Graph  $G=(V,E)$ 
2 Let  $N(v)$  be the adjacent vertices of the vertex  $v$ 
3 Obtain an initial coloring on  $G$ 
4 Let  $C$  be the number of colors produced
5 Let  $b = V/C$  be the perfect balance
6 Let  $Q$  be the set of vertices of the over-full color classes
7 for each  $v \in Q$  do in parallel
8   Let  $c$  be the current color of the vertex  $v$ 
9   if (the size of the color class  $c \leq b$ )
10     continue // Color Class is Balanced
11   RETRY:
12     // Speculative Computation
13      $R = \emptyset$  // Track Forbidden Colors
14      $C = \emptyset$  // Track Critical Adjacent Vertices
15     for each  $u \in N(v)$  do
16        $R = R \cup u.\text{color}$ 
17       if ((isOverFull( $u.\text{color}$ ) == true) && (get_threadID( $u$ ) != tid))
18          $C = C \cup u$  // Critical Adjacent Vertices Are the Vertices of Over-Full Color
19         Classes That Are Assigned to Another Thread
20      $k = \text{compute\_speculative\_color}(R)$ 
21     Let  $k$  be the index of the minimum under-full color class that is permissible
22     to the vertex  $v$ 
23     if ( $k$  exists) // Validate Coloring
24       if ( $C == \emptyset$ ) // Skip the Validation Step, If There Are No Critical Adjacent Vertices
25          $v.\text{color} = k$ 
26         Atomically decrease the size of the color class  $c$ 
27         Atomically increase the size of the color class  $k$ 
28       else
29         begin\_transaction
30         bool valid = true
31         for each  $u \in C$  do // Validate the Colors of the Critical Adjacent Vertices Over
32         the Speculative Color
33           if ( $u.\text{color} == k$ )
34             valid = false
35             break
36         if (valid == true) // If the Validation Succeeded, Set the Speculative Color to
37         the Vertex  $v$ 
38            $v.\text{color} = k$ 
39           end\_transaction
40           Atomically decrease the size of the color class  $c$ 
41           Atomically increase the size of the color class  $k$ 
42         else // If the Validation Failed, Immediately Retry to Find a New Color for the
43         Vertex  $v$ 
44           end\_transaction
45           goto RETRY // Eager Resolution
46     else
47       continue

```

Figure 2.14: The *BalColorTM* algorithm.



### 2.4.5 The *BalColorTM* Algorithm

Figure 2.14 presents the *balanced* counterpart of *ColorTM*, named as *BalColorTM*. Similarly to CLU and VFF, in *BalColorTM* (i) only the vertices of the over-full color classes are considered for re-coloring, i.e., to be moved from over-full to under-full color classes in order to achieve high vertex-balance across color classes, and (ii) graph coloring balance is achieved *without increasing* the number of color classes produced by the initial graph coloring (e.g., using *ColorTM*).

Similarly to *ColorTM*, *BalColorTM* (Figure 2.14) has one single parallel step (lines 7-42): multiple parallel threads repeatedly iterate over each vertex of the over-full color classes until either a valid re-coloring to an under-full class is performed, or there is no permissible re-coloring for this vertex to an under-full color class (line 42). For each vertex of an over-full color class  $c$ , there are two sub-steps. In the first sub-step (lines 8-20), the parallel thread keeps track the forbidden set of colors assigned to the adjacent vertices of the vertex  $v$  (line 16), and the set of the critical adjacent vertices (lines 17-18) of the vertex  $v$ . In *BalColorTM*, note that the *critical* adjacent vertices of a vertex  $v$  (line 17) are the adjacent vertices that (i) belong to *over-full* color classes (recall that the vertices assigned under-full color classes are *not* considered to be re-colored/moved, and thus they do not cause any coloring inconsistency during runtime), and (ii) are assigned to different threads compared to the parallel thread in which the vertex  $v$  is assigned to. Then, the parallel thread speculatively computes a color  $k$  of an under-full color class that is permissible to be assigned to the vertex  $v$  (lines 19-20). If a permissible color  $k$  exists (without increasing the number of color classes produced by the initial graph coloring), the parallel thread attempts to assign the speculative color  $k$  to the vertex  $v$  in the second sub-step (lines 21-42). If there is *no* permissible color  $k$  of an under-full color class (line 41), the parallel threads continue to process the next vertices (line 42). In the second sub-step, if there are critical adjacent vertices that need to be validated, the parallel thread validates the speculative color  $k$  over the colors of the critical adjacent vertices within an HTM transaction (lines 27-39). If the validation succeeds (line 33), the parallel thread moves the vertex  $v$  from the color class  $c$  to the color class  $k$  by re-coloring it (line 34), and atomically updates the sizes of the color classes  $c$  and  $k$  (lines 36-37) accordingly. If the validation step fails due to a coloring inconsistency appearing during runtime (line 38), the parallel thread *eagerly* retries to find a new permissible color of an under-full color class for the vertex  $v$  (line 40). Finally, note that *BalColorTM* iterates over the vertices of each over-full color class until that particular over-full class becomes balanced at a certain point in the execution (lines 9-10), i.e., until the size of the particular color class becomes smaller or equal to  $b = V/C$ . Then, the vertices belonging to that color class are no longer considered for re-coloring (line 10). Overall, *BalColorTM* terminates when either vertex-balance across color classes is achieved or vertex-balance across color classes is no longer available, i.e., there are no more permissible re-colorings for any vertex belonging to an over-full color class.

Similarly to *ColorTM*, *BalColorTM* *completely* avoids barrier synchronization, since it includes only one single parallel step, thus minimizing synchronization costs. Moreover, it also integrates an *eager* approach to detect and resolve coloring conflicts appearing during runtime among parallel threads, that concurrently move vertices from over-full to under-full color classes. With the eager color-

ing policy, *BalColorTM* provides high performance by minimizing access latency costs to application data. Finally, *BalColorTM* effectively implements short critical sections (short running transactions with small transaction footprints) by (i) speculatively performing the computations to find permissible colors for the vertices of the over-full color classes *outside* the critical section (lines 9-13), and (ii) accessing inside the critical sections *only* the necessary data to ensure correctness, i.e., for each vertex  $v$  *BalColorTM* *only* accesses the colors of a small subset of  $v$ 's adjacent vertices (critical adjacent vertices). Via short running transactions, *BalColorTM* achieves low synchronization costs and provides high amount of parallelism.

## 2.5 Evaluation Methodology

We conduct our evaluation using a 2-socket Intel Haswell server with an Intel Xeon E5-2697 v3 processor with 28 physical cores and 56 hardware threads. The processor runs at 2.6 GHz and each physical core has its own L1 and L2 caches of sizes 32 KB and 256 KB, respectively. Each socket includes a shared 35 MB L3 cache. We statically pin each software thread to a hardware thread, and enable hyperthreading only on 56-thread executions, unless otherwise stated. In our evaluation (Section 2.6), the numbers reported are averaged across 5 runs of each experiment.

Table 2.1 shows the characteristics of the large real-world graphs used in our evaluation. We select 18 representative graphs from the Suite Matrix Collection that vary in vertex and graph degrees, and are used in different application domains. For each graph, Table 2.1 presents the number of vertices (#vertices), the number of edges (#edges), the maximum ( $deg_{max}$ ) degree, the average ( $deg_{avg}$ ) degree and the standard deviation of the vertices' degrees ( $deg_{std}$ ), and the last column of this table shows the ratio of the standard deviation of the vertices' degrees to the average degree ( $\frac{deg_{std}}{deg_{avg}}$ ).

## 2.6 Evaluation

This section evaluates the proposed *ColorTM* and *BalColorTM* algorithms. First, we compare the coloring quality and the performance over prior state-of-the-art graph coloring algorithms, as well as the execution behavior of *ColorTM* (Section 2.6.1). Second, we compare the color balancing quality and the performance of *BalColorTM* over prior state-of-the-art balanced graph coloring algorithms, as well as the execution behavior of *BalColorTM* (Section 2.6.2). Finally, we evaluate the performance of Community Detection [355] by parallelizing it using *ColorTM* and *BalColorTM* (Section 2.6.3) via chromatic scheduling.

### 2.6.1 Analysis of Parallel Graph Coloring Algorithms

We compare the following parallel graph coloring implementations:

- The sequential Greedy algorithm presented in Figure 2.1.
- The SeqSolve algorithm presented in Figure 2.2.
- The IterSolve algorithm presented in Figure 2.3.
- The IterSolveR algorithm presented in Figure 2.4.

Graph Name	#Vertices	#Edges	deg <sub>max</sub>	deg <sub>avg</sub>	deg <sub>std</sub>	$\frac{\text{deg}_{\text{std}}}{\text{deg}_{\text{avg}}}$
Queen_4147 ( <b>qun</b> )	4147110	329499284	81	79.45	6.34	0.080
Geo_1438 ( <b>geo</b> )	1437960	63156690	57	43.92	4.39	0.100
Flan_1565 ( <b>fln</b> )	1564794	117406044	81	75.03	11.43	0.152
Bump_2911 ( <b>bum</b> )	2911419	127729899	195	43.87	6.96	0.159
Serena ( <b>ser</b> )	1391349	64531701	249	46.38	9.24	0.199
delaunay_n24 ( <b>del</b> )	16777216	100663202	26	5.99	1.34	0.222
rgg_n_2_23_s0 ( <b>rgg</b> )	8388608	127002786	40	15.14	3.89	0.257
kmer_A2a ( <b>kmr</b> )	170728175	360585172	40	2.11	0.57	0.267
cage15 ( <b>cag</b> )	5154859	99199551	47	19.24	5.73	0.298
road_usa ( <b>usa</b> )	23947347	57708624	9	2.41	0.93	0.386
dielFilterV3real ( <b>dlf</b> )	1102824	89306020	270	80.98	36.56	0.451
audikw_1 ( <b>aud</b> )	943695	77651847	345	82.29	42.44	0.516
vas_stokes_2M ( <b>vas</b> )	2146677	65129037	637	30.34	37.18	1.226
stokes ( <b>stk</b> )	11449533	349321980	720	30.51	41.44	1.358
uk-2002 ( <b>uk</b> )	18520486	298113762	2450	16.10	27.53	1.710
soc-LiveJournal1 ( <b>soc</b> )	4847571	68993773	20293	14.23	36.08	2.535
arabic-2005 ( <b>arb</b> )	22744080	639999458	9905	28.14	78.84	2.802
FullChip ( <b>fch</b> )	2987012	26621990	2312481	8.91	1806.80	202.725

Table 2.1: Large Real-World Graph Dataset.

- A variant of our proposed algorithm (Figure 2.13) that uses fine-grained locking instead of HTM, henceforth referred to as ColorLock. Specifically, each vertex of the graph is associated with a software-based lock. In the beginning of the critical section (line 18 in Figure 2.13), parallel threads acquire the corresponding locks of both the current vertex  $v$  and the *critical* adjacent vertices of the vertex  $v$ . Then, when the critical section ends (lines 26 and 28 in Figure 2.13), parallel threads release the acquired locks. To avoid deadlocks, we impose a global order when acquiring/releasing locks based on the vertices' id: parallel threads acquire/release locks of multiple vertices starting from the lock associated with the vertex with the smallest vertex id, iterating via an increasing order of the vertices' ids, and finishing to the lock associated with the vertex with the highest vertex id.
- Our proposed *ColorTM* algorithm (Figure 2.13) that leverages HTM. Each transaction can retry up to 50 times, before resorting to a non-transactional fallback path. The non-transactional path is a coarse-grained locking solution for the critical section (lines 18-28 in Figure 2.13).

For a fair comparison, in all graph coloring schemes we color the vertices in the order they appear in the input graph representation (first-fit ordering heuristic [57]).

### Analysis of the Coloring Quality

Table 2.2 compares the coloring quality of all parallel graph coloring implementations in single-threaded and multithreaded executions.

Coloring Scheme	1 thread	14 threads	28 threads	56 threads
Greedy	42.58	-	-	-
SeqSolve	42.58	42.34	42.33	42.18
IterSolve	42.58	44.05	43.94	44.04
IterSolveR	42.58	43.61	43.88	44.58
ColorLock	42.58	45.75	45.67	46.14
<b>ColorTM</b>	42.58	46.20	45.77	46.28

Table 2.2: The geometric mean on the number of colors produced across all large real-world graphs (lower is better) for each parallel graph coloring implementation using one core (1 thread), all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

We make two key observations. First, there is low variability on the number of colors used across the different graph coloring schemes. The parallel graph coloring schemes provide similar graph coloring quality, because the number of colors produced is primarily determined by the order in which the vertices are colored [57, 58]. In this work, we use the *first-fit* ordering heuristic in all schemes, i.e., coloring the vertices in the order they appear in the input graph representation, and we leave the experimentation of other ordering heuristics for future work. Second, we find that in most schemes the coloring quality becomes slightly worse as the number of threads increases. As the number of threads increases, the number of coloring conflicts that arise during runtime typically increases, and thus parallel threads might resolve coloring inconsistencies by introducing a few additional color classes. The SeqSolve scheme does not typically increase the number of colors used in multithreaded executions, because the coloring inconsistencies are resolved using one single thread. Overall, we conclude that since all graph coloring schemes employ the same ordering heuristic, they provide similar coloring quality.

## Performance Comparison

Figure 2.15 evaluates the scalability achieved by all parallel graph coloring implementations in our large real-world graphs, when increasing the number of threads from 1 to 56, i.e., the maximum available hardware thread capacity of our machine.

We draw three findings. First, *ColorTM* and ColorLock achieve the lowest execution time across all schemes in single-threaded executions. Using one single thread, *ColorTM* and ColorLock on average outperform SeqSolve by  $1.55\times$  and  $1.42\times$ , respectively, and they on average outperform IterSolve by  $1.17\times$  and  $1.06\times$ , respectively. With only one thread, *ColorTM* and ColorLock have identical executions to the sequential Greedy algorithm (Figure 2.1): thanks to the optimizations proposed in Section 2.4.2, the list of critical adjacent vertices that need to be validated inside the critical section is empty, and thus *ColorTM* and ColorLock *completely* eliminate using synchronization (either HTM or fine-grained locking). Second, we find that IterSolveR exhibits the lowest scalability across all schemes. IterSolveR merges two parallel for-loops into a single parallel for-loop in order to eliminate one of the two barriers used in IterSolve. Even though IterSolveR reduces the barrier synchronization

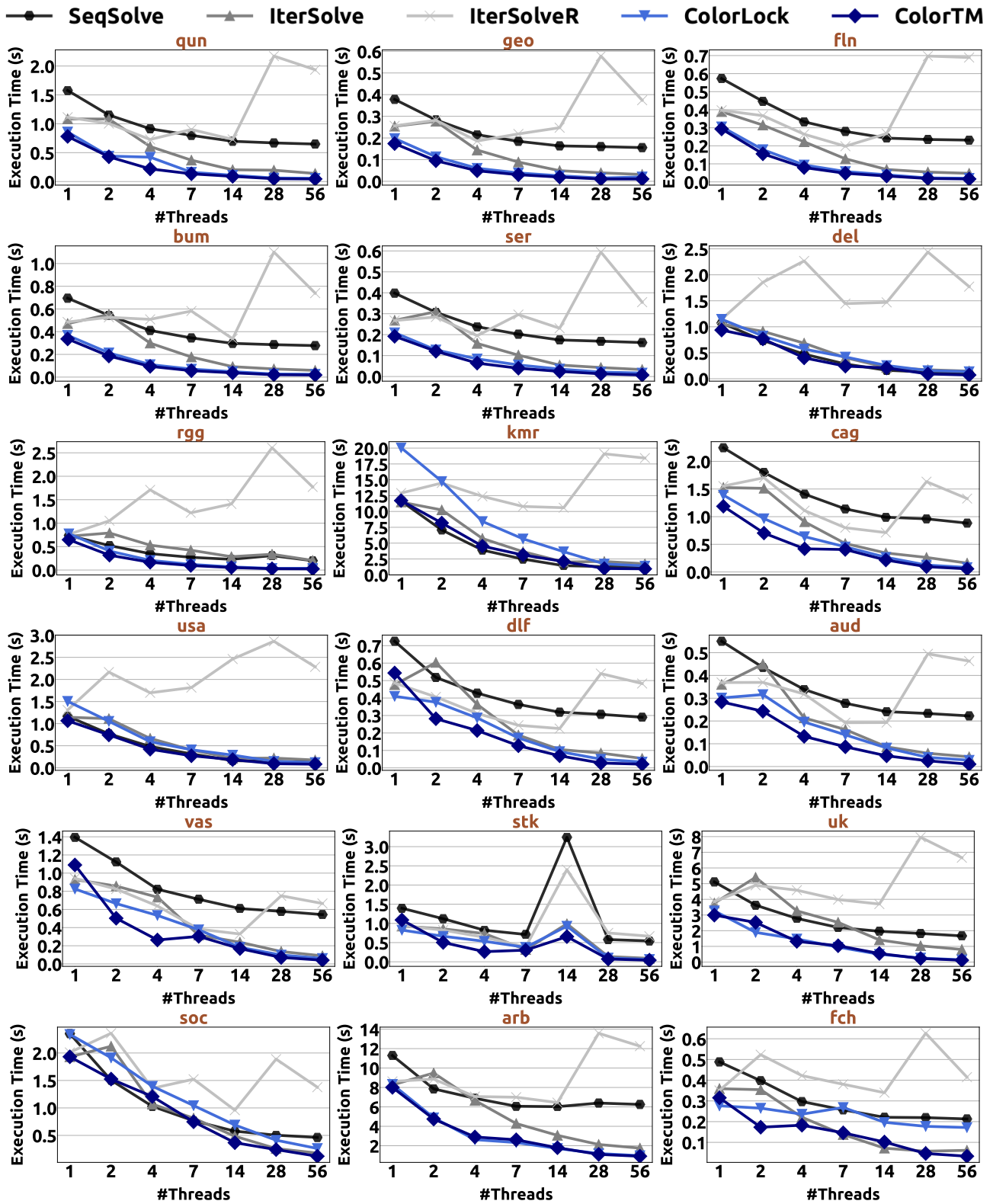


Figure 2.15: Scalability achieved by all parallel graph coloring implementations in large real-world graphs.

costs, it increases the load imbalance among parallel threads, thus causing significant performance overheads. Third, we observe that the scalability of SeqSolve, IterSolve, and IterSolveR is highly affected by the NUMA effect, i.e., the non-uniform memory access latencies to the application data. For example, when increasing the number of threads from 7 to 14 (only one NUMA socket is used) the

performance of SeqSolve, IterSolve, IterSolveR, ColorLock and *ColorTM* improves by  $1.24\times$ ,  $1.75\times$ ,  $1.06\times$ ,  $1.62\times$  and  $1.65\times$ , respectively, averaged across all large graphs. However, when increasing the number of threads from 14 to 28, i.e., using both NUMA sockets of our machine, the performance of SeqSolve and IterSolve *only* improves by  $1.03\times$  and  $1.26\times$ , respectively, while the performance of and IterSolveR decreases by  $2.13\times$ , averaged across all large graphs. In contrast, when increasing the number of threads from 14 to 28, the performance of ColorLock and *ColorTM* significantly improves by  $1.77\times$  and  $1.97\times$ , respectively, averaged across all graphs. This is because our proposed algorithmic design implemented in ColorLock and *ColorTM* leverages better the deep memory hierarchy of commodity multicore platforms thanks to its *eager* conflict detection and resolution policy, thus achieving lower data access costs. Overall, we conclude that our proposed algorithmic design achieves the best scalability in modern multicore platforms.

Figure 2.16 compares the speedup achieved by all schemes over the sequential Greedy scheme, when varying the number of hardware threads used in all large real-world graphs.

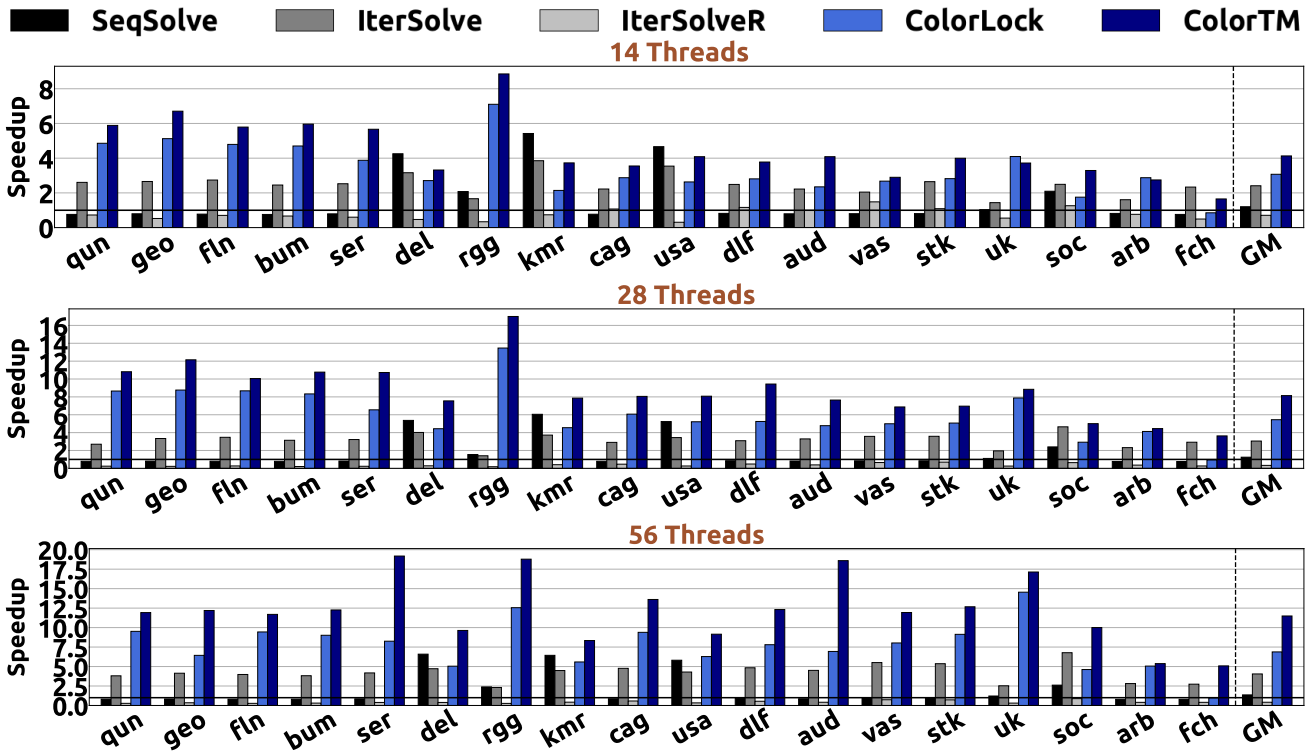


Figure 2.16: Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

We make two key observations. First, all parallel graph coloring schemes achieve lower speedup in very irregular graphs including the soc, arb and fch graphs, compared to all the remaining real-world graphs. In very irregular graphs, the number of edges per vertex significantly vary across vertices [10, 109, 155]: typically only a few vertices have a much larger number of edges over the vast majority of the remaining vertices of the graph. Therefore, in irregular graphs parallel threads typically cause more coloring inconsistencies than regular graphs, which are resolved during run-

time, increasing the execution time. Second, we find that *ColorTM* achieves significant performance improvements over all the prior state-of-the-art graph coloring schemes. *ColorTM* outperforms SeqSolve, IterSolve, and IterSolveR by  $3.43\times$ ,  $1.71\times$  and  $5.83\times$  respectively, when using 14 threads, and by  $8.46\times$ ,  $2.84\times$  and  $27.66\times$  respectively, when using the maximum hardware thread capacity of our machine (56 threads). This is because SeqSolve, IterSolve, and IterSolveR traverse *all* the vertices of the graph at least twice, and employ a *lazy* conflict resolution policy, thus incurring high data access costs. Instead, *ColorTM* traverses more than once *only* the conflicted vertices, and resolves coloring inconsistencies with an *eager* approach, thus better leveraging the deep memory hierarchy of multi-core platforms and reducing data access costs. In addition, *ColorTM* outperforms ColorLock by  $1.34\times$  and  $1.67\times$  when using 14 and 56 threads, respectively. As explained, HTM is a speculative hardware-based synchronization mechanism, and thus *ColorTM* provides high performance improvements over ColorLock thanks to significantly minimizing data access and synchronization costs. Note that in the fine-grained locking approach of ColorLock, for each adjacent vertex accessed inside the critical section, the parallel thread needs to acquire and release the corresponding software-based lock, thus performing additional memory accesses in the memory hierarchy for accessing the lock variable. Overall, we conclude that *ColorTM* significantly outperforms all prior state-of-the-art parallel graph coloring algorithms across a wide variety of large real-world graphs.

To confirm the performance benefits of *ColorTM* across multiple computing platforms, we evaluate all schemes on a 2-socket Intel Broadwell server with an Intel Xeon E5-2699 v4 processor at 2.2 GHz having 44 physical cores and 88 hardware threads. Figure 2.17 compares the speedup achieved by all schemes over the sequential Greedy scheme in all large real-world graphs using 88 threads, i.e., the maximum hardware thread capacity of the Intel Broadwell server. We find that *ColorTM* provides significant performance benefits over prior state-of-the-art graph coloring algorithms, achieving  $11.98\times$ ,  $4.33\times$  and  $22.06\times$  better performance over SeqSolve, IterSolve, and IterSolveR, respectively.

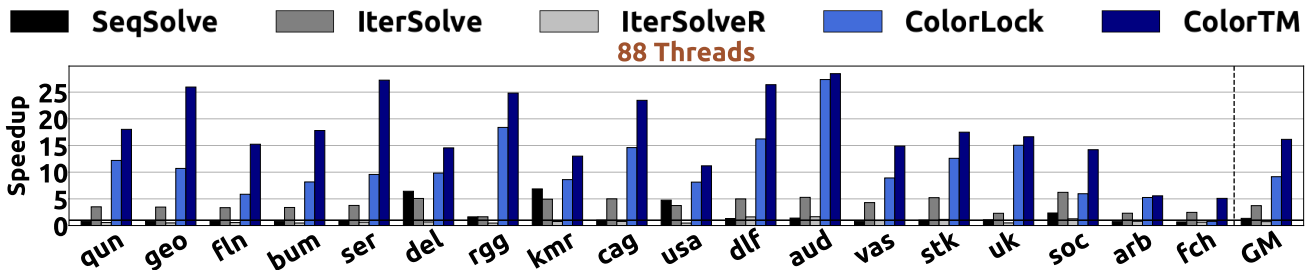


Figure 2.17: Speedup achieved by all parallel graph coloring implementations over the sequential Greedy scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads).

### Analysis of *ColorTM* Execution

We further analyze the HTM-related execution behavior of our proposed *ColorTM* and *BalColorTM* algorithms. Figure 2.18 presents the abort ratio of *ColorTM*, i.e., the number of transactional aborts divided by the number of attempted transactions, in all real-world graphs, as the number of threads increases. In the 14-thread execution, we pin all thread on one single NUMA socket. In the 28-thread execution, we pin threads on both NUMA sockets of our machine with hyperthreading disabled. In

the (14+14)-thread execution, we pin all 28 threads on the same *single* socket with hyperthreading enabled. In the 56-thread execution, we use the maximum hardware thread capacity of our machine.

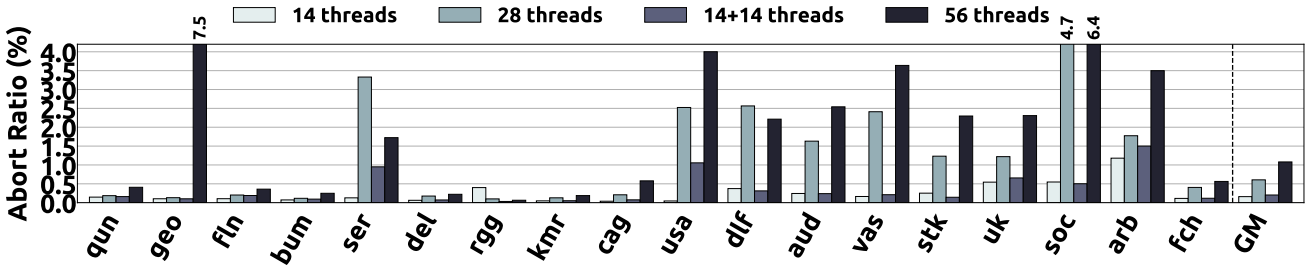


Figure 2.18: Abort ratio exhibited by *ColorTM* in all large real-world graphs.

We make three key observations. First, we find that the abort ratio becomes high in real-world graphs which have high maximum degree and high standard deviation of the vertices' degrees, e.g., *dlf*, *aud*, *vas*, *stk*, *uk*, *soc* and *arb* graphs. In graphs with high vertex degree, the transaction data access footprint is large and parallel threads compete for the same adjacent vertices with a high probability, thus causing aborts in HTM. Second, we observe that when using both sockets of our machine, the transactional aborts in *ColorTM* significantly increase due to the NUMA effect. Specifically, averaged across all graphs the (14+14)-thread execution of *ColorTM* exhibits  $2.97\times$  lower abort ratio compared to the 28-thread execution of *ColorTM*. Due to the NUMA effect, the memory accesses to the application data are very expensive. As a result, the duration of the transactions increases, thus increasing the probability of conflict aborts among running transactions (See more details in the next experiment). Third, we observe that *ColorTM* exhibits a very low abort ratio. *ColorTM* has only 1.08% abort ratio on average across all real-world graphs, when using the maximum hardware thread capacity (56 threads) of our machine. Our proposed *speculative* algorithmic design effectively reduces the amount of computations and data accesses performed inside the critical section (inside the HTM transaction), thus effectively decreasing the transaction's footprint and duration. As a result, *ColorTM* provides high amount of parallelism and low interference among parallel threads. We conclude that *ColorTM* has low synchronization and interference costs among a large number of parallel threads, even in real-world graphs with high vertex degree.

Figure 2.19 presents the breakdown of different types of aborts exhibited by *ColorTM* in a representative subset of real-world graphs. We break down the transactional aborts into four types: (i) *conflict* aborts: they appear when a running transaction executed by a parallel thread attempts to write the read-set of another running transaction executed by a different thread, (ii) *capacity* aborts: they appear when the memory footprint of a running transaction exceeds the size of the hardware transactional buffers, (iii) *lock* aborts: current HTM implementations [336, 337, 416, 417] provide no guarantee that any transaction will eventually commit inside the transactional path, and thus the programmer provides an alternative non-transactional fallback path, i.e., falling back to the acquisition of coarse-grained lock that allows only a single thread to enter the critical section, and forces aborts to the transactions of all the remaining threads <sup>2</sup>, and (iv) *other* aborts: they appear when a transac-

<sup>2</sup>To achieve this, the lock is added to each transaction's read set, so that when the lock is acquired by a thread (write to the lock variable), the remaining threads are aborted and wait until the lock is released.



tion fails due to other reasons such as cache line evictions, interrupts and/or when the duration of a transaction exceeds the scheduling quantum and the OS scheduler schedules out the software thread from the hardware thread, aborting the transaction. Note that since the fallback path lock is just a variable in the source code, some conflict aborts are caused by the writes in this lock variable. Thus, a part of the lock aborts is counted as conflict aborts in our measurements.

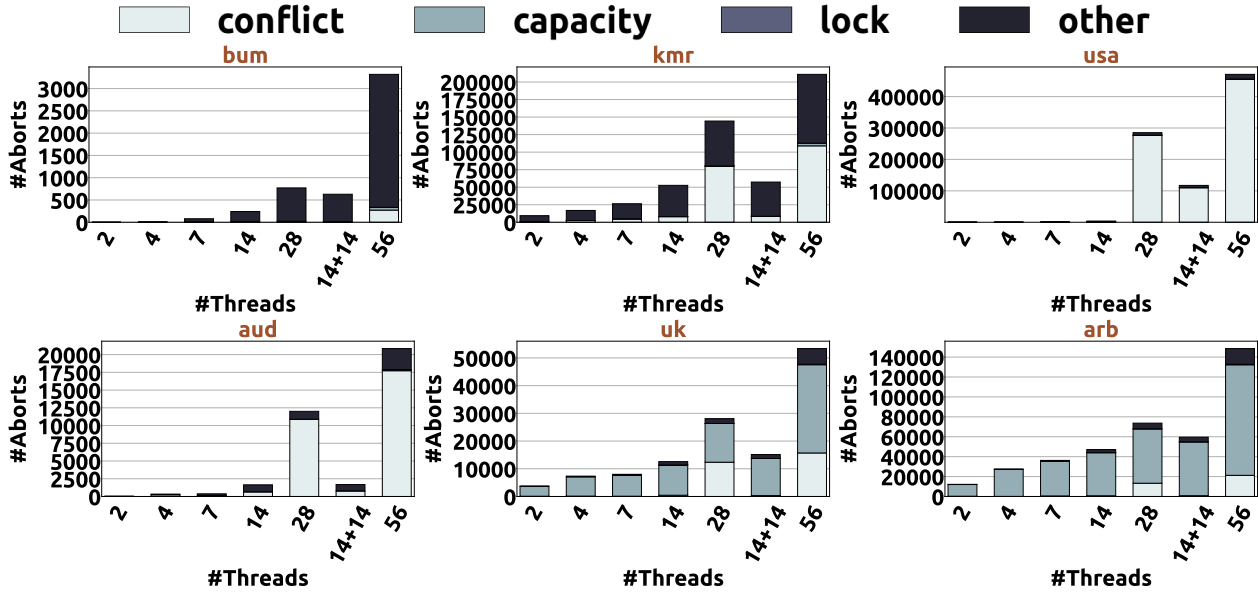


Figure 2.19: Breakdown of different types of aborts exhibited by *ColorTM* in real-world graphs.

We draw three findings. First, we find that the *conflict* aborts significantly increase across all graphs when using both sockets of our machine due to the NUMA effect. For example, the number of conflicts aborts in the 28-thread executions is  $3.32\times$  higher compared to that in the 14-thread executions. As already mentioned, the NUMA effect significantly increases the duration of the running transactions, and thus the probability of causing conflict aborts among running transactions is high. Second, as number of threads increases, e.g., when comparing the 56-thread execution over the 28-thread execution, the number of conflict aborts increases by  $1.05\times$ . This is because partitioning the graph to a higher number of threads results in a higher number of crossing edges among parallel threads, which in turn results in a larger list of critical adjacent vertices that is validated inside the HTM transactions. Therefore, the transaction footprint increases, thus increasing the probability of causing conflict aborts. Third, we find that in graphs with very high maximum degree, e.g., *uk* and *arb* graphs, the capacity aborts constitute a large portion of total aborts. In such graphs, the data access footprint of the transactions is large, resulting to a high probability of exceeding the hardware buffers. Overall, our analysis demonstrates that current HTM implementations are severely limited by the NUMA effect [335], and incur high performance costs when using more than one NUMA socket on the machine. To this end, we recommend hardware designers to improve the HTM implementations in NUMA machines, and suggest software designers to propose intelligent algorithmic schemes and data partitioning approaches that minimize the expensive memory accesses to remote NUMA sockets inside the HTM transactions.

## 2.6.2 Analysis of Balanced Graph Coloring Algorithms

We compare the following balanced graph coloring implementations:

- The CLU algorithm presented in Figure 2.5.
- The VFF algorithm presented in Figure 2.6.
- The Recoloring algorithm presented in Figure 2.7.
- Our proposed *BalColorTM* algorithm (Figure 2.14) that leverages HTM. Each transaction is retried up to 50 times, before resorting to a non-transactional fallback path. The non-transactional path is a coarse-grained lock scheme for the critical section (lines 27-39 in Figure 2.14).

For a fair comparison, in all graph coloring schemes we color the vertices in the order they appear in the color classes produced by the initial coloring.

### Analysis of Color Balancing Quality

Table 2.3 compares the quality of balance in the color class sizes produced by the balanced-oblivious *ColorTM* and all our evaluated balanced graph coloring implementations. Similarly to [49], we evaluate the color balancing quality using the relative standard deviation of the color class sizes expressed in %, which is defined as the ratio of the standard deviation of the color class sizes to the average color class size. The closer the value of this metric is to 0.00, the better is the color balance. For the *ColorTM* and Recoloring schemes, we also include in parentheses the number of color classes produced. As already explained in Section 2.2.3, the CLU, VFF, and *BalColorTM* schemes produce the same number of color classes with the initial coloring. In this experiment, we evaluate all algorithms using the maximum hardware thread capacity of our machine, i.e., 56 threads, in order to evaluate the color balancing quality of all schemes using the maximum available parallelism provided by the underlying hardware platform.

We draw three findings from Table 2.3. First, we observe that the balanced-oblivious *ColorTM* scheme incurs very high disparity in the sizes of the color classes produced. Specifically, the color balancing quality of *ColorTM* is  $1887.01\times$ ,  $287.70\times$ ,  $10.32\times$ , and  $4266.03\times$  worse than that of CLU, VFF, Recoloring and *BalColorTM*, respectively. Second, even though Recoloring is effective over *ColorTM* by providing better color balancing quality, its color balancing quality is the worst compared to all the remaining balanced graph coloring schemes. In addition, in highly irregular graphs (graphs with high maximum degree and high standard deviation in the vertices' degrees) such as uk, soc and arb, Recoloring significantly increases the number of color classes produced over the initial coloring. Recoloring re-colors the vertices of the graph with a different order compared to that used in the initial graph coloring scheme, which in turn may introduce new additional color classes. Third, we find that *BalColorTM* provides the best color balancing quality compared to all prior state-of-the-art balanced graph coloring schemes. Specifically, the color balancing quality of *BalColorTM* is  $2.26\times$ ,  $14.82\times$  and  $413.31\times$  better compared to that of CLU, VFF and Recoloring, respectively. Overall, we conclude that our proposed *BalColorTM* provides the best color balancing quality over prior state-of-the-art schemes in all large real-world graphs.

To better illustrate the effect of balancing the vertices across color classes, we present in Fig-

Input Graph	Initial Coloring <i>ColorTM</i>		Balanced Graph Coloring Schemes			
			CLU	VFF	Recoloring	<i>BalColorTM</i>
<b>qun</b>	63.62	(48)	0.212	1.669	14.739 (48)	0.009
<b>geo</b>	70.28	(36)	0.321	0.635	17.664 (34)	0.020
<b>fln</b>	65.42	(45)	0.576	0.611	20.384 (51)	0.044
<b>bum</b>	64.32	(36)	0.179	0.647	17.950 (33)	0.009
<b>ser</b>	73.64	(39)	0.405	0.751	16.651 (38)	0.024
<b>del</b>	100.06	(9)	0.002	0.013	35.136 (10)	0.001
<b>rgg</b>	115.30	(22)	0.018	3.783	21.799 (23)	0.003
<b>kmr</b>	189.79	(11)	0.0003	0.0002	31.492 (12)	0.0004
<b>cag</b>	122.89	(19)	0.014	0.649	34.197 (20)	0.005
<b>usa</b>	105.09	(5)	0.001	0.024	0.0005 (5)	0.0005
<b>dlf</b>	57.95	(54)	2.58	2.53	22.551 (57)	3.01
<b>aud</b>	84.02	(60)	5.243	2.780	19.498 (54)	3.575
<b>vas</b>	144.18	(38)	0.084	18.527	25.373 (34)	0.016
<b>stk</b>	141.41	(35)	0.016	17.684	25.375 (34)	0.003
<b>uk</b>	1882.66	(944)	0.437	0.237	65.994 (1355)	1.732
<b>soc</b>	945.35	(324)	1.136	1.466	58.190 (459)	1.886
<b>arb</b>	3351.79	(3248)	0.681	1.499	68.521 (4772)	3.410
<b>fch</b>	125.70	(9)	0.012	0.271	33.854 (10)	0.451

Table 2.3: Color balancing quality achieved by *ColorTM* and all balanced graph coloring implementations in the large real-world graphs. We present the relative standard deviation (in %) on the sizes of the color classes obtained by each scheme (lower is better). In *ColorTM* and Recoloring, we provide inside the parentheses the number of color classes produced. The CLU, VFF and *BalColorTM* produce the same number of color classes with the initial coloring scheme.

Figure 2.20 shows the sizes of all the color classes produced by *ColorTM*, CLU, VFF, Recoloring and *BalColorTM* for a representative subset of our evaluated real-world graphs. The uk, soc and arb graphs are web social networks [418] with a highly power-law distribution [10, 109, 155]: only a *few* vertices have a very *high* degree, while the vast majority of the remaining vertices of the graph has very low degree. In such graphs, *ColorTM* inserts the vast majority of the vertices in the first few color classes, and the remaining *few* vertices are assigned to different separate color classes. Moreover, as explained, Recoloring introduces a large number of *new* additional color classes in such real-world graphs.

## Performance Comparison

Figure 2.21 evaluates the scalability achieved by all balanced graph coloring implementations in a representative subset of our evaluated large real-world graphs, as the number of threads increases from 1 to 56, i.e., up to the maximum available hardware thread capacity of our machine. We present the execution time of *only* the kernel that balances the vertices across color classes (excluding the execution time of the initial graph coloring).

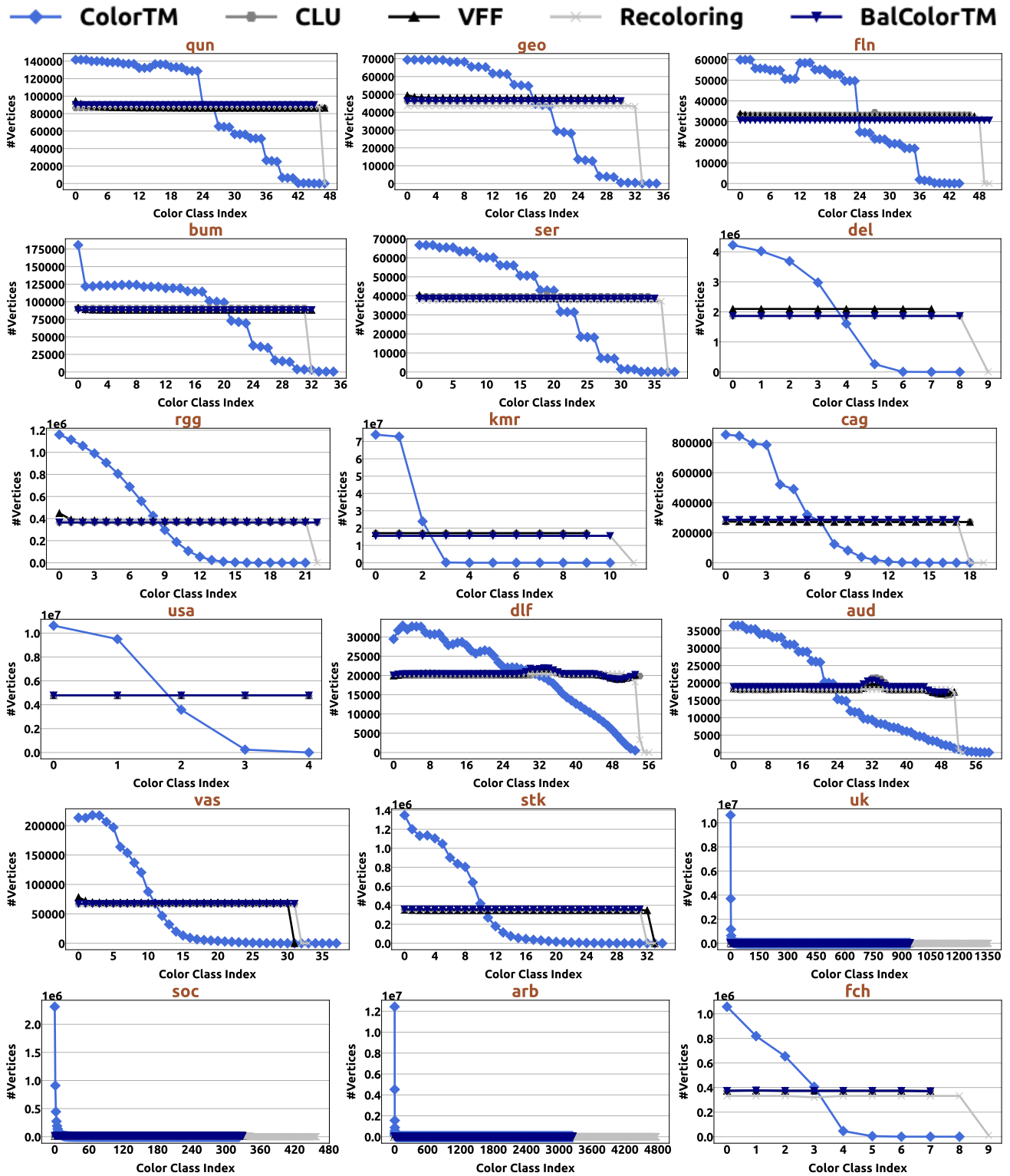


Figure 2.20: Distribution of color class sizes produced by *ColorTM* and all our evaluated balanced graph coloring schemes. Note that small color class sizes result in reduced parallelism in the real-world end-application.

We draw three findings. First, we observe that Recoloring achieves the worst performance over all balanced graph coloring schemes. Even in the single-threaded executions, Recoloring performs by  $3.21\times$ ,  $2.26\times$  and  $3.69\times$  worse than CLU, VFF and *BalColorTM*, respectively, because it executes a much larger amount of computation, memory accesses and synchronization. Recall that Recoloring

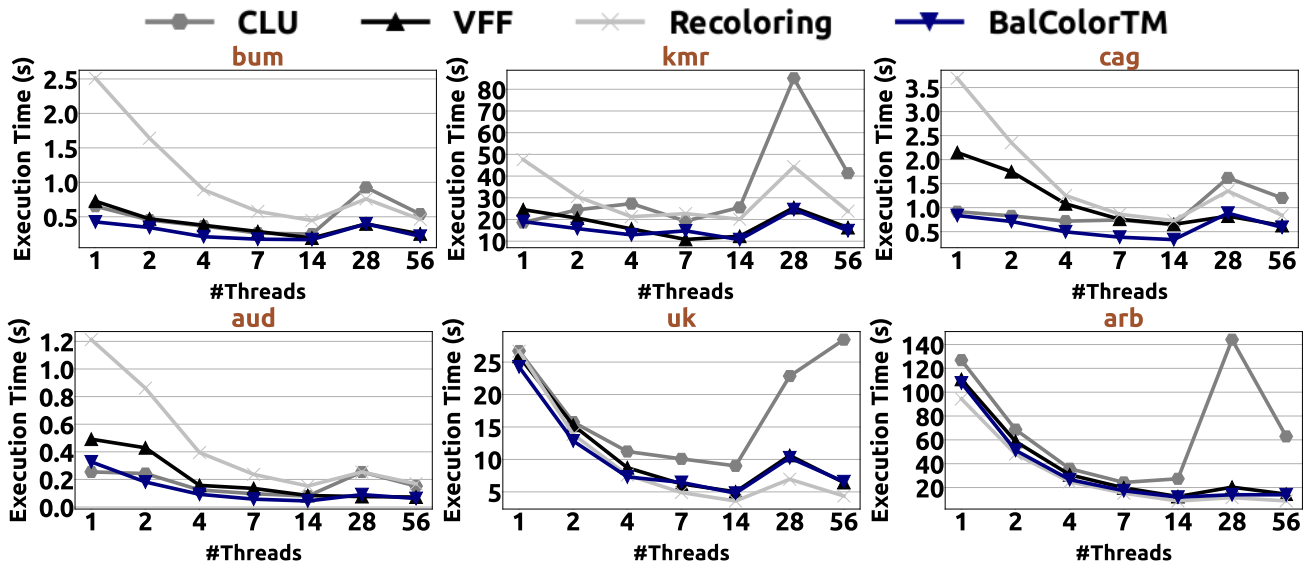


Figure 2.21: Scalability achieved by all balanced graph coloring implementations in large real-world graphs.

processes and re-colors *all* the vertices of the graph, while the remaining balanced graph coloring schemes re-color only a *subset* of the vertices of the graph. Note that in *uk* and *arb* graphs, all balanced graph coloring schemes need to re-color a *large* portion of the graph's vertices, thus performing closely to each other. Second, we find that the scalability of all schemes is affected by the NUMA effect, however *BalColorTM* on average scales well even when using all available hardware threads and both NUMA sockets of our machine. When increasing the number of threads from 28 to 56, the performance of *BalColorTM* improves by  $1.55\times$  averaged across all large graphs. Third, we find that in contrast to the graph coloring kernel, in many real-world graphs the performance of the balanced graph coloring kernel scales up to 14 threads, and degrades when using 56 threads. This is because the balanced graph coloring kernel has a lower amount of parallelism (a small subset of the vertices of the graph are re-colored by parallel threads) than the graph coloring kernel. Thus, our analysis demonstrates that when a kernel has low levels of parallelism, the best performance is achieved using a smaller number of parallel threads than the available hardware threads on the multicore platform. To this end, we suggest software designers of real-world end-applications to *on-the-fly* adjust the number of parallel threads used to parallelize each different sub-kernel of the end-application based on the parallelization needs of each particular sub-kernel.

Figure 2.22 compares the speedup achieved by all balanced graph coloring schemes normalized to the CLU scheme in all large real-world graphs. We compare the actual kernel time that balances the vertices across color classes.

We observe that *BalColorTM* outperforms all prior state-of-the-art balanced graph coloring schemes across all various large real-world graphs with a large number of parallel threads used. *BalColorTM* outperforms CLU, VFF and Recoloring by on average  $1.89\times$ ,  $1.33\times$  and  $2.06\times$  respectively, when using 14 threads. Moreover, *BalColorTM* outperforms CLU, VFF and Recoloring by on average  $2.61\times$ ,  $1.05\times$  and  $1.68\times$  respectively, when using 56 threads, i.e., the maximum hardware thread capacity of our machine. Overall, *BalColorTM* performs best over all prior schemes in all large real-world graphs. Therefore, considering the fact that *BalColorTM* also provides the best color balancing quality over

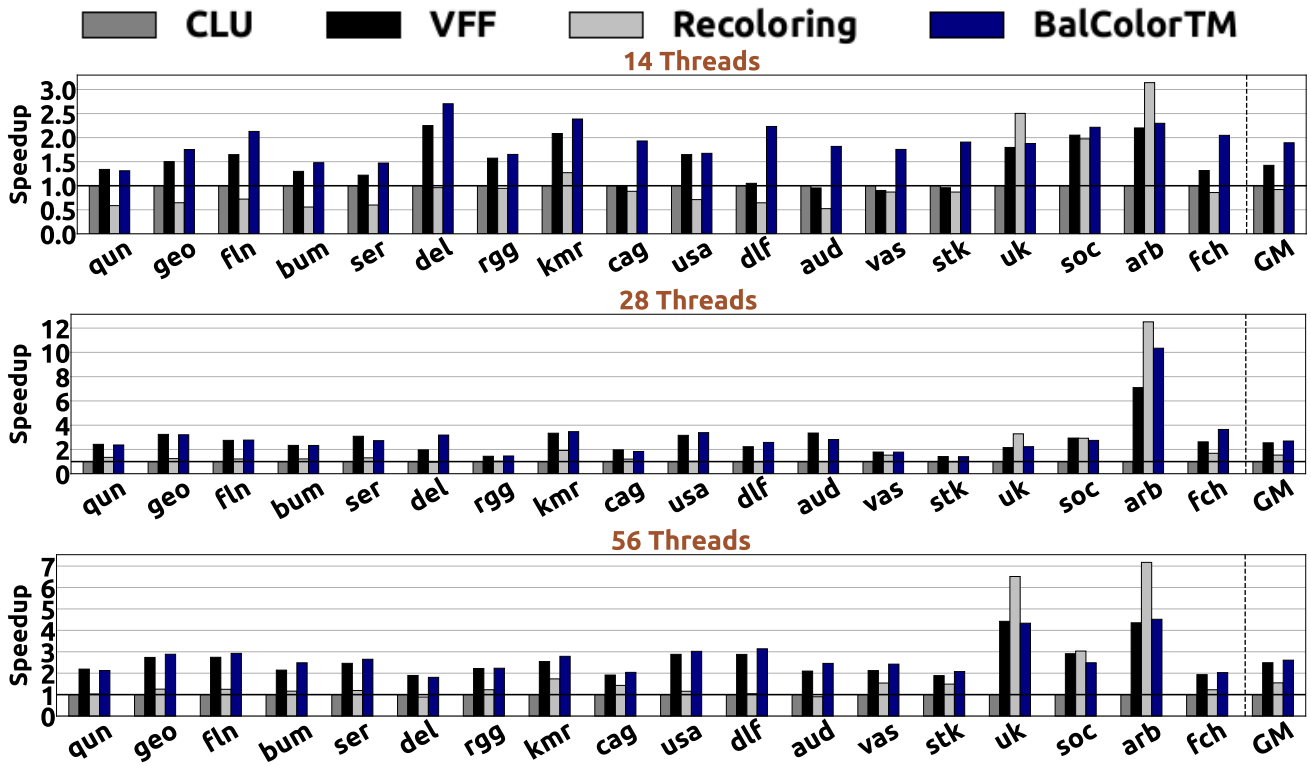


Figure 2.22: Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

prior schemes, we conclude that our proposed algorithmic design is a highly efficient and effective parallel graph coloring algorithm for modern mutlicore platforms.

To confirm the performance benefits of *BalColorTM* across multiple computing platforms, we evaluate all schemes on a 2-socket Intel Broadwell server with an Intel Xeon E5-2699 v4 processor at 2.2 GHz having 44 physical cores and 88 hardware threads. Figure 2.23 compares the speedup achieved by all balanced graph coloring schemes normalized to the CLU scheme in all large real-world graphs using 88 threads, i.e., the maximum hardware thread capacity of the Intel Broadwell server. We find that *BalColorTM* provides significant performance benefits over prior state-of-the-art graph coloring algorithms, achieving  $1.82\times$ ,  $1.22\times$  and  $1.84\times$  better performance over CLU, VFF, and Recoloring, respectively.

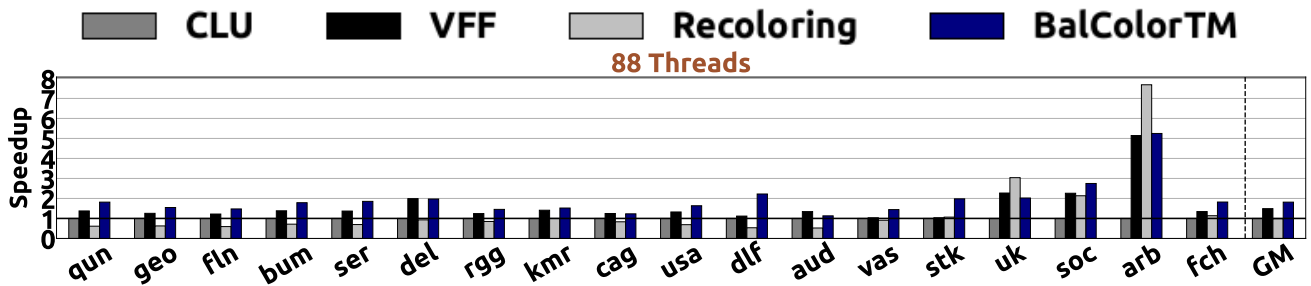


Figure 2.23: Speedup achieved by all balanced graph coloring implementations over the CLU scheme in large real-world graphs using the maximum hardware thread capacity of an Intel Broadwell server with hyperthreading enabled (88 threads).

## Analysis of *BalColorTM* Execution

Figure 2.24 presents the abort ratio of *BalColorTM*, i.e., the number of transactional aborts divided by the number of attempted transactions, in all real-world graphs, as the number of threads increases. In the 14-thread execution, we pin all thread on one single socket. In the 28-thread execution, we pin threads on both NUMA sockets of our machine with hyperthreading disabled. In the (14+14)-thread execution, we pin all 28 threads on the same single socket with hyperthreading enabled. In the 56-thread execution, we use the maximum hardware thread capacity of our machine.

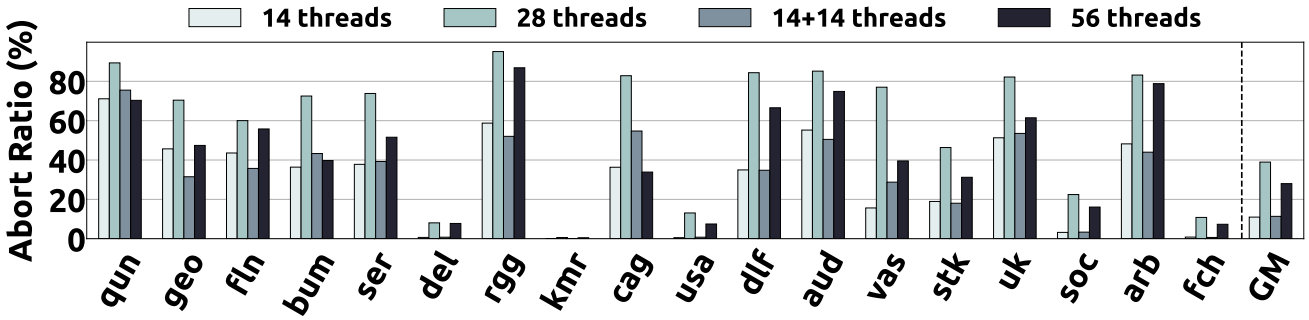


Figure 2.24: Abort ratio exhibited by *BalColorTM* in all large real-world graphs.

We make two key observations. First, we observe that *BalColorTM* on average incurs higher abort ratio over *ColorTM*, reaching up to 80% abort ratio in some multithreaded executions. Specifically, *BalColorTM* incurs  $68.55\times$ ,  $64.35\times$ ,  $55.83\times$  and  $25.91\times$  higher abort ratio (averaged across all real-world graphs) over *ColorTM*, when using 14, 28, (14+14), and 56 threads, respectively. This is because *BalColorTM* processes and re-colors a much smaller number of vertices (a small subset of the vertices of the graph) compared to *ColorTM*, which instead processes and colors *all* the vertices of the graph. As a result, parallel threads compete for the same data and memory locations with a much higher probability in *BalColorTM* compared to *ColorTM*, thus incurring higher abort ratio and synchronization costs. Second, we find that in *all* real-world graphs the vast majority of transactional aborts are *conflict* aborts. Specifically, the portion of conflict aborts is more than 95% in all real-world graphs for all multithreaded executions. Typically, the lower parallelization needs a parallel kernel has, the higher data contention among parallel threads it incurs. Overall, our analysis demonstrates that using a high number of parallel threads results in high contention on shared data due to low amount of parallelism of the balanced graph coloring kernel. The aforementioned high contention causes high synchronization overheads. To this end, we recommend software designers of real-world end-applications to design adaptive parallelization schemes that trade off the amount of parallelism provided for lower synchronization costs.

### 2.6.3 Analysis of a Real-World Scenario

In this section, we study the performance benefits of our proposed graph coloring schemes, i.e., *ColorTM* and *BalColorTM*, when parallelizing a widely used real-world end-application, i.e., Community Detection, via chromatic scheduling. Specifically, we compare the following parallel implementations to execute the Community Detection application:



- The parallelization scheme for the Louvain method [419–421] provided by Grappolo suite [12], henceforth referred to as SimpleCD, in which the vertices are processed as they appear in the input graph representation. The algorithm consists of multiple iterations. First, each vertex is placed in a community of its own. Then, multiple iterations are performed until a convergence criterion is met. Within each iteration, all vertices are processed concurrently by multiple parallel threads, and a greedy decision is made to decide whether each vertex should be moved to a different community (selected from one of its adjacent vertices) or should remain in its current community, targeting to maximize the net modularity gain. For more details, we refer the reader to [419, 422–424].
- The chromatic scheduling parallelization approach using *ColorTM* to color the vertices of the graph, henceforth referred to as *ColorTMCD*, in which the vertices are processed in the order they are distributed in the color classes. The end-to-end Community Detection execution can be broken down in two steps: (i) the time to color the vertices of the graph with *ColorTM*, and (ii) the time to classify the vertices of the graph into communities via chromatic scheduling parallelization approach. The (ii) step processes the color classes produced by the (i) step sequentially, and all vertices of the same color class are processed in parallel.
- The chromatic scheduling parallelization approach using *ColorTM* to color the vertices of the graph and *BalColorTM* to balance the vertices across color classes produced, henceforth referred to as *BalColorTMCD*, in which the vertices are processed in the order they are distributed in the color classes. The end-to-end Community Detection execution can be broken down in three steps: (i) the time to color the vertices of the graph with *ColorTM*, (ii) the time to balance the vertices of the graph across color classes, and (iii) the time to classify the vertices of the graph into communities via chromatic scheduling parallelization approach. The (iii) step processes the color classes produced by the (ii) step sequentially, and all vertices of the same color class are processed in parallel.

Figure 2.25 evaluates the scalability of all the end-to-end Community Detection parallel implementations in a representative subset of large real-world graphs, as the number of parallel threads increases. We present the *total* end-to-end execution time, i.e., in *ColorTMCD* we account for the time to color the vertices of the graph (coloring step), and in *BalColorTMCD* we account for the time to color the vertices of the graph (coloring step), and the time to balance the vertices across color classes (balancing step).

We draw two findings. First, we find that *ColorTMCD* and *BalColorTMCD* scale well in large real-world graphs. For example, when increasing the number of threads from 1 to 56, *ColorTMCD* improves performance by  $12.34\times$  and  $3.44\times$  in *bum* and *arb* graphs, respectively. Similarly, when increasing the number of threads from 1 to 56, *BalColorTMCD* improves performance by  $11.38\times$  and  $3.63\times$  in *bum* and *arb* graphs, respectively. However, we observe that in *uk* and *arb* graphs, SimpleCD outperforms both *ColorTMCD* and *BalColorTMCD*. In these two graphs, *ColorTM* and *BalColorTM* produce the largest number of color classes compared to all the remaining real-world graphs (See Table 2.3), i.e., they produce 944 and 3248 colors for the *uk* and *arb* graphs, respectively. As a result, in *uk* and *arb* graphs the chromatic scheduling parallelization approach of *ColorTMCD*



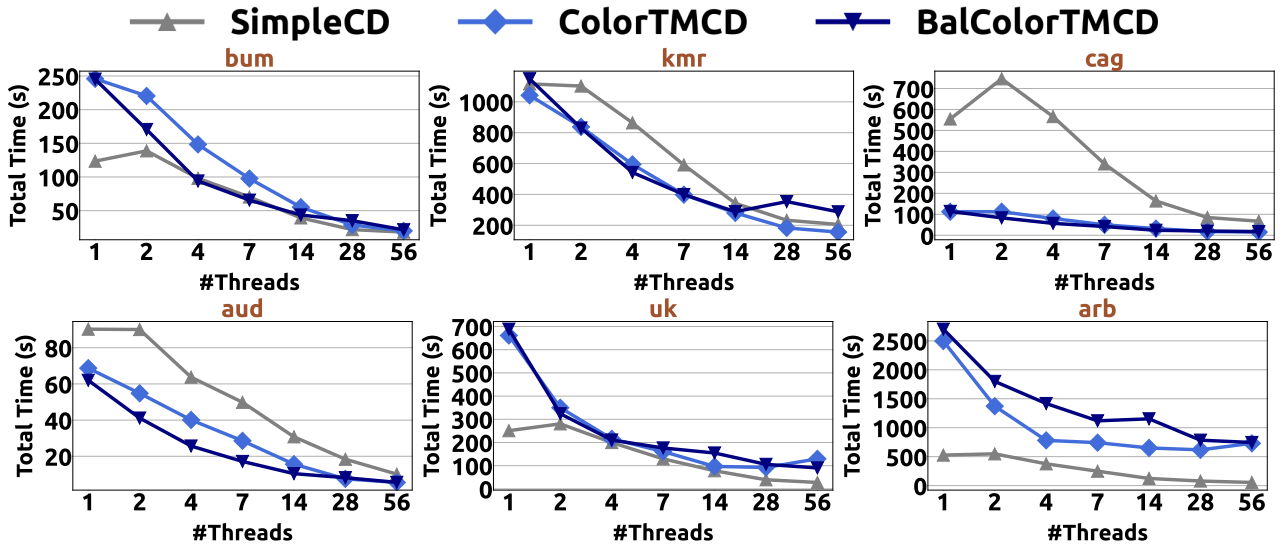


Figure 2.25: Scalability of the end-to-end Community Detection execution achieved by (i) the Grap-polo [12] parallelization approach of the Louvain method (SimpleCD) and (ii) the chromatic scheduling parallelization approach with *ColorTM* (*ColorTMCD*) and (iii) the chromatic scheduling parallelization approach with both *ColorTM* and *BalColorTM* (*BalColorTMCD*) in large real-world graphs.

and *BalColorTMCD* executes 944 and 3248 times of *barrier* synchronization among parallel threads, respectively, thus incurring higher synchronization costs over SimpleCD. Second, the scalability of *BalColorTMCD* is affected more by the NUMA effect compared to that of *ColorTMCD*. Specifically, when increasing the number of threads from 14 to 28, the performance of *ColorTMCD* improves by  $1.63\times$  averaged across all real-world graphs, while the performance of *BalColorTMCD* only improves by  $1.22\times$ . Similarly, when increasing the number of threads from 14 to 56, the performance of *ColorTMCD* improves by  $1.98\times$ , while the performance of *BalColorTMCD* improves by  $1.50\times$ . We find that even though balancing the sizes of color classes provides higher load balance across parallel threads of real-world end-applications, it might be because more remote expensive memory accesses across NUMA sockets of modern multicore machines.

Figure 2.26 shows the actual kernel time (without accounting for performance overheads introduced by the coloring and balancing steps) of Community Detection by comparing the speedup of *ColorTMCD* and *BalColorTMCD* over SimpleCD in all our evaluated large real-world graphs.

We draw two key findings. First, *BalColorTM* can on average outperform *ColorTM*, when considering only the actual kernel time of Community Detection, by providing better load balance among parallel threads. When only the actual kernel time of Community Detection is considered (excluding the performance overheads introduced by the coloring and balancing steps), *BalColorTMCD* on average outperforms *ColorTMCD* by  $1.27\times$ ,  $1.01\times$  and  $1.12\times$  when using 14, 28, and 56 threads, respectively. Second, parallelizing the Community Detection using *ColorTM* and *BalColorTM* provides significant performance speedups over SimpleCD, the state-of-the-art parallelization approach of Louvain method of Community Detection [12, 419–421]. Specifically, *ColorTMCD* improves the performance of the actual kernel time of Community Detection compared to SimpleCD by  $1.40\times$ ,  $1.34\times$ , and  $1.20\times$ , when using 14, 28, and 56 threads, respectively. In addition, *BalColorTMCD* improves the performance of the actual kernel time of Community Detection compared to SimpleCD

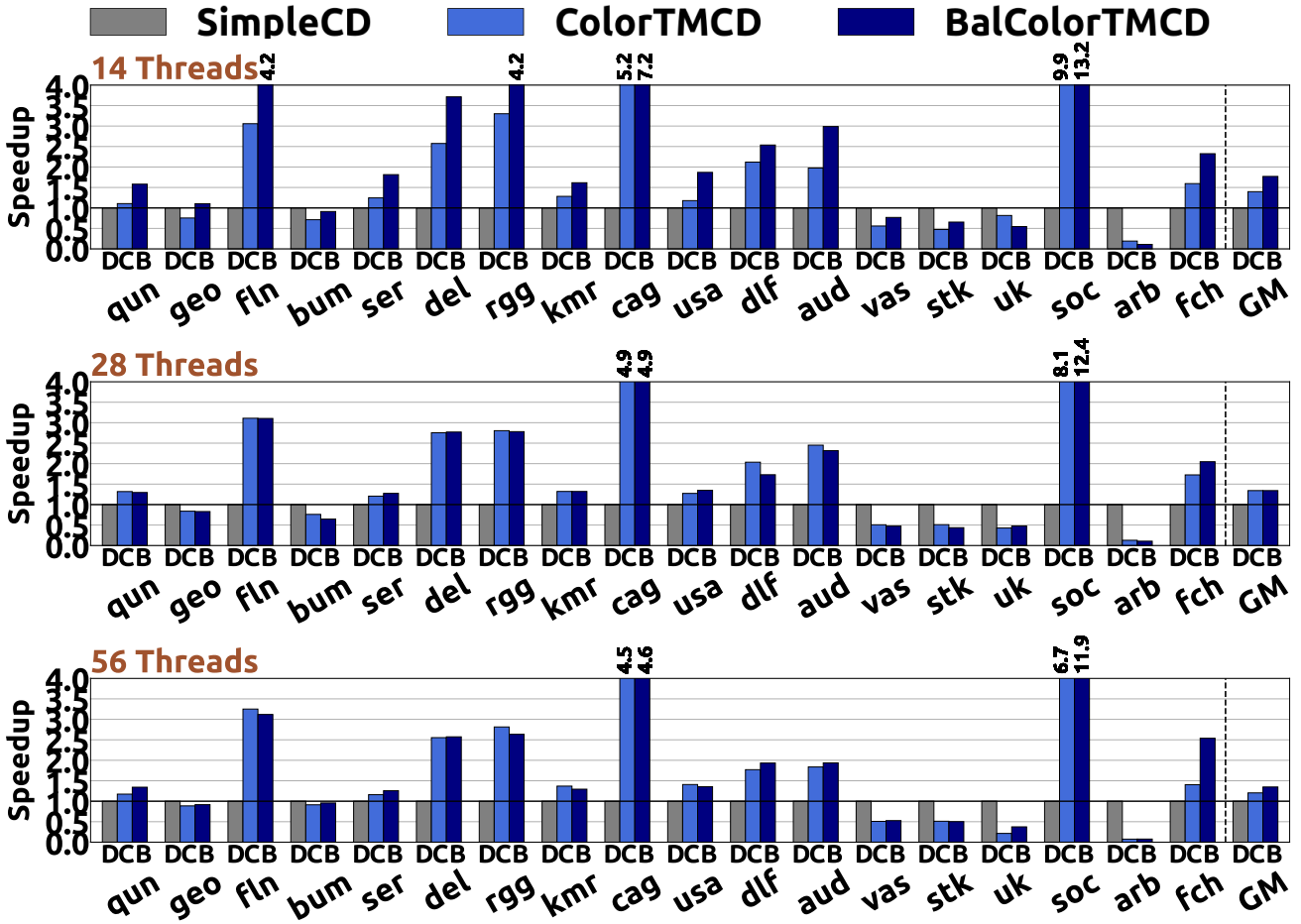


Figure 2.26: Speedup of the actual kernel of the Community Detection execution achieved by (i) SimpleCD (D), (ii) *ColorTMCD* (C) and (iii) *BalColorTMCD* (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

by  $1.77\times$ ,  $1.34\times$ , and  $1.34\times$ , when using 14, 28, and 56 threads, respectively. We conclude that our proposed graph coloring algorithmic designs can provide high performance benefits in real-world end-applications which are parallelized using coloring.

Figure 2.27 presents the speedup breakdown of *ColorTMCD* and *BalColorTMCD* over SimpleCD in all our evaluated large real-world graphs. The performance is broken down in three steps: (i) the coloring step to color the vertices of the graph (**Coloring**), (ii) the balancing step to balance the vertices across color classes (**Balancing**), and (iii) the actual Community Detection kernel time (**CommunityDetection**).

We make two key observations. First, *BalColorTMCD* on average outperforms *ColorTMCD* when using up to 14 threads (using one single NUMA socket). When considering the end-to-end execution including the performance overheads introduced by the coloring and balancing steps, *BalColorTMCD* outperforms *ColorTMCD* by  $1.19\times$  when using 14 threads, while it performs on average  $1.18\times$  and  $1.10\times$  worse over *ColorTMCD*, when using 28 and 56 threads, respectively. We find that the performance overhead introduced in the balancing step of *BalColorTMCD* is not compensated in the runtime of the actual kernel time of Community Detection when using both NUMA sockets of our machine. Second, we observe that both *ColorTMCD* and *BalColorTMCD* can provide high performance in Com-

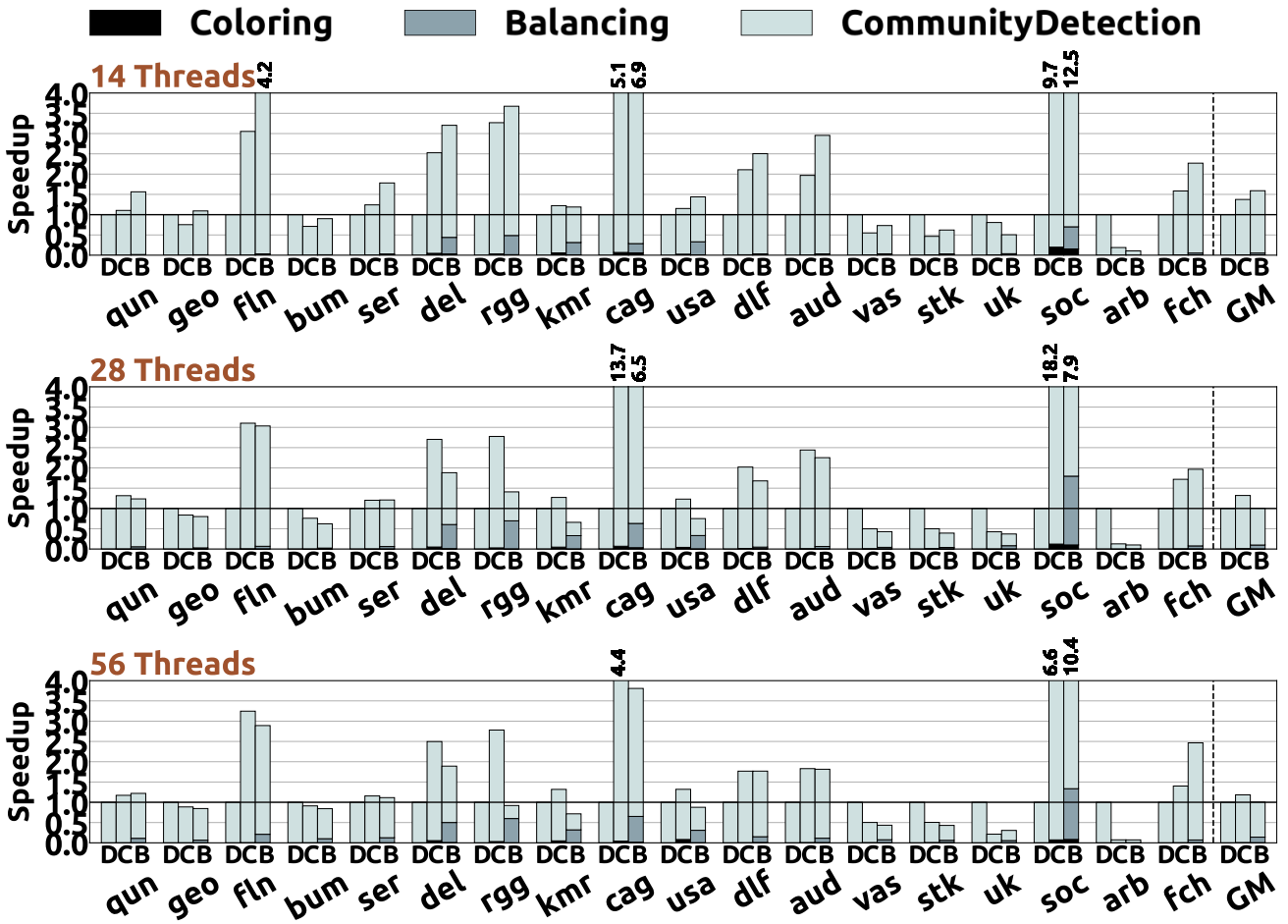


Figure 2.27: Speedup breakdown of the end-to-end Community Detection execution achieved by (i) SimpleCD (D), (ii) *ColorTMCD* (C) and (iii) *BalColorTMCD* (B) in large real-world graphs using all cores of one socket (14 threads), all cores of two sockets (28 threads), and the maximum hardware thread capacity of our machine with hyperthreading enabled (56 threads).

munity Detection. *ColorTMCD* on average outperforms SimpleCD by  $1.38\times$ ,  $1.33\times$  and  $1.19\times$ , when using 14, 28 and 56 threads, respectively. *BalColorTMCD* on average outperforms SimpleCD by  $1.64\times$ ,  $1.10\times$  and  $1.08\times$ , when using 14, 28 and 56 threads, respectively. In addition, we observe that *BalColorTMCD* provides significant performance speedups over Simple CD in many graphs such as fln, del, cag, aud, soc and fch, reaching up to  $10.36\times$  with 56 threads. Overall, we conclude that our proposed parallel graph coloring algorithms can provide significant performance improvements in real-world end-applications, e.g., parallelizing Community Detection with chromatic scheduling, across a wide variety of input data sets with diverse characteristics.

## 2.7 Recommendations

This section presents our key takeaways in the form of recommendations for software and hardware designers.

**Recommendation #1.** *Optimize the Hardware Transactional Memory implementation on NUMA multicore systems.* Figures 2.18 and 2.24 demonstrate the number of transactional aborts significantly increases when using both NUMA sockets of our machine. Accessing data of remote NUMA sockets

within HTM transactions increases the duration of the transactions, thus potentially causing transactional aborts: long-running HTM transactions increase the probability of incurring read-write conflicts among them, while they might suffer from time interrupt aborts when the OS scheduler schedules out the software threads from the hardware threads. Overall, we find that current HTM implementations are severely limited by the NUMA effect [335], which degrades the benefits of HTM on synchronization among parallel threads. To this end, we suggest that hardware designers of multicore systems provide a NUMA-aware HTM implementation for modern multicore systems.

**Recommendation #2.** *Design intelligent data partitioning techniques of real-world graphs across NUMA sockets of modern systems.* Figure 2.19 shows that the number of *conflicts* (read-write) aborts among running HTM transactions significantly increases when using both sockets of our evaluated machine. This is because expensive accesses to remote data increase the duration of the HTM transactions, and thus the probability of causing conflicts aborts among long-running transactions becomes very high. Thus, we conclude that the performance of parallel algorithms might significantly degrade when accessing application data from remote NUMA sockets within the critical section. Therefore, we recommend that software designers of parallel graph processing kernels design effective data partitioning techniques of real-world graphs across NUMA sockets of modern systems to minimize contention and synchronization overheads among parallel threads.

**Recommendation #3.** *Design adaptive parallel applications that on-the-fly adjust the number of parallel threads used to parallelize their sub-kernels based on the parallelization needs of each particular sub-kernel.* Figure 2.15 shows that all parallel graph coloring schemes scale up to 56 threads, i.e., all available hardware threads of our machine. However, Figure 2.21 shows that balanced graph coloring schemes typically scale up 14 threads, thus achieving the best performance with 14 parallel threads, while their performance degrades when using all available hardware threads of our machine (56 threads). The graph coloring kernel has high parallelization needs, since all the vertices of the large real-world graph need to be processed (colored) by parallel threads. Instead, the balance coloring kernel has lower parallelization needs, since typically a small subset of the vertices of the graph need to be processed (re-colored) by parallel threads. We demonstrate in Section 2.6.3 that the execution times of the graph coloring and balance coloring kernels add to the overall execution time of the real-world end-application. Thus, we conclude that to achieve high system performance in the end-to-end execution of real-world applications, we need to dynamically tune the number of parallel threads used to parallelize the sub-kernels of the end-applications depending on the parallelization needs of each particular sub-kernel. To this end, we recommend that software designers provide adaptive parallel applications that on-the-fly adjust the number of parallel threads used to parallelize each sub-kernel of the end-applications based on the parallelization needs of the particular sub-kernel.

## 2.8 Related Work

A handful of prior works [1, 48–52, 52–58] has examined the graph coloring kernel in modern multicore platforms. Welsh and Powell [48] propose the original sequential Greedy algorithm that colors

the vertices of the graph using the *first-fit* heuristic. Recent prior works [53–56] parallelize Greedy by proposing the SeqSolve, IterSolve and IterSolveR schemes described in Section 2.2.2. We compare *ColorTM* with these prior schemes in Section 2.6.1, and demonstrate that our proposed *ColorTM* outperforms these state-of-the-art schemes across a wide variety of real-world graphs. Jones and Plassmann [50] design an algorithm, named JP, that colors the vertices of the graph by identifying independent sets of vertices: in each iteration, the algorithm finds and selects an independent set of vertices that can be colored concurrently. However, JP is a recursive algorithm that typically runs longer than the original Greedy [1, 57, 58], since it performs more computations and needs more synchronization points, i.e., parallel threads need to synchronize at each iteration of processing independent sets of vertices. Moreover, the original paper [50] shows that JP provides good performance mostly in  $\mathcal{O}(1)$ -degree graphs. In contrast, our work efficiently parallelizes the original and widely used Greedy algorithm for graph coloring, and our proposed parallel algorithms achieve significant performance improvements across a wide variety of real-world graphs and using a large number of parallel threads.

Deveci et al. [51] present an edge-centric parallelization scheme for graph coloring which is better suited for GPUs. *ColorTM* and *BalColorTM* can be straightforwardly extended to color the vertices of a graph by equally distributing the edges of the graph among parallel threads. We leave the exploration of edge-centric graph coloring schemes for future work. Future work also comprises the experimentation of the graph coloring kernel on multicore computing platforms such as modern GPUs [425–428] and Processing-In-Memory systems [5, 10, 32, 33, 35, 155, 156, 161, 208, 212]. Maciej et al. [58] and Hasenplaugh et al. [57] propose new vertex *ordering* heuristics for graph coloring. Ordering heuristics define the order in which Greedy colors the vertices of the graph in order to improve the coloring quality by minimizing the number of colors used. Instead, our work aims to improve system performance by proposing efficient parallelization schemes. For a fair comparison, we employ the *first-fit* ordering heuristic (the vertices of the graph are colored in the order they appear in the input graph representation) in all parallel algorithms evaluated in Sections 2.6.1 and 2.6.1. *ColorTM* and *BalColorTM* can support various ordering heuristics [57, 58, 343, 344, 351, 409–413, 429] by assigning the vertices of the graph to parallel threads with a particular order. We leave the evaluation of various vertex ordering heuristics for future work.

Lu et al. [49] design *balanced* graph coloring algorithms to efficiently balance the vertices across the color classes. We compare *BalColorTM* with their proposed algorithms, i.e., CLU, VFF, Recoloring, in Section 2.2.3, and demonstrate that our proposed *BalColorTM* scheme on average performs best across all large real-world graphs. Tas et al. [52] propose balanced graph coloring algorithms for bipartite graphs, i.e., graphs whose vertices can be divided into two disjoint and independent sets  $U$  and  $V$ , and every edge  $(u, v)$  either connects a vertex from  $U$  to  $V$  or a vertex from  $V$  to  $U$ . In contrast, *ColorTM* and *BalColorTM* are designed to be general, and efficiently color any *arbitrary* real-world graph using a large number of parallel threads. In addition, Tas et al. [52] also explore the distance-2 graph coloring kernel on multicore architectures, in which any two vertices  $u$  and  $v$  with an edge-distance at most 2 are assigned with different colors. Instead, our work efficiently parallelizes the distance-1 graph coloring kernel on multicore platforms, in which any two adjacent vertices

of the graph connected with a *direct* edge are assigned with different colors. Finally, prior works propose algorithms for edge coloring [430], dynamic or streaming coloring [431–436], k-distance coloring [437, 438] and sequential exact coloring [439–441]. All these works are not closely related to our work, since we focus on designing high-performance parallel algorithms for the distance-1 vertex graph coloring kernel.

## 2.9 Summary

In this work, we explore the graph coloring kernel on multicore platforms, and propose *ColorTM* and *BalColorTM*, two novel algorithmic designs for high performance and balanced graph coloring on modern computing platforms. *ColorTM* and *BalColorTM* achieve high system performance through two key techniques: (i) *eager* conflict detection and resolution of the coloring inconsistencies that arise when adjacent vertices are concurrently processed by different parallel threads, and (ii) *speculative* computation and synchronization among parallel threads by leveraging Hardware Transactional Memory. Via the eager coloring conflict detection and resolution policy, *ColorTM* and *BalColorTM* effectively leverage the deep memory hierarchy of modern multicore platforms and minimize access costs to application data. Via the speculative computation and synchronization approach, *ColorTM* and *BalColorTM* minimize synchronization costs among parallel threads and provide high amount of parallelism. Our evaluations demonstrate that our proposed parallel graph coloring algorithms outperform prior state-of-the-art approaches across a wide range of large real-world graphs. *ColorTM* and *BalColorTM* can also provide significant performance improvements in real-world scenarios. We conclude that *ColorTM* and *BalColorTM* are highly efficient graph coloring algorithms for modern multicore systems, and hope that this work encourages further studies of the graph coloring kernel in modern computing platforms.

# CHAPTER 3

---

## *SmartPQ*

---

### 3.1 Overview

Concurrent data structures are widely used in the software stack, i.e., kernel, libraries and applications. Prior works [15, 61, 86, 442] discuss the need for efficient and scalable concurrent data structures for commodity Non-Uniform Memory Access (NUMA) architectures. Pointer chasing data structures such as linked lists, skip lists and search trees have inherently low-contention, since their operations need to de-reference a non-constant number of pointers before completing. Recent works [86, 443, 444] have shown that lock-free algorithms [70, 445–449] of such data structures can scale to hundreds of threads. On the other hand, data structures such as queues and stacks typically incur high-contention, when accessed by many threads. In these data structures, concurrent threads compete for the *same* memory locations, incurring excessive traffic and non-uniform memory accesses between nodes of a NUMA system.

In this work, we focus on priority queues, which are widely used in a variety of applications, including task scheduling in real-time and computing systems [363], discrete event simulations [361,

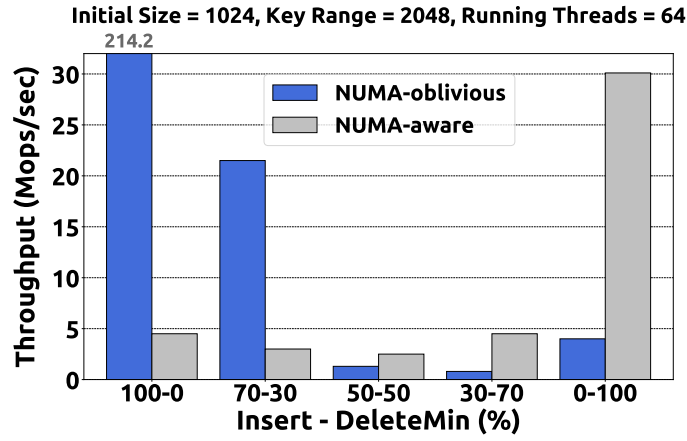


Figure 3.1: Throughput achieved by a NUMA-oblivious [13, 14] and a NUMA-aware [15] priority queue, both initialized with 1024 keys. We use 64 threads that perform a mix of *insert* and *deleteMin* operations in parallel, and the key range is set to 2048 keys. We use *all* NUMA nodes of a 4-node NUMA system, the characteristics of which are presented in Section 3.4.

362] and graph applications [356–358], e.g., Single Source Shortest Path [359] and Minimum Spanning Tree [360]. Similarly to skip-lists and search trees, in *insert* operation, concurrent priority queues typically have high levels of parallelism and low-contention, since threads may work on different parts of the data structure. Therefore, concurrent NUMA-oblivious implementations [59, 60, 62–65, 77, 78] can scale up to a high number of threads. In contrast, in *deleteMin* operation, *all* threads compete for deleting the highest-priority element of the queue, thus competing for the *same* memory locations (similarly to queues and stacks), and creating a contention spot. In *deleteMin*-dominated workloads, concurrent priority queues typically incur high-contention and low parallelism. To achieve higher parallelism, *relaxed* priority queues have been proposed in the literature [13, 364], in which *deleteMin* operation returns an element among the *first few* (high-priority) elements of the priority queue. However, such NUMA-oblivious implementations are still inefficient in NUMA architectures, as we demonstrate in Section 3.4. Therefore, to improve performance in NUMA systems, NUMA-aware implementations have been proposed [15, 86].

We examine NUMA-aware and NUMA-oblivious concurrent priority queues with a wide variety of contention scenarios in NUMA architectures, and find that the performance of a priority queue implementation is becoming increasingly dependent on both the contention levels of the workload and the underlying computing platform. This is illustrated in Figure 3.1, which shows the throughput achieved by a NUMA-oblivious and a NUMA-aware priority queue using a 4-node NUMA system. Even though in a *insert*-dominated scenario, e.g., when having 100% *insert* operations, the NUMA-oblivious implementation achieves significant performance gains over the NUMA-aware one, when contention increases, i.e., the percentage of *deleteMin* operations increases, the NUMA-oblivious implementation incurs non-negligible performance slowdowns over the NUMA-aware priority queue. We conclude that none of the priority queues performs best across *all* contention workloads.

Our **goal** in this work is to design a concurrent priority queue that (i) achieves *the highest performance under all various contention scenarios*, and (ii) performs best even when the contention of the workload *varies* over time.



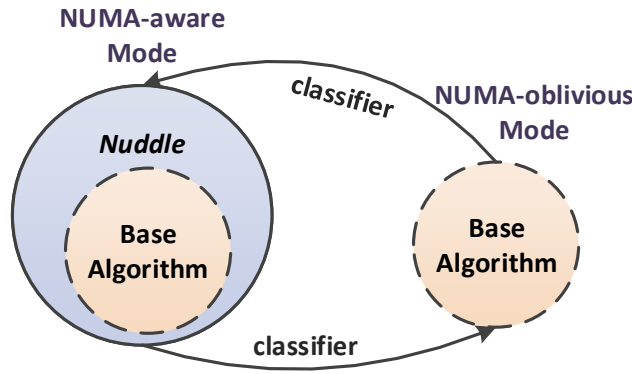


Figure 3.2: High-level overview of *SmartPQ*. *SmartPQ* dynamically adapts its algorithm to the contention levels of the workload based on the prediction of a simple classifier.

To this end, our contribution is twofold. First, we introduce *NUMA Node Delegation (Nuddle)*, a generic technique to obtain NUMA-aware data structures, by effectively transforming *any* concurrent NUMA-oblivious data structure into the corresponding NUMA-aware implementation. In other words, *Nuddle* is a framework to wrap any *concurrent* NUMA-oblivious data structure and transform it into an efficient NUMA-aware one. *Nuddle* extends *ffwd* [15] by enabling multiple server threads, instead of only one, to execute operations in parallel on behalf of client threads. In contrast to *ffwd*, which aims to provide single threaded data structure performance, *Nuddle* targets data structures which are able to scale up to a number of threads such as priority queues.

Second, we propose *SmartPQ*, an adaptive concurrent priority queue that achieves the *highest* performance under *all* contention workloads and *dynamically* adapts itself over time between a NUMA-oblivious and a NUMA-aware algorithmic mode. *SmartPQ* integrates (i) *Nuddle* to *efficiently* switch between the two algorithmic modes with *very low* overhead, and (ii) a simple decision tree *classifier*, which predicts the best-performing algorithmic mode given the expected contention levels of a workload.

Figure 3.2 presents an overview of *SmartPQ*, where we use the term *base algorithm* to denote *any* arbitrary concurrent NUMA-oblivious data structure. *SmartPQ* relies on three key ideas. First, client threads can execute operations using either *Nuddle* (NUMA-aware mode) or its underlying NUMA-oblivious base algorithm (NUMA-oblivious mode). Second, *SmartPQ* incorporates a decision-making mechanism to decide upon transitions between the two modes. Third, *SmartPQ* exploits the fact that the actual underlying implementation of *Nuddle* is a *concurrent* NUMA-oblivious data structure. Client threads in both algorithmic modes access the data structure in the same way, i.e., with no actual change in the way data is accessed. Therefore, *SmartPQ* switches from one mode to another with *no* synchronization points between transitions.

We evaluate a wide range of contention scenarios and compare *Nuddle* and *SmartPQ* with state-of-the-art NUMA-oblivious [13, 77] and NUMA-aware [15] concurrent priority queues. We also evaluate *SmartPQ* using synthetic benchmarks that *dynamically* vary their contention workload over time. Our evaluation shows that *SmartPQ* adapts between its two algorithmic modes with negligible performance overheads, and achieves the highest performance in *all* contention workloads and at *any* point in time with 87.9% success rate.

The main **contributions** of this work are:

- We propose *Nuddle*, a generic technique to obtain NUMA-aware concurrent data structures.
- We design a simple classifier to predict the best-performing implementation among NUMA-oblivious and NUMA-aware priority queues given the contention levels of a workload.
- We propose *SmartPQ*, an adaptive concurrent priority queue that achieves the highest performance, even when contention varies over time.
- We evaluate *Nuddle* and *SmartPQ* with a wide variety of contention scenarios, and demonstrate that *SmartPQ* performs best over prior state-of-the-art concurrent priority queues.

## 3.2 NUMA Node Delegation (*Nuddle*)

### 3.2.1 Overview

NUMA Node Delegation (*Nuddle*) is a generic technique to obtain NUMA-aware data structures by automatically transforming *any* concurrent NUMA-oblivious data structure into an efficient NUMA-aware implementation. *Nuddle* extends *ffwd* [15], a client-server software mechanism which is based on the delegation technique [81–85].

Figure 3.3 left shows the high-level overview of *ffwd*, which has three key design characteristics. First, *all* operations performed by multiple client threads are *delegated* to one *single* dedicated thread, called *server* thread. The server thread performs operations in the data structure on behalf of its client threads. This way, the data structure remains in the memory hierarchy of a *single* NUMA node, avoiding non-uniform memory accesses to remote data. Second, *ffwd* eliminates the need for synchronization, since the shared data structure is no longer accessed by multiple threads: only a single server thread directly modifies the data structure, and therefore, *ffwd* uses a *serial asynchronous* implementation of the underlying data structure. Third, *ffwd* provides an efficient communication protocol between the server thread and client threads that minimizes cache coherence overheads. Specifically, *ffwd* reserves dedicated cache lines to exchange request and response messages between the client threads and server thread. Multiple client threads are grouped together to minimize the response messages from the server thread: one response cache line is shared among multiple client threads belonging to the same client thread group. For more details, we refer the reader to the original paper [15].

Figure 3.3 right presents the high-level overview of *Nuddle*, which is based on three key ideas. First, *Nuddle* deploys *multiple* servers to perform operations on behalf of multiple client threads. Specifically, client threads are grouped in client thread groups, and each server thread serves multiple client thread groups. This way, multiple server threads *concurrently* perform operations on the data structure, achieving high levels of parallelism up to a number of server threads. Second, *Nuddle* locates all server threads to the *same* NUMA node to keep the data structure in the memory hierarchy of one *single* NUMA node, and propose a NUMA-aware approach. Client threads can be located at any NUMA node. Third, since multiple servers can *concurrently* update the *shared* data structure, *Nuddle* uses a *concurrent* NUMA-oblivious implementation (i.e., which includes synchronization primitives

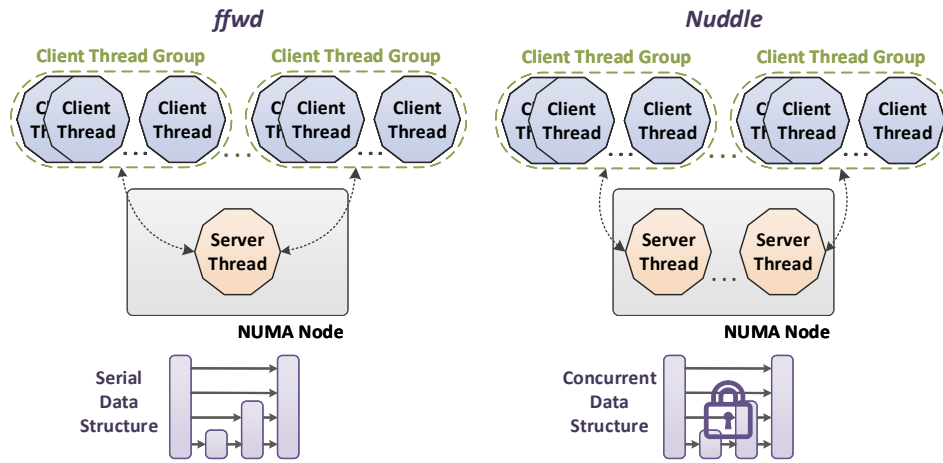


Figure 3.3: High-level design of *ffwd* [15] and *Nuddle*. *Nuddle* locates *all* server threads at the *same* NUMA node to design a NUMA-aware scheme, and associates each of them to multiple client thread groups. *Nuddle* uses the communication protocol proposed in *ffwd* [15].

when accessing the shared data) of the underlying data structure to ensure correctness. Third, *Nuddle* employs the same client-server communication protocol with *ffwd* to carefully manage memory accesses and minimize cache coherence traffic and latency.

*ffwd* targets inherently serial data structures, whose concurrent performance cannot be better than that of *single* threaded performance. In contrast, *Nuddle* targets data structures that can scale up to a number of concurrent threads. Priority queue is a typical example of such a data structure. In *insert* operation, priority queue can scale up to multiple threads, which can *concurrently* update the shared data. In contrast, *deleteMin* operation is inherently serial: at each time only *one* thread can update the shared data, since *all* threads compete for the highest-priority element of the queue. However, as we mentioned, in relaxed priority queues (e.g., SprayList [13]), even *deleteMin* operation can be parallelized to some extent.

### 3.2.2 Implementation Details

Figures 3.4, 3.5 and 3.6 present the code of a priority queue implementation using *Nuddle*. We denote with red color the core operations of the *base algorithm*, which is used as the underlying concurrent NUMA-oblivious implementation of *Nuddle*. Note that even though in this work we focus on priority queues, *Nuddle* is a *generic* framework for any type of concurrent data structure.

**Helper Structures.** *Nuddle* includes three *helper* structures (Figure 3.4), which are needed for client-server communication. First, the main structure of *Nuddle*, called *struct nuddle\_pq*, wraps the *base algorithm* (*nm\_oblv\_set*), and includes a few additional fields, which are used to associate client thread groups to server threads in the initialization step. Second, each client thread has its own *struct client* structure with a dedicated request and a dedicated response cache line. The request cache line is exclusively written by the client thread and read by the associated server thread, while the response cache line is exclusively written by the server thread and read by all client threads that belong to the same client thread group. Third, each server thread has its own *struct server* structure that includes an array of requests (*my\_clients*), each of them is shared with a client thread,

```

43 #define cache_line_size 128
44 typedef char cache_line[cache_line_size];
45
46 struct nuddle_pq {
47     nm_oblv_set *base_pq;
48     int servers, groups, clnt_per_group;
49     int server_cnt, clients_cnt, group_cnt;
50     cache_line *requests[groups][clnt_per_group];
51     cache_line *responses[groups];
52     lock *global_lock;
53 };
54
55 struct client {
56     cache_line *request, *response;
57     int clnt_pos;
58 };
59
60 struct server {
61     nm_oblv_set *base_pq;
62     cache_line *my_clients[], *my_responses[];
63     int my_groups, clnt_per_group;
64 };

```

Figure 3.4: Helper structures of *Nuddle*.

and an array of responses (*my\_responses*), each of them is shared with all client threads of the *same* client thread group.

**Initialization Step.** Figure 3.5 describes the initialization functions of *Nuddle*. *initPQ()* initializes (i) the underlying data structure using the corresponding function of the *base algorithm* (line 25), and (ii) the additional fields of *struct nuddle\_pq*. For this function, programmers need to specify the number of server threads and the maximum number of client threads to properly allocate cache lines needed for communication among them. Programmers also specify the size of the client thread group (line 27), which is typically 7 or 15, if the cache line is 64 or 128 bytes, respectively. As explained in *ffwd* [15], assuming 8-byte return values, a dedicated 64-byte (or 128-byte) response cache line can be shared between up to 7 (or 15) client threads, because it also has to include one additional toggle bit for each client thread. After initializing *struct nuddle\_pq*, each running thread calls either *initClient()* or *initServer()* depending on its role. Each thread initializes its own helper structure (*struct client* or *struct server*) with request and response cache lines of the corresponding shared arrays of *struct nuddle\_pq*. Server threads undertake client thread groups with a round-robin fashion, such that the load associated with client threads is balanced among them. In function *initServer()*, it is the programmer’s responsibility to properly pin software server threads to hardware contexts (line 56), such that server threads are located in the *same* NUMA node, and the programmer fully benefits from the *Nuddle* technique. Moreover, given that client threads of the *same* client thread group share the same response cache line, the programmer could pin client threads of the *same* client thread group to hardware contexts of the *same* NUMA node to minimize cache coherence overheads. Finally, since the request and response arrays of *struct nuddle\_pq* are *shared* between all threads, a global lock is used when updating them to ensure mutual exclusion.

**Main API.** Figure 3.6 shows the core functions of *Nuddle*, where we omit the corresponding

```

65 struct nuddle_pq *initPQ(int servers, int max_clients) {
66     struct nuddle_pq *pq = allocate_nuddle_pq();
67     __base_init(pq->base_pq);
68     pq->servers = servers;
69     pq->clnt_per_group = client_group(cache_line_size);
70     pq->groups = (max_clients +
71         pq->clnt_per_group - 1) / pq->clnt_per_group;
72     pq->server_cnt = 0;
73     pq->client_cnt = 0;
74     pq->group_cnt = 0;
75     pq->requests = malloc(groups * clnt_per_group);
76     pq->responses = malloc(groups);
77     init_lock(pq->global_lock);
78     return pq;
79 }
80
81 struct client *initClient(struct nuddle_pq *pq) {
82     struct client *cl = allocate_client();
83     acquire_lock(pq->global_lock);
84     cl->request = &(pq->requests[group_cnt][clients_cnt]);
85     cl->response = &(pq->responses[group_cnt]);
86     cl->pos = pq->client_cnt;
87     pq->client_cnt++;
88     if (pq->client_cnt % pq->clnt_per_group == 0) {
89         pq->clients_cnt = 0;
90         pq->group_cnt++;
91     }
92     release_lock(pq->global_lock);
93     return cl;
94 }
95
96 struct server *initServer(struct nuddle_pq *pq, int core)
97 {
98     set_affinity(core);
99     struct server *srv = allocate_server();
100    srv->base_pq = pq->base_pq;
101    srv->my_groups = 0;
102    srv->clnt_per_group = pq->clnt_per_group;
103    acquire_lock(pq->global_lock);
104    int j = 0;
105    for(i = 0; i < pq->groups; i++)
106        if(i % pq->servers == pq->server_cnt) {
107            srv->my_clients[j] = pq->requests[i][0..gr_clnt];
108            srv->my_responses[j++] = pq->responses[i];
109            srv->my_groups++;
110        }
111    pq->server_cnt++;
112    release_lock(pq->global_lock);
113    return srv;
114 }

```

Figure 3.5: Initialization functions of *Nuddle*.

functions for *deleteMin* operation, since they are very similar to that of *insert* operation. Both *insert* and *deleteMin* operations of *Nuddle* have similar API with the classic API of prior state-of-the-art priority queue implementations [13, 59, 77, 78]. However, we separate the corresponding functions for client threads and server threads. A client thread writes its request to a dedicated request cache

line (line 75) and then waits for the server thread's response. In contrast, a server thread directly executes operations in the data structure using the core functions of the *base algorithm* (line 82). Moreover, a server thread can serve client threads using the *serve\_requests()* function. A server thread iterates over its own client thread groups and executes the requested operations in the data structure. The server thread buffers individual return values for clients to a local cache line (*resp* in lines 92 and 94) until it finishes processing all requests for the current client thread group. Then, it writes all responses to the *shared* response cache line of that client thread group (line 96), and proceeds to its next client thread group.

```

115 int insert_client(struct client *cl, int key, int64_t value)
116 {
117     cl->request = write_req("insert", key, value);
118     while (cl->response[cl->pos] == 0) ;
119     return cl->response[cl->pos];
120 }
121
122 int insert_server(struct server *srv, int key, int64_t value)
123 {
124     return __base_insrt(srv->base_pq, key, value);
125 }
126
127 void serve_requests(struct server *srv) {
128     for(i = 0; i < srv->mygroups; i++) {
129         cache_line resp;
130         for(j = 0; j < srv->clnt_per_group; j++) {
131             key = srv->my_clients[i][j].key;
132             value = srv->my_clients[i][j].value;
133             if (srv->my_clients[i][j].op == "insert")
134                 resp[j] = __base_insrt(srv->base_pq, key, value);
135             else if (srv->my_clients[i][j].op == "deleteMin")
136                 resp[j] = __base_delMin(srv->base_pq);
137         }
138         srv->my_responses[i] = resp;
139     }
140 }

```

Figure 3.6: Functions used by server threads and client threads to perform operations using *Nuddle*.

### 3.3 SmartPQ

We propose *SmartPQ*, an adaptive concurrent priority queue which tunes itself by *dynamically* switching between NUMA-oblivious and NUMA-aware algorithmic modes, in order to perform best in *all* contention workloads and at *any* point in time, even when contention varies over time.

Designing an adaptive priority queue involves addressing two major challenges: (i) how to switch from one algorithmic mode to the other with *low overhead*, and (ii) *when* to switch from one algorithmic mode to the other.

To address the first challenge, we exploit the fact that the actual underlying implementation of *Nuddle* is a *concurrent* NUMA-oblivious implementation. We select *Nuddle*, as the NUMA-aware algorithmic mode of *SmartPQ*, and its underlying *base algorithm*, as the NUMA-oblivious algorithmic

mode of *SmartPQ*. Threads can perform operations in the data structure using either *Nuddle* or its underlying *base algorithm*, with *no actual change* in the way data is accessed. As a result, *SmartPQ* can switch between the two algorithmic modes *without* needing a synchronization point between transitions, and without violating correctness.

To address the second challenge, we design a simple decision tree classifier (Section 3.3.1), and train it to select the best-performing algorithmic mode between *Nuddle*, as the NUMA-aware algorithmic mode of *SmartPQ*, and its underlying *base algorithm*, as the NUMA-oblivious mode of *SmartPQ*. Finally, we add a *lightweight* decision-making mechanism in *SmartPQ* (Section 3.3.2) to dynamically tune itself over time between the two algorithmic modes. We describe more details in next sections.

### 3.3.1 Selecting the Algorithmic Mode

#### The Need for a Machine Learning Approach

Selecting the best-performing algorithmic mode can be solved in various ways. For instance, one could take an empirical exhaustive approach: measure the throughput achieved by the two algorithmic modes for all various contention scenarios on the target NUMA system, and then use the algorithmic mode that achieves the highest throughput on future runs of the same contention workload on the target NUMA system. Even though this is the most accurate method, it (i) incurs *substantial* overhead and effort to sweep over *all* various contention workloads, and (ii) would need a large amount of memory to store the best-performing algorithmic mode for all various scenarios. Furthermore, it is not trivial to construct a statistical model to predict the best-performing algorithmic mode, since the performance of an algorithm is also affected by the characteristics of the underlying computing platform. Figure 3.7 summarizes these observations by comparing *Nuddle* with its underlying *base algorithm* in a 4-node NUMA system. For the *base algorithm*, we use *alistarh\_herlihy* priority queue [13, 14], since this is the NUMA-oblivious implementation that achieves the highest performance, according to our evaluation (Section 3.4).

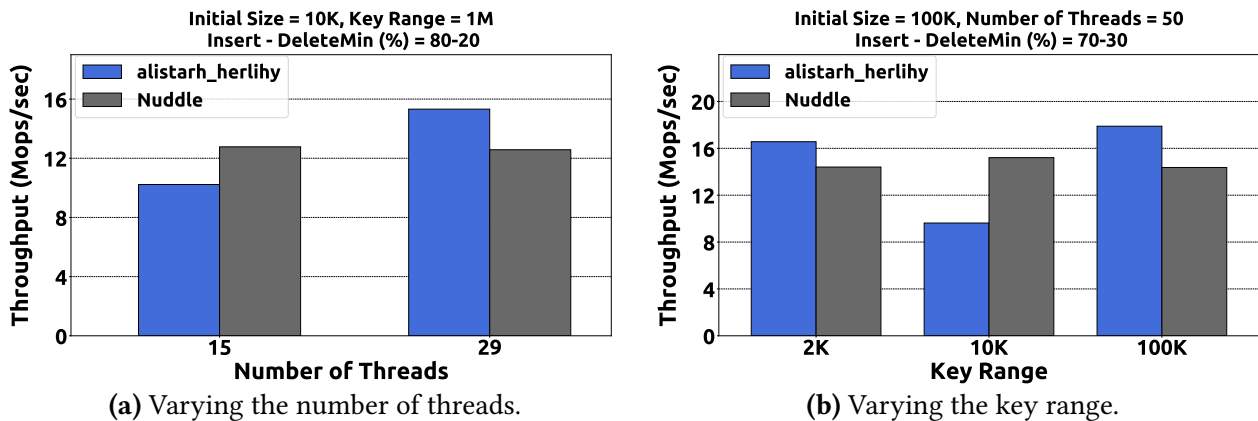


Figure 3.7: Throughput achieved by *Nuddle* (using 8 server threads) and its underlying NUMA-oblivious *base algorithm*, i.e., *alistarh\_herlihy* [13, 14], when we vary (a) the number of threads that perform operations in the shared data structure, and (b) the key range of the workload.

Figure 3.7a demonstrates that the best-performing algorithmic mode depends on multiple pa-



rameters, such as the number of threads that perform operations in the shared data structure, the size of the data structure, the operation workload, i.e., the percentage of *insert/deleteMin* operations. Specifically, when the number of threads increases, we may expect that the performance of the NUMA-oblivious *alistarh\_herlihy* degrades due to higher contention. In contrast, with 80% *insert* operations when increasing the number of threads to 29, *alistarh\_herlihy* outperforms *Nuddle*. This is because the size of the priority queue and the range of keys used in the workload are relatively large, while the percentage of *deleteMin* operations is low. In this scenario, threads may not compete for the same elements, working on different parts of the data structure, and thus, the NUMA-oblivious *alistarh\_herlihy* achieves higher throughput compared to the NUMA-aware *Nuddle*.

Figure 3.7b demonstrates that the best-performing algorithmic mode cannot be straightforwardly predicted, and also depends on the characteristics of underlying hardware [4]. In *insert*-dominated workloads, as the key range increases, threads may update different parts of the shared data structure. We might, thus, expect that after a certain point of increasing the key range, the NUMA-oblivious *alistarh\_herlihy* will always outperform *Nuddle*, since the contention decreases. However, we note that, even though the performance of *Nuddle* remains constant, as expected, the performance of *alistarh\_herlihy* highly varies as the key range increases due to the hyperthreading effect. When using more than 32 threads, hyperthreading is enabled in our NUMA system (Section 3.4). The hyperthreading pair of threads shares the L1 and L2 caches, and thus, these threads may either thrash or benefit from each other depending on the characteristics of L1 and L2 caches (e.g., size, eviction policy), and the elements accessed in each operation.

Considering the aforementioned non-straightforward behavior, we resort to a machine learning approach as the basis of our prediction mechanism.

## Decision Tree Classifier

We formulate the selection of the algorithmic mode as a classification problem, and leverage supervised learning techniques to train a simple classifier to predict the best-performing algorithmic mode for each contention workload. For our classifier, we select decision trees, since they are commonly used in classification models for multithreaded workloads [88, 96–99, 450–452], and incur low training and inference overhead. Moreover, they are easy to interpret and thus, be incorporated to our proposed priority queue (Section 3.3.2). We generate the decision tree classifier using the scikit-learn machine learning toolkit [453].

**1) Class Definition:** We define the following classes: (a) the **NUMA-oblivious** class that stands for the NUMA-oblivious algorithmic mode, (b) the **NUMA-aware** class that stands for the NUMA-aware algorithmic mode, and (c) the **neutral** class that stands for a tie, meaning that either a NUMA-aware or a NUMA-oblivious implementation can be selected, since they achieve similar performance. We include a neutral class for two reasons: (i) when using *only one* socket of a NUMA system, NUMA-aware implementations deliver similar throughput with NUMA-oblivious implementations, and (ii) in an adaptive data structure, which *dynamically* switches between the two algorithmic modes, we want to configure a transition from one algorithmic mode to another to occur when the difference



Feature	Definition
#Threads	The number of active threads that perform operations in the data structure
Size	The current size of the priority queue
Key_range	The range of keys used in the workload
% <i>insert/deleteMin</i>	The percentage of <i>insert/deleteMin</i> operations

Table 3.1: The features of the contention workload which are used for classification.

in their throughput is relatively high, i.e., greater than a certain threshold. Otherwise, the adaptive data structure might continuously oscillate between the two modes, without delivering significant performance improvements or even causing performance degradation.

**2) *Extracted Features*:** Table 3.1 explains the four features of the contention workload which are used in our classifier targeting priority queues. We assume that the contention workload is known a priori, and thus, we can easily extract the features needed for classification. Section 3.5 discusses how to on-the-fly extract these features.

**3) *Generation of Training Data*:** To train our classifier, we develop microbenchmarks, in which threads repeatedly execute random operations on the priority queue for 5 seconds. We select *Nuddle*, as the NUMA-aware implementation, and *alistarh.herlihy*, as its underlying NUMA-oblivious implementation, since this is the best-performing NUMA-oblivious priority queue (Section 3.4). We use a variety of values for the features needed for classification (Table 3.1). Our training data set consists of 5525 different contention workloads. Finally, we pin software threads to hardware contexts of the evaluated NUMA system in a round-robin fashion, and thus, the classifier is trained with this thread placement. We leave the exploration of the thread placement policy for future work.

**4) *Labeling of Training Data*:** Regarding the labeling of our training data set, we set the threshold for tie between the two algorithmic modes to an empirical value of 1.5 Million operations per second. When the difference in throughput between the two algorithmic modes is less than this threshold, the *neutral* class is selected as label. Otherwise, we select the class that corresponds to the algorithmic mode that achieves the highest throughput.

The final decision tree classifier has only 180 nodes, half of which are leaves. It has a very low depth of 8, that is the length of the longest path in the tree, and thus, a *very low* traversal cost (2-4 ms in our evaluated NUMA system).

### 3.3.2 Implementation Details

Figure 3.8 presents the modified code of *Nuddle* adding the decision-making mechanism (using green color) to implement *SmartPQ*. We extend the main structure of *Nuddle*, renamed to *struct smartpq*, by adding an additional field, called *algo*, to keep track the current algorithmic mode, (either NUMA-oblivious or NUMA-aware). Similarly, *struct client* and *struct server* structures are extended with an additional *algo* field (e.g., line 111), which is a pointer to the *algo* field of *struct smartpq*. Each active thread initializes this pointer either in *initClient()* or

```

141 struct smartpq {
142     nm_oblv_set *base_pq;
143     int servers, groups, clnt_per_group;
144     int server_cnt, clients_cnt, group_cnt;
145     cache_line *requests[groups][clnt_per_group];
146     cache_line *responses[groups];
147     lock *global_lock;
148     int *algo; // 1: NUMA-oblivious (default), 2: NUMA-aware
149 };
150
151 struct client {
152     nm_oblv_set *base_pq;
153     int *algo;
154     cache_line *request, *response;
155     int clnt_pos;
156 };
157
158 struct client *initClient(struct smartpq *pq) {
159     ... lines 40-49 of Fig. 5 ...
160     cl->base_pq = pq->base_pq;
161     cl->algo = &(pq->algo);
162     release_lock(pq->global_lock);
163     return cl;
164 }
165
166 int insert_client(struct client *cl, int key, float value) {
167     if(*(cl->algo) == 1) {
168         return __base_insert(cl->base_pq, key, value);
169     } else { // *(cl->algo) == 2
170         ... lines 75-77 of Fig. 6 ...
171     }
172 }
173
174 void serve_requests(struct server *srv) {
175     if(*(srv->algo) == 2){
176         for(i = 0; i < srv->mygroups; i++) {
177             cache_line resp;
178             for(j = 0; j < srv->clnt_per_group; j++) {
179                 key = srv->my_clients[i][j].key;
180                 value = srv->my_clients[i][j].value;
181                 if (srv->my_clients[i][j].op == "insert")
182                     resp[j] = __base_insrt(srv->base_pq, key, value);
183                 else if (srv->my_clients[i][j].op == "deleteMin")
184                     resp[j] = __base_delMin(srv->base_pq);
185             }
186             srv->my_responses[i] = resp;
187         }
188     } else
189         return;
190 }
191
192 void decisionTree(struct server struct client *str, int nthreads,
193                  int size, int key-range, double insert\_deleteMin) {
194     int algo = 0;
195     ... code for decision tree classifier ...
196     if (algo != 0) // 0: neutral
197         *(str->algo) = algo;
198 }

```

Figure 3.8: The modified code of *Nuddle* with the decision-making mechanism to implement *SmartPQ*.

*initServer()* depending on its role (e.g., line 119). This way, all threads share the same algorithmic mode at *any* point in time. In *struct client*, we also add a pointer to the shared data structure (line 110), which is used by client threads to *directly* perform operations in the data structure in case of NUMA-oblivious mode. Specifically, we modify the core functions of client threads, i.e., *insert\_client()* and *deleteMin\_client()*, such that client threads either directly execute their operations in the data structure (e.g., line 126), or delegate them to server threads (e.g., line 127-128), with respect to the current algorithmic mode. In contrast, the core functions of server threads do not need any modification. Finally, we wrap the code of *serve\_requests* function, i.e., the lines 86-97 of Figure 3.6, with an if/else statement on the *algo* field (lines 133, 146 in Fig. 3.8), such that server threads poll at client threads' requests only in NUMA-aware mode. In NUMA-oblivious mode, *serve\_requests* function returns without doing nothing. This way, programmers do not need to take care of calls on this function in their code, when the NUMA-oblivious mode is selected.

The *decisionTree()* function describes the interface with our proposed decision tree classifier, where the input arguments are associated with its features. In frequent time lapses, one or more threads may call this function to check if a transition to another algorithmic mode is needed. If this is the case, the *algo* field of *struct smartpq* is updated (line 154 in Fig. 3.8), and *SmartPQ* switches algorithmic mode, i.e., all active threads start executing their operations using the new algorithmic mode. If the classifier predicts the neutral class (line 153), the *algo* field is not updated, and thus *SmartPQ* remains at the currently selected algorithmic mode.

### 3.4 Experimental Evaluation

In our experimental evaluation, we use a 4-socket Intel Sandy Bridge-EP server equipped with 8-core Intel Xeon CPU E5-4620 processors providing a total of 32 physical cores and 64 hardware contexts. The processor runs at 2.2GHz and each physical core has its own L1 and L2 cache of sizes 64KB and 256KB, respectively. A 16MB L3 cache is shared by all cores in a NUMA socket and the RAM is 256GB. We use GCC 4.9.2 with -O3 optimization flag enabled to compile all implementations.

Our evaluation includes the following concurrent priority queue implementations:

- *alistarh\_fraser* [13, 70]: A NUMA-oblivious, relaxed priority queue [13] based on Fraser's skip-list [70] available at ASCYLIB library [443].
- *alistarh\_herlihy* [13, 14]: A NUMA-oblivious, relaxed priority queue [13] based on Herlihy's skip-list [14] available at ASCYLIB library [443].
- *lotan\_shavit* [77]: A NUMA-oblivious priority queue available at ASCYLIB library [443].
- *ffwd* [15]: A NUMA-aware priority queue based on the delegation technique [81–85], which includes *only* one server thread to perform operations on behalf of *all* client threads.
- *Nuddle*: Our proposed NUMA-aware priority queue, which uses *alistarh\_herlihy* as the underlying *base algorithm*.
- *SmartPQ*: Our proposed adaptive priority queue, which uses *Nuddle* as the NUMA-aware mode, and *alistarh\_herlihy* as the NUMA-oblivious *base algorithm*.

We evaluate the concurrent priority queue implementations in the following way:

- Each execution lasts 5 seconds, during which each thread performs randomly chosen operations. We also tried longer durations and got similar results.
- Between consecutive operations in the data structure each thread executes a delay loop of 25 pause instructions. This delay is intentionally added in our benchmarks to better simulate a real-life application, where operations in the data structure are intermingled with other instructions in the application.
- At the beginning of each run, the priority queue is initialized with elements the number of which is described at each figure.
- Each software thread is pinned to a hardware context. Hyperthreading is enabled when using more than 32 software threads. When exceeding the number of available hardware contexts of the system, we oversubscribe software threads to hardware contexts.
- We pin the first 8 threads to the first NUMA node, and consecutive client thread groups of 7 client threads each, to NUMA nodes in a round-robin fashion.
- In NUMA-oblivious implementations, any allocation needed in the operation is executed on demand, and memory affinity is determined by the first touch policy.
- In NUMA-aware implementations, since our NUMA system has 64-byte cache lines, the response cache line is shared between up to 7 client threads, using 8-byte return values.
- In *Nuddle*, the first 8 threads represent server threads. Server threads repeatedly execute the *serve\_requests* function, and then a randomly chosen operation until time is up.
- We have disabled the automatic Linux Balancing [454] to get consistent and stable results.
- All reported results are the average of 10 independent executions with no significant variance.

### 3.4.1 Throughput of *Nuddle*

Figure 3.9 presents the throughput achieved by concurrent priority queue implementations for various sizes and operation workloads. NUMA-aware priority queue implementations, i.e., *ffwd* and *Nuddle*, achieve high throughput in *deleteMin*-dominated workloads: *Nuddle* performs best in *all deleteMin*-dominated workloads, while *ffwd* outperforms NUMA-oblivious implementations in the small-sized priority queues (e.g., 100K elements). In large-sized priority queues, *insert* operations have a larger impact on the total execution time (due to a longer traversal), and thus *Nuddle* and NUMA-oblivious implementations perform better than *ffwd*, since they provide *higher* thread-level parallelism. Note that *ffwd* has *single-threaded* performance, since at any point in time only *one* (server) thread performs operations in the data structure. Moreover, as it is expected, the performance of both *ffwd* and *Nuddle* saturates at the number of server threads used (e.g., 8 server threads for *Nuddle*) to perform operations in the data structure. Finally, we note that the communication between server and client threads used in NUMA-aware implementations has negligible overhead; when the number of client threads increases, even though the communication traffic over the interconnect increases, there is *no* performance drop. Overall, we conclude that *Nuddle* achieves the highest throughput in *all deleteMin*-dominated workloads, and is the most efficient NUMA-aware approach, since it provides high thread-level parallelism.

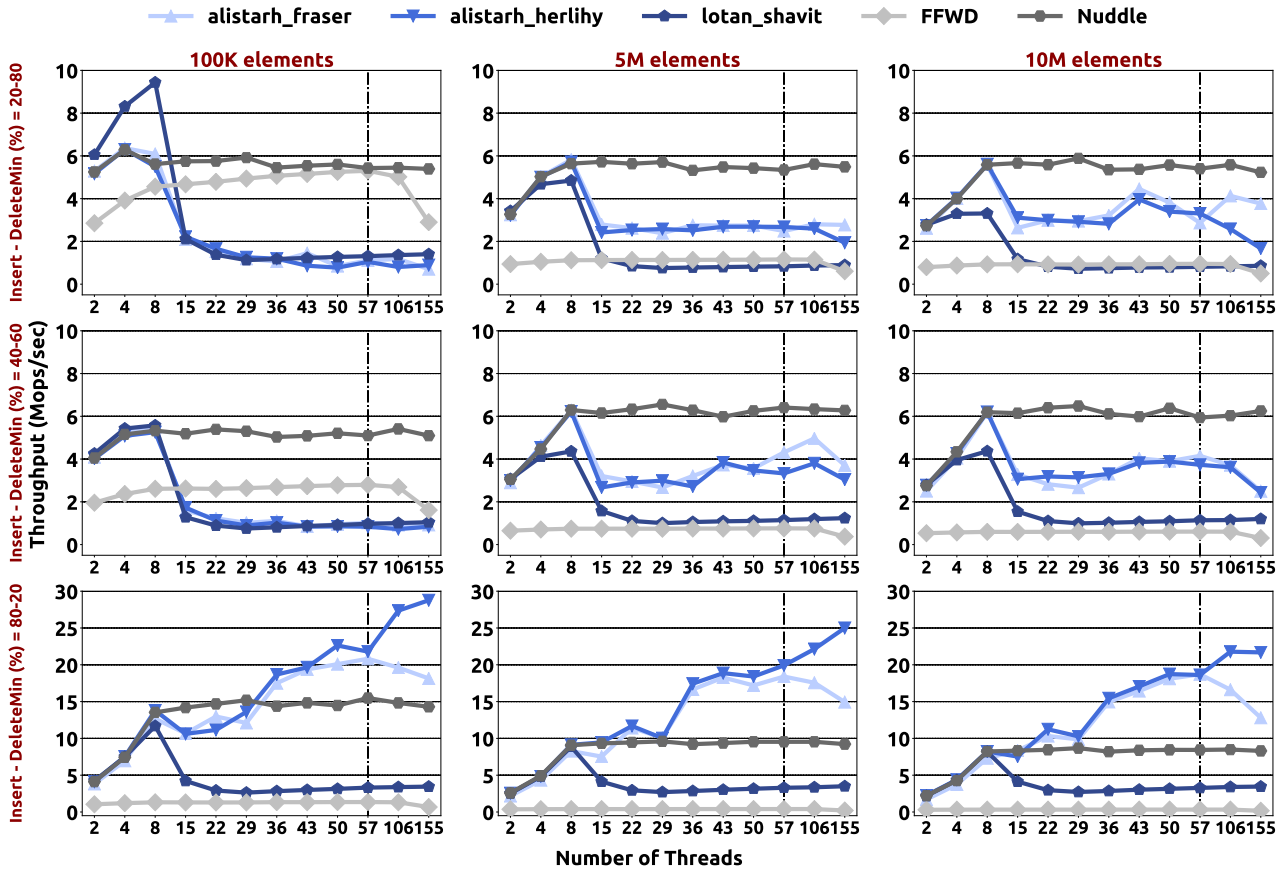


Figure 3.9: Throughput of concurrent priority queue implementations. The columns show different priority queue sizes using the key range of double the elements of each size. The rows show different operation workloads. The vertical line in each plot shows the point after which we oversubscribe software threads to hardware contexts.

On the other hand, NUMA-oblivious implementations incur high performance degradation in *high-contention* scenarios, such as *deleteMin*-dominated workloads, when using more than one NUMA node (i.e., after 8 threads). As already discussed in prior works [5, 16, 83, 455–457], the non-uniformity in memory accesses and cache line invalidation traffic significantly affects performance in high-contention scenarios. In *insert*-dominated workloads, which incur lower contention, even though *lotan\_shavit* priority queue still incurs performance degradation when using more than one NUMA nodes of the system, the *relaxed* NUMA-oblivious implementations, i.e., *alistarh\_fraser* and *alistarh\_herlihy* priority queues, achieve high scalability. This is because relaxed priority queues decrease both (i) the contention among threads, and (ii) the cache line invalidation traffic: the *deleteMin* operation returns (with a high probability) an element among the *first few* (high-priority) elements of the queue, and thus, threads do not frequently compete for the same elements. Finally, we observe that *alistarh\_herlihy* priority queue achieves higher performance benefits over *alistarh\_fraser* priority queue, when we oversubscribe software threads to the available hardware contexts of our system. Overall, we find that in *insert*-dominated workloads, the *relaxed* NUMA-oblivious implementations significantly outperform the NUMA-aware ones.

To sum up, we conclude that there is no one-size-fits-all solution, since none of the priority queues performs best across *all* contention workloads. *Nuddle* achieves the highest throughput in high con-

tention scenarios, while *alistarh\_herlihy* performs best in low and medium contention scenarios. It is thus desirable to design a new approach for a concurrent priority queue to perform best under *all* various contention scenarios.

### 3.4.2 Throughput of *SmartPQ*

#### Classifier Accuracy

We evaluate the efficiency of our proposed classifier (Section 3.3.1) using two metrics: (i) accuracy, and (ii) misprediction cost. First, we define the accuracy of the classifier as the percentage of *correct* predictions, where a prediction is considered correct, if the classifier predicts the algorithmic mode (either the NUMA-aware *Nuddle* or the NUMA-oblivious *alistarh\_herlihy*) that achieves the best performance between the two. We use a test set of 10780 different contention workloads, where we randomly select the values of the features in each workload. In the above test set, our classifier has 87.9% accuracy, i.e., it mispredicts 1300 times in 10780 different contention workloads. Second, we define the misprediction cost as the performance difference between the correct (best-performing) algorithmic mode and the wrong predicted mode, normalized to the performance of the wrong predicted mode. Specifically, assuming the throughput of the wrong predicted and correct (best-performing) algorithmic mode is  $Y$  and  $X$  respectively, the misprediction cost is defined as  $((X - Y)/Y) * 100\%$ . In 1300 mispredicted workloads, the geometric mean of misprediction cost for our classifier is 30.2%. We conclude that the proposed classifier has *high* accuracy, and in case of misprediction, incurs low performance degradation.

#### Varying the Contention Workload

We present the performance benefit of *SmartPQ* in synthetic benchmarks, in which we vary the contention workload over time, and compare it with *Nuddle* and its underlying *base algorithm*, i.e., *alistarh\_herlihy* priority queue. In all benchmarks, we change the contention workload every 25 seconds. In *SmartPQ*, we set one dedicated server thread to call the decision tree classifier *every* second, in order to check if a transition to another algorithmic mode is needed. Figure 3.10 and Figure 3.11 show the throughput achieved by all three schemes, when we vary one and multiple features in the contention workload, respectively. Table 3.2 and Table 3.3 show the features of the workload as they vary during the execution for the benchmarks evaluated in Figure 3.10 and Figure 3.11, respectively. Note that the current size of the priority queue changes during the execution due to successful *insert* and *deleteMin* operations.

We make three observations. First, as already shown in Section 3.4.1, there is no one-size-fits-all solution, since neither *Nuddle* nor *alistarh\_herlihy* performs best across all various contention workloads. For instance, in Figure 3.10b, even though the performance of *Nuddle* remains constant, it outperforms *alistarh\_herlihy*, when having 15 running threads, i.e., using 2 NUMA nodes of the system. Second, we observe that *SmartPQ* successfully adapts to the best-performing algorithmic mode, and performs best in *all* contention scenarios. In Figure 3.11, even when multiple features in the

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	<b>1149</b>	<b>100K</b>	50	75-25
25	<b>812</b>	<b>2K</b>	50	75-25
50	<b>485</b>	<b>1M</b>	50	75-25
75	<b>2860</b>	<b>10K</b>	50	75-25
100	<b>2256</b>	<b>50M</b>	50	75-25

(a) Varying the key range in the workload.

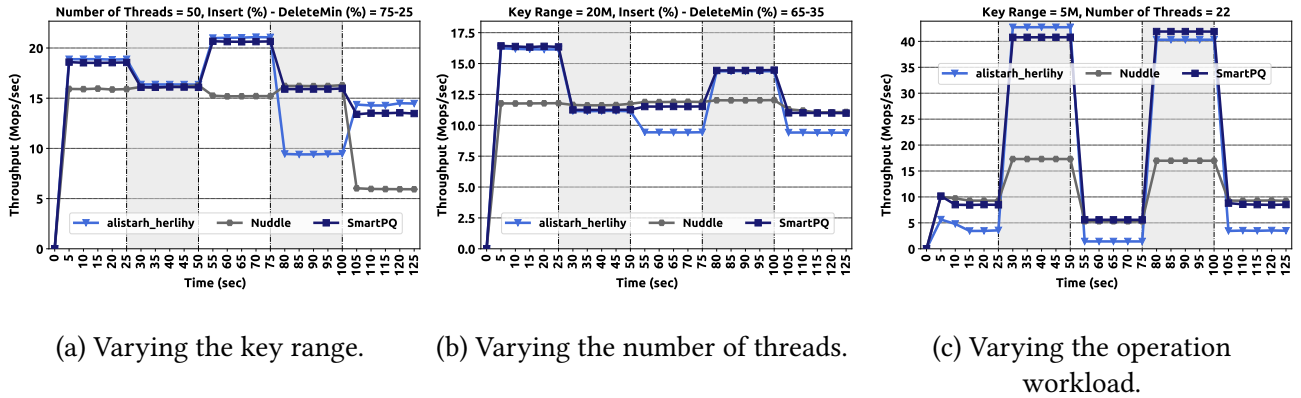
Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	<b>1166</b>	20M	<b>57</b>	65-35
25	<b>15567</b>	20M	<b>29</b>	65-35
50	<b>15417</b>	20M	<b>15</b>	65-35
75	<b>15297</b>	20M	<b>43</b>	65-35
100	<b>15346</b>	20M	<b>15</b>	65-35

(b) Varying the number of threads that perform operations in the data structure.

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	<b>1M</b>	5M	22	<b>50-50</b>
25	<b>140</b>	5M	22	<b>100-0</b>
50	<b>7403</b>	5M	22	<b>30-70</b>
75	<b>962</b>	5M	22	<b>100-0</b>
100	<b>8236</b>	5M	22	<b>0-100</b>

(c) Varying the percentage of *insert/deleteMin* operations.

Table 3.2: Features of the contention workload for benchmarks evaluated in Figure 3.10. We use bold font on the features that change in each execution phase.

Figure 3.10: Throughput achieved by *SmartPQ*, *Nuddle* and its underlying *base algorithm* (*alistarh\_herlihy*), in synthetic benchmarks, in which we vary a) the key range, b) the number of threads that perform operations in the data structure, and c) the percentage of *insert/deleteMin* operations in the workload.

contention workload vary during the execution, *SmartPQ* outperforms *alistarh\_herlihy* and *Nuddle* by  $1.87\times$  and  $1.38\times$  on average, respectively. Note that any of the contention workloads evaluated in Figures 3.10 and 3.11 belongs in the training data set used for training our classifier. Third, we note that the decision-making mechanism of *SmartPQ* has very low performance overheads. Across all

evaluated benchmarks, *SmartPQ* achieves *only up to* 5.3% performance slowdown (i.e., when using a range of 50M keys in Figure 3.10a) over the corresponding baseline implementation (*alistarh\_herlihy* priority queue). Note that since the proposed decision tree classifier has very low traversal cost (Section 3.3.1), we intentionally set a frequent time interval (i.e., one second) for calling the classifier, such that *SmartPQ* detects the contention workload change *on time*, and quickly adapts itself to the best-performing algorithmic mode. We also tried large time intervals, and observed that *SmartPQ* slightly delays to detect the contention workload change, thus achieving lower throughput in the transition points.

Overall, we conclude that *SmartPQ* performs best across *all* contention workloads and at *any* point in time, and incurs negligible performance overheads over the corresponding baseline implementation.

Time (sec)	Current Size	Key Range	Number of Threads	Insert - DeleteMin (%)
0	<b>1M</b>	10M	57	50-50
25	<b>26</b>	10M	<b>36</b>	<b>70-30</b>
50	<b>12</b>	<b>20M</b>	36	<b>50-50</b>
75	<b>79</b>	20M	36	<b>80-20</b>
100	<b>29K</b>	20M	<b>50</b>	80-20
125	<b>319K</b>	<b>100M</b>	50	<b>50-50</b>
150	<b>13</b>	100M	<b>57</b>	50-50
175	<b>524K</b>	100M	<b>22</b>	<b>100-0</b>
200	<b>524K</b>	100M	22	<b>50-50</b>
225	<b>1142</b>	100M	22	50-50
250	<b>463</b>	<b>200M</b>	<b>57</b>	<b>0-100</b>
275	<b>253</b>	200M	57	<b>100-0</b>
300	<b>33K</b>	<b>20M</b>	57	<b>0-100</b>
325	<b>142</b>	20M	<b>29</b>	<b>80-20</b>
350	<b>25K</b>	20M	29	<b>50-50</b>

Table 3.3: Features of the contention workload for benchmarks evaluated in Figure 3.11. We use bold font on the features that change in each execution phase.

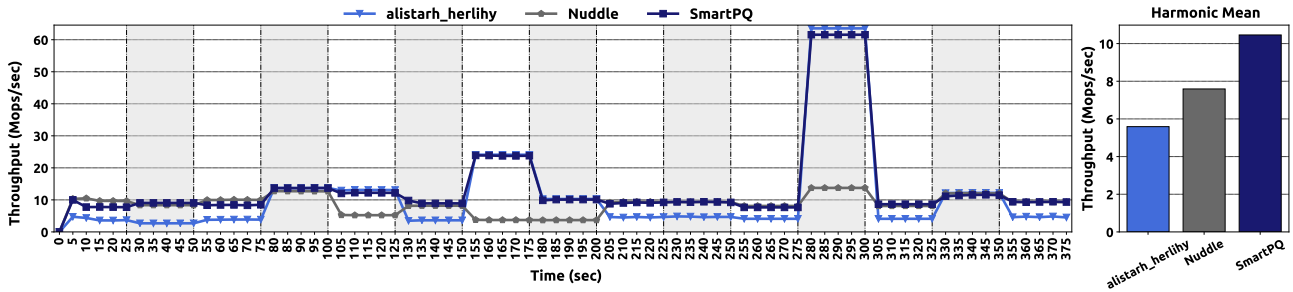


Figure 3.11: Throughput achieved by *SmartPQ*, *Nuddle* and its underlying *base algorithm* (*alistarh\_herlihy*), in synthetic benchmarks, in which we vary multiple features in the contention workload.



### 3.5 Discussion and Future Work

In Section 3.3.1, we assume that the contention workload is known a priori to extract the features needed for classification. To *on-the-fly* extract these features, and *dynamically* detect when contention changes, the main structure of *SmartPQ*, i.e., *struct smartpq*, needs to be enriched with additional fields to keep track of workload statistics (e.g., the number of completed *insert/deleteMin* operations, the number of active threads that perform operations on the data structure, the minimum and/or maximum key that has been requested so far). Active threads that perform operations on the data structure could atomically update these statistics. In frequent time lapses, either a background thread or an active thread could extract the features needed for classification based on the workload statistics, and call the classifier to predict if a transition to another algorithmic mode is needed. Finally, an additional parameter could be also added in *SmartPQ* to configure how often to collect workload statistics.

In our experimental evaluation, we pin server threads on a single NUMA node and client threads on all nodes. We have chosen to do so (i) for simplicity, given that this approach fits well with our microbenchmark-based evaluation, and (ii) because this is par with prior works on concurrent data structures [13, 15, 59, 61, 67, 74, 76, 86, 334, 442, 443, 447, 458–461]. In a real-world scenario, where *SmartPQ* is used as a part of a high-level application, client threads do *not* need to be pinned in hardware contexts, and they can be allowed to run in any core of the system. However, for our approach to be meaningful server threads need to be limited on a single NUMA node. This can easily be done by creating the server threads when *SmartPQ* is initialized, and pinning them to hardware contexts that are located at the same NUMA node. In this case, server threads are background threads that only accept and serve requests from various client threads, which are part of the high-level application.

Finally, even though we focus on a microbenchmark-based evaluation to cover a *wide variety* of contention scenarios, it is one of our future directions to explore the efficiency of *SmartPQ* in real-world applications, such as web servers [462, 463], graph traversal applications [59, 359] and scheduling in operating systems [464]. As future work, we also aim to investigate the applicability of our approach in other data structures, that may have similar behavior with priority queues (e.g., skip lists, search trees), and extend our proposed classifier (e.g., adding more features) to cover a variety of NUMA CPU-centric systems with different architectural characteristics.

### 3.6 Recommendations

**Recommendation.** *Design adaptive parallel algorithms and concurrent data structures that on-the-fly adjust their parallelization approach and synchronization scheme depending on the dynamic workload demands and contention.*

Our work demonstrates (Figures 3.1 and 3.7) that there is no one-size-fits-all algorithmic mode (between NUMA-oblivious and NUMA-aware) for a concurrent priority queue in modern computing systems: the best-performing algorithmic mode depends on multiple characteristics, including the

contention/operation workload, the size of the data structure and the underlying hardware platform [4]. Such characteristics can *dynamically* change during runtime, when performing various operations (e.g., *insert*, *deleteMin*) in the data structures used. Therefore, we conclude that to achieve high system performance in real-world scenarios, we need to *dynamically* tune the configuration of parallel kernels based on the characteristics of the current load at each time. To this end, we recommend that software designers propose adaptive parallel algorithms and concurrent data structures that *dynamically* adjust their parallelization technique and synchronization approach depending on the dynamic contention and workload demands. For example, machine learning, dynamic profiling and statistical approaches [4,298,465] could be integrated in parallel kernels to improve performance.

### 3.7 Related Work

To our knowledge, this is the first work to propose an adaptive priority queue for NUMA systems, which performs best under *all* various contention workloads, and even when contention varies over time. We briefly discuss prior work.

**Concurrent Priority Queues.** A large corpus of work proposes concurrent algorithms for priority queues [13,59–68,77,78], or generally for skip lists [14,69–76]. Recent works [77,78] designed lock-free priority queues that separate the logical and the physical deletion of an element to increase parallelism. Alistarh et al. [13] design a relaxed priority queue, called *SprayList*, in which *deleteMin* operation returns with a high probability, an element among the *first*  $\mathcal{O}(p \log 3p)$  elements of the priority queue, where  $p$  is the number of threads. Sagonas et al [67] design a contention avoiding technique, in which *deleteMin* operation returns the highest-priority element of the priority queue under *low* contention, while it enables relaxed semantics when high contention is detected. Specifically, under high-contention a few *deleteMin* operations are queued, and later several elements are deleted from the head of the queue *at once* via a combined deletion operation. Heidarshenas et al. [364] design a novel architecture for relaxed priority queues. These prior approaches are NUMA-oblivious implementations. Thus, in NUMA systems, they incur significant performance degradation in high-contention scenarios (e.g., *deleteMin*-dominated workloads in Section 3.4.1). In contrast, Calciu et al. [61] propose a *NUMA-friendly* priority queue employing the combining and elimination techniques. Elimination allows the complementary operations, i.e., *insert* with *deleteMin*, to complete *without* updating the data structure, while combining is a technique similar to the delegation technique [81–85] of *Nuddle* and *ffwd* [15]. Finally, Daly et al. [442] propose an efficient technique to obtain NUMA-aware skip lists, which however, can only be applied to skip list-based data structures. In contrast, *Nuddle* is a *generic* technique to obtain NUMA-aware data structures.

**Black-Box Approaches.** Researchers have also proposed black-box approaches: any data structure can be made wait-free or NUMA-aware without effort or knowledge on parallel programming or NUMA architectures. Herlihy [466] provides a universal method to design wait-free implementations of any sequential object. However, this method remains impractical due to high overheads. Hendler et al. [79] propose flat combining; a technique to reduce synchronization overheads by executing multiple client threads’ requests *at once*. Despite significant improvements [80], this technique provides

high performance *only* for a few data structures (e.g., synchronous queues). *ffwd* [15] is black-box approach, which uses the delegation technique [81–85] to eliminate cache line invalidation traffic over the interconnect. However, *ffwd* is limited to single threaded performance. Calciu et al. [86] propose a black-box technique, named *Node Replication*, to obtain concurrent NUMA-aware data structures. In *Node Replication*, every NUMA node has replicas of the shared data structure, which are synchronized via a shared log. Even though *ffwd* and *Node Replication* are generic techniques to obtain NUMA-aware data structures, similarly to *Nuddle*, both of them use a *serial asynchronized* implementation as the underlying *base algorithm*. Therefore, if they are used as the NUMA-aware algorithmic mode in an adaptive data structure, which dynamically tunes itself between a NUMA-oblivious and a NUMA-aware mode, both *ffwd* and *Node Replication* need a synchronization point between transitions to ensure correctness. Consequently, they would incur high performance overheads, when transitions between algorithmic modes happen at a non-negligible frequency.

**Machine Learning in Data Structures.** Even though machine learning is widely used to improve performance in many emerging applications [88–100], there are a handful of works [87, 101] that leverage machine learning to design *highly-efficient* concurrent data structures. Recently, Eastep et al. [101] use reinforcement learning to on-the-fly tune a parameter in the flat combining technique [79, 80], which is used in skip lists and priority queues. Kraska et al. [87] demonstrate that machine learning models can be trained to predict the position or existence of elements in key-value lookup sets, and discuss under which conditions learned index models can outperform the traditional indexed data structures (e.g., B-trees).

### 3.8 Summary

We propose *SmartPQ*, an adaptive concurrent priority queue for NUMA architectures, which performs best under *all* various contention scenarios, and even when contention varies over time. *SmartPQ* has two key components. First, it is built on top of *Nuddle*; a generic low-overhead technique to obtain efficient NUMA-aware data structures using *any* concurrent NUMA-oblivious implementation as its backbone. Second, *SmartPQ* integrates a lightweight decision-making mechanism, which is based on a simple decision tree classifier, to decide when to switch between *Nuddle*, i.e., a NUMA-aware algorithmic mode, and its underlying *base algorithm*, i.e., a NUMA-oblivious algorithmic mode. Our evaluation over a wide range of contention scenarios demonstrates that *SmartPQ* switches between the two algorithmic modes with negligible overheads, and significantly outperforms prior schemes, even when contention varies over time. We conclude that *SmartPQ* is an efficient concurrent priority queue for NUMA systems, and hope that this work encourages further study on adaptive concurrent data structures for NUMA architectures.



# CHAPTER 4

---

## *SynCron*

---

### 4.1 Overview

Recent advances in 3D-stacked memories [467–472] have renewed interest in Near-Data Processing (NDP) [32, 36, 268, 473]. NDP involves performing computation close to where the application data resides. This alleviates the expensive data movement between processors and memory, yielding significant performance improvements and energy savings in parallel applications. Placing low-power cores or special-purpose accelerators (hereafter called NDP cores) close to the memory dies of high-bandwidth 3D-stacked memories is a commonly-proposed design for NDP systems [21–25, 27–29, 32–37, 158, 159, 208, 212, 213, 216, 265–267, 366–368, 473–477]. Typical NDP architectures support several NDP units connected to each other, with each unit comprising multiple NDP cores close to memory [21, 32, 34, 35, 366–368]. Therefore, NDP architectures provide high levels of parallelism, low memory access latency, and large aggregate memory bandwidth.

Recent research demonstrates the benefits of NDP for parallel applications, e.g., for genome analysis [25, 27], graph processing [29, 32–37], databases [28, 29], security [214], pointer-chasing work-

loads [76, 215, 216, 369], and neural networks [21–24]. In general, these applications exhibit high parallelism, low operational intensity, and relatively low cache locality [161, 478–481], which make them suitable for NDP.

Prior works discuss the need for efficient synchronization primitives in NDP systems, such as locks [76, 369] and barriers [32, 34, 35, 212]. Synchronization primitives are widely used by multi-threaded applications [1, 4, 44, 85, 217, 482–486], and must be carefully designed to fit the underlying hardware requirements to achieve high performance. Therefore, to fully leverage the benefits of NDP for parallel applications, an effective synchronization solution for NDP systems is necessary.

Approaches to support synchronization are typically of two types [487, 488]. First, synchronization primitives can be built through *shared memory*, most commonly using the atomic read-modify-write (*rmw*) operations provided by hardware. In CPU systems, atomic *rmw* operations are typically implemented upon the underlying hardware cache coherence protocols, but many NDP systems do *not* support hardware cache coherence (e.g., [32, 34, 35, 159, 367]). In GPUs and Massively Parallel Processing systems (MPPs), atomic *rmw* operations can be implemented in dedicated hardware atomic units, known as *remote atomics*. However, synchronization using remote atomics has been shown to be inefficient, since sending every update to a fixed location creates high global traffic and hotspots [153, 370, 371, 381, 382]. Second, synchronization can be implemented via a *message-passing* scheme, where cores exchange messages to reach an agreement. Some recent NDP works (e.g., [32, 35, 212, 385]) propose message-passing barrier primitives among NDP cores of the system. However, these synchronization schemes are still inefficient, as we demonstrate in Section 4.6, and also lack support for lock, semaphore and condition variable synchronization primitives.

Hardware synchronization techniques that do not rely on hardware coherence protocols and atomic *rmw* operations have been proposed for multicore systems [299–301, 303–305, 307, 308]. However, such synchronization schemes are tailored for the specific architecture of each system, and are not efficient or suitable for NDP systems (Section 4.8). For instance, CM5 [308] provides a barrier primitive via a dedicated physical network, which would incur high hardware cost to be supported in large-scale NDP systems. LCU [307] adds a control unit to *each* CPU core and a buffer to each memory controller, which would also incur high cost to implement in *area-constrained* NDP cores and controllers. SSB [300] includes a small buffer attached to each controller of the last level cache (LLC) and MiSAR [299] introduces an accelerator distributed at the LLC. Both schemes are built on the shared cache level in CPU systems, which most NDP systems do *not* have. Moreover, in NDP systems with *non-uniform* memory access times, most of these prior schemes would incur significant performance overheads under high-contention scenarios. This is because they are oblivious to the non-uniformity of NDP, and thus would cause excessive traffic across NDP units of the system upon contention (Section 4.6.7).

Overall, NDP architectures have several important characteristics that necessitate a new approach to support efficient synchronization. First, most NDP architectures [21, 22, 24, 28, 32, 34, 35, 76, 158, 159, 208, 212, 213, 216, 266, 267, 385, 473, 474] lack shared caches that can enable low-cost communication and synchronization among NDP cores of the system. Second, hardware cache coherence protocols are typically not supported in NDP systems [21–24, 28, 32, 34, 35, 76, 158, 208, 212, 213, 216, 266, 385, 474],

due to high area and traffic overheads associated with such protocols [159, 367]. Third, NDP systems are non-uniform, distributed architectures, in which inter-unit communication is more expensive (both in performance and energy) than intra-unit communication [28, 29, 32, 34, 35, 37, 212, 366].

In this work, we present *SynCron*, an efficient synchronization mechanism for NDP architectures. *SynCron* is designed to achieve the goals of performance, cost, programming ease, and generality to cover a wide range of synchronization primitives through four key techniques. First, we offload synchronization among NDP cores to dedicated low-cost hardware units, called Synchronization Engines (SEs). This approach avoids the need for complex coherence protocols and expensive *rmw* operations, at low hardware cost. Second, we directly buffer the synchronization variables in a specialized cache memory structure to avoid costly memory accesses for synchronization. Third, *SynCron* coordinates synchronization with a hierarchical message-passing scheme: NDP cores only communicate with their local SE that is located in the same NDP unit. At the next level of communication, all local SEs of the system’s NDP units communicate with each other to coordinate synchronization at a global level. Via its hierarchical communication protocol, *SynCron* significantly reduces synchronization traffic across NDP units under high-contention scenarios. Fourth, when applications with frequent synchronization oversubscribe the hardware synchronization resources, *SynCron* uses an efficient and programmer-transparent overflow management scheme that avoids costly fallback solutions and minimizes overheads.

We evaluate *SynCron* using a wide range of parallel workloads including pointer-chasing, graph applications, and time series analysis. Over prior approaches (similar to [32, 212]), *SynCron* improves performance by  $1.27\times$  on average (up to  $1.78\times$ ) under high-contention scenarios, and by  $1.35\times$  on average (up to  $2.29\times$ ) under low-contention scenarios. In real applications with fine-grained synchronization, *SynCron* comes within 9.5% of the performance and 6.2% of the energy of an ideal zero-overhead synchronization mechanism. Our proposed hardware unit incurs very modest area and power overheads (Section 4.6.8) when integrated into the compute die of an NDP unit.

The main **contributions** of this work are:

- We investigate the challenges of providing efficient synchronization in Near-Data-Processing architectures, and propose an end-to-end mechanism, *SynCron*, for such systems.
- We design low-cost synchronization units that coordinate synchronization across NDP cores, and directly buffer synchronization variables to avoid costly memory accesses to them. We propose an efficient message-passing synchronization approach that organizes the process hierarchically, and provide a hardware-only programmer-transparent overflow management scheme to alleviate performance overheads when hardware synchronization resources are exceeded.
- We evaluate *SynCron* using a wide range of parallel workloads and demonstrate that it significantly outperforms prior approaches both in performance and energy consumption. *SynCron* also has low hardware area and power overheads.

## 4.2 Background and Motivation

### 4.2.1 Baseline Architecture

Numerous works [21–23, 28, 29, 32–37, 76, 212, 214, 216, 286, 367, 369, 385, 489] show the potential benefit of NDP for parallel, irregular applications. These proposals focus on the design of the compute logic that is placed close to or within memory, and in many cases provide special-purpose near-data accelerators for specific applications. Figure 4.1 shows the baseline organization of the NDP architecture we assume in this work, which includes several NDP units connected with each other via serial interconnection links to share the same physical address space. Each NDP unit includes the memory arrays and a compute die with multiple low-power programmable cores or fixed-function accelerators, which we henceforth refer to as NDP cores. NDP cores execute the offloaded NDP kernel and access the various memory locations across NDP units with non-uniform access times [28, 29, 32, 34, 35, 37, 367]. We assume that there is no OS running in the NDP system. In our evaluation, we use programmable in-order NDP cores, each including small private L1 I/D caches. However, *SynCron* can be used with any programmable, fixed-function or reconfigurable NDP accelerator. We assume software-assisted cache-coherence (provided by the operating system or the programmer), similar to [212, 367]: data can be either thread-private, shared read-only, or shared read-write. Thread-private and shared read-only data can be cached by NDP cores, while shared read-write data is uncacheable.

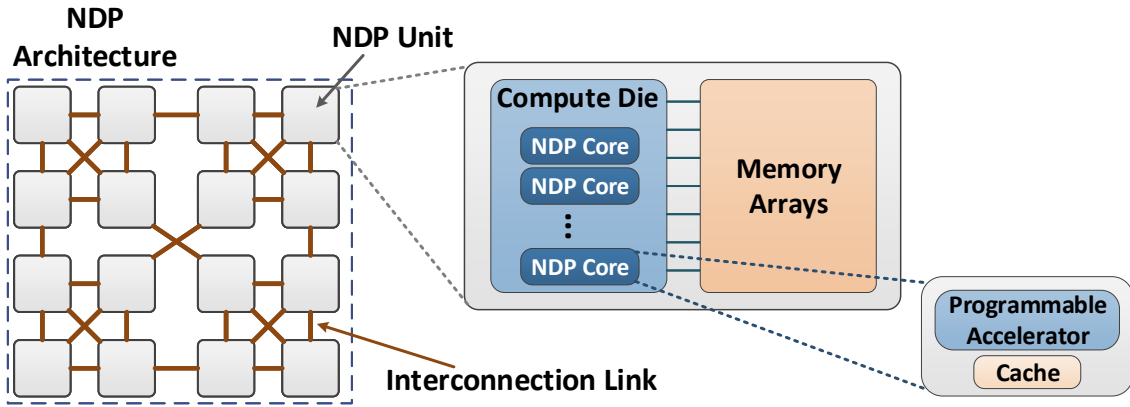


Figure 4.1: High-level organization of an NDP architecture.

We focus on three characteristics of NDP architectures that are of particular importance in the synchronization context. First, NDP architectures typically do not have a shared level of cache memory [21, 22, 24, 28, 32, 34, 35, 76, 158, 159, 208, 212, 213, 216, 266, 267, 385, 473, 474], since the NDP-suited workloads usually do not benefit from deep cache hierarchies due to their poor locality [212, 367, 478, 481]. Second, NDP architectures do not typically support conventional hardware cache coherence protocols [21–24, 28, 32, 34, 35, 76, 158, 208, 212, 213, 216, 266, 385, 474], because they would add area and traffic overheads [159, 367], and would incur high complexity and latency [154], limiting the benefits of NDP. Third, communication across NDP units is expensive, because NDP systems are non-uniform distributed architectures. The energy and performance costs of inter-unit



communication are typically orders of magnitude greater than the costs of intra-unit communication [28, 29, 32, 34, 35, 37, 212, 366], and thus inter-unit communication may slow down the execution of NDP cores [34].

## 4.2.2 The Solution Space for Synchronization

Approaches to support synchronization are typically either via shared memory or message-passing schemes.

### Synchronization via Shared Memory

In this case, cores coordinate via a consistent view of shared memory locations, using atomic read/write operations or atomic read-modify-write (*rmw*) operations. If *rmw* operations are *not* supported by hardware, Lamport’s bakery algorithm [490] can provide synchronization to  $N$  participating cores, assuming sequential consistency [491]. However, this scheme scales poorly, as a core accesses  $O(N)$  memory locations at *each* synchronization retry. In contrast, commodity systems (CPUs, GPUs, MPPs) typically support *rmw* operations in hardware.

GPUs and MPPs support *rmw* operations in specialized hardware units (known as *remote atomics*), located in each bank of the shared cache [492, 493], or the memory controllers [315, 372]. Remote atomics are also supported by an NDP work [212] at the vault controllers of Hybrid Memory Cube (HMC) [468, 470]. Implementing synchronization primitives using remote atomics requires a spin-wait scheme, i.e., executing consecutive *rmw* retries. However, performing and sending every *rmw* operation to a shared, fixed location can cause high global traffic and create hotspots [153, 370, 371, 381, 382]. In NDP systems, consecutive *rmw* operations to a remote NDP unit would incur high traffic *across* NDP units, with high performance and energy overheads.

Commodity CPU architectures support *rmw* operations either by locking the bus (or equivalent link), or by relying on the hardware cache coherence protocol [494, 495], which many NDP architectures do not support. Therefore, coherence-based synchronization [456, 496–506] cannot be directly implemented in NDP architectures. Moreover, based on prior works on synchronization [16, 455, 457, 482, 507, 508], coherence-based synchronization would exhibit low scalability on NDP systems for two reasons. First, it performs poorly with a *large* number of cores, due to low scalability of conventional hardware coherence protocols [494, 509–511]. Most NDP systems include several NDP units [32, 34, 35, 366], each typically supporting hundreds of small, area-constrained cores [21, 32, 34, 35]. Second, the non-uniformity in memory accesses significantly affects the scalability of coherence-based synchronization [16, 455–457]. Prior work on coherence-based synchronization [16] observes that the latency of a lock acquisition that needs to transfer the lock *across* NUMA sockets can be up to  $12.5\times$  higher than that *within* a socket. We expect such effects to be aggravated in NDP systems, since they are by nature *non-uniform* and *distributed* [28, 29, 32, 34, 35, 37, 212, 366] with very low memory access latency within an NDP unit.

We validate these observations on both a real CPU and our simulated NDP system. On an Intel Xeon Gold server, we evaluate the operation throughput achieved by two coherence-based lock

Million Operations per Second	1 thread single-socket	14 threads single-socket	2 threads same-socket	2 threads different-socket
TTAS lock [497]	8.92	2.28	9.91	4.32
Hierarchical Ticket lock [499]	8.06	2.91	9.01	6.79

Table 4.1: Throughput of two coherence-based lock algorithms on an Intel Xeon Gold server using the liblock library [16].

algorithms (Table 4.1), i.e., TTAS [497] and Hierarchical Ticket Lock (HTL) [499], using a microbenchmark taken from the *liblock* library [16]. When increasing the number of threads from 1 to 14 within a single socket, throughput drops by  $3.91\times$  and  $2.77\times$  for TTAS and HTL, respectively. Moreover, when pinning two threads on different NUMA sockets, throughput drops by up to  $2.29\times$  over when pinning them on the same socket, due to non-uniform memory access times of lock variables.

In our simulated NDP system, we evaluate the performance achieved by a stack data structure protected with a coarse-grained lock. Figure 4.2 shows the slowdown of the stack when using a coherence-based lock [487] (*mesi-lock*), implemented upon a MESI directory coherence protocol, over using an ideal lock with zero cost for synchronization (*ideal-lock*). First, we observe that the high contention for the cache line containing the *mesi-lock* and the resulting coherence traffic inside the network significantly limit scalability of the stack as the number of cores increases. With 60 NDP cores within a single NDP unit (Figure 4.2a), the stack with *mesi-lock* incurs  $2.03\times$  slowdown over *ideal-lock*. Second, we notice that the non-uniform memory accesses to the cache line containing the *mesi-lock* also impact the scalability of the stack. When increasing the number of NDP units while keeping total core count constant at 60 (Figure 4.2b), the slowdown of the stack with *mesi-lock* increases to  $2.66\times$  (using 4 NDP units) over *ideal-lock*. In *non-uniform* NDP systems, the scalability of coherence-based synchronization is severely limited by the long transfer latency and low bandwidth of the interconnect used between the NDP units.

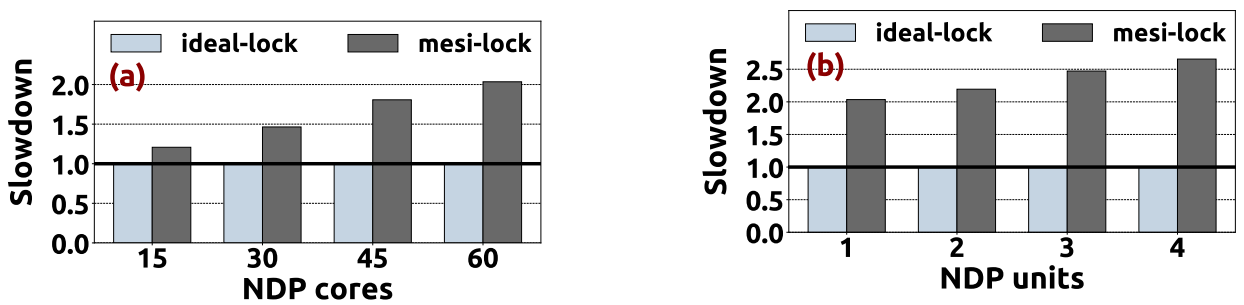


Figure 4.2: Slowdown of a stack data structure using a coherence-based lock over using an *ideal* zero-cost lock, when varying (a) the NDP cores within a single NDP unit and (b) the number of NDP units while keeping core count constant at 60.

### Message-passing Synchronization

In this approach, cores coordinate with each other by exchanging messages (either in software or hardware) in order to reach an agreement. For instance, a recent NDP work [32] implements a barrier primitive via hardware message-passing communication among NDP cores, i.e., one core of the

system works as a *master* core to collect the synchronization status of the rest. To improve system performance in *non-uniform* HMC-based NDP systems, Gao et al. [212] propose a *tree-style* barrier primitive, where cores exchange messages to first synchronize within a vault, then across the vaults of an HMC cube, and finally across HMC cubes. In general, optimized message-passing synchronization schemes proposed in the literature [212,303,488,512–514] aim to minimize (i) the number of messages sent among cores, and (ii) expensive network traffic. To avoid the major issues of synchronization via shared memory described above, we design our approach building on the message-passing synchronization concept.

### 4.3 *SynCron*: Overview

*SynCron* is an end-to-end solution for synchronization in NDP architectures that improves performance, has low cost, eases programmability, and supports multiple synchronization primitives. *SynCron* relies on the following key techniques:

- 1. Hardware support for synchronization acceleration:** We design low-cost hardware units, called Synchronization Engines (SEs), to coordinate the synchronization among NDP cores of the system. SEs eliminate the need for complex cache coherence protocols and expensive *rmw* operations, and incur modest hardware cost.
- 2. Direct buffering of synchronization variables:** We add a specialized cache structure, the Synchronization Table (ST), inside an SE to keep synchronization information. Such direct buffering avoids costly memory accesses for synchronization, and enables high performance under low-contention scenarios.
- 3. Hierarchical message-passing communication:** We organize the communication hierarchically, with each NDP unit including an SE. NDP cores communicate with their local SE that is located in the same NDP unit. SEs communicate with each other to coordinate synchronization at a global level. Hierarchical communication minimizes expensive communication *across* NDP units, and achieves high performance under high-contention scenarios.
- 4. Integrated hardware-only overflow management:** We incorporate a hardware-only overflow management scheme to efficiently handle scenarios when ST is fully occupied. This programmer-transparent technique effectively limits performance degradation under overflow scenarios.

#### 4.3.1 Overview of *SynCron*

Figure 4.3 provides an overview of our approach. *SynCron* exposes a simple programming interface such that programmers can easily use a variety of synchronization primitives in their multithreaded applications when writing them for NDP systems. The interface is implemented using two new instructions that are used by NDP cores to communicate synchronization requests to SEs. These are general enough to cover all semantics for the most widely used synchronization primitives.

We add one SE in the compute die of each NDP unit. For a particular synchronization variable allocated in an NDP unit, the SE that is physically located in the same NDP unit is considered the

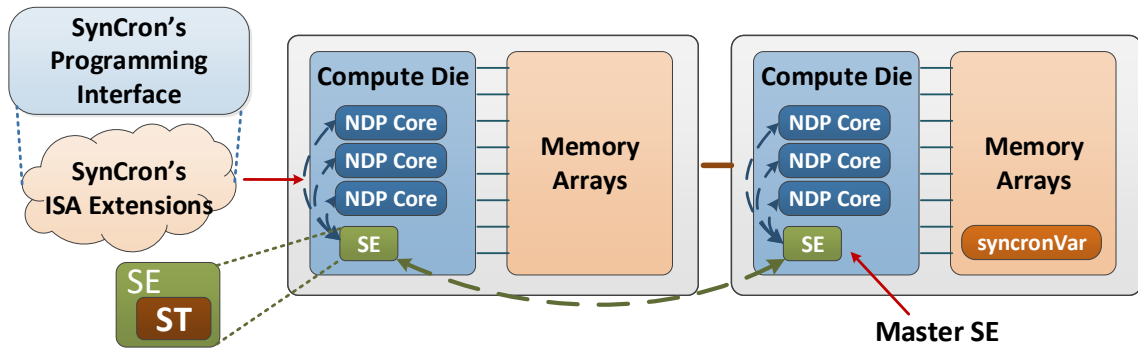


Figure 4.3: High-level overview of *SynCron*.

*Master SE*. In other words, the *Master SE* is defined by the address of the synchronization variable. It is responsible for the global coordination of synchronization on that variable, i.e., among all SEs of the system. All other SEs are responsible only for the local coordination of synchronization among the cores in the same NDP unit with them.

NDP cores act as clients that send requests to SEs via hardware message-passing. SEs act as servers that process synchronization requests. In the proposed hierarchical communication, NDP cores send requests to their local SEs, while SEs of different NDP units communicate with the *Master SE* of the specific variable, to coordinate the process at a global level, i.e., among all NDP units.

When an SE receives a request from an NDP core for a synchronization variable, it directly buffers the variable in its ST, keeping all the information needed for synchronization in the ST. If the ST is full, we use the main memory as a fallback solution. To hierarchically coordinate synchronization via main memory in ST overflow cases, we design (i) a generic structure, called *synchronVar*, to keep track of required synchronization information, and (ii) specialized *overflow* messages to be sent among SEs. The hierarchical communication among SEs is implemented via corresponding support in message encoding, the ST, and *synchronVar* structure.

### 4.3.2 *SynCron's Operation*

*SynCron* supports locks, barriers, semaphores, and condition variables. Here, we present *SynCron's* operation for locks. *SynCron* has similar behavior for the other three primitives.

**Lock Synchronization Primitive:** Figure 4.4 shows a system composed of two NDP units with two NDP cores each. In this example, all cores request and compete for the same lock. First, all NDP cores send *local* lock acquire messages to their SEs ❶. After receiving these messages, each SE keeps track of its requesting cores by reserving one new entry in its ST, i.e., directly buffering the lock variable in ST. Each ST entry includes a local waiting list (i.e., a hardware bit queue with one bit for each local NDP core), and a global waiting list (i.e., a bit queue with one bit for each SE of the system). To keep track of the requesting cores, each SE sets the bits corresponding to the requesting cores in the local waiting list of the ST entry. When the local SE receives a request for a synchronization variable *for the first time*, it sends a *global* lock acquire message to the *Master SE* ❷, which in turn sets the corresponding bit in the global waiting list in its ST. This way, the *Master SE* keeps track of all requests to a particular variable coming from an SE, and can arbitrate between different SEs. The local SE can

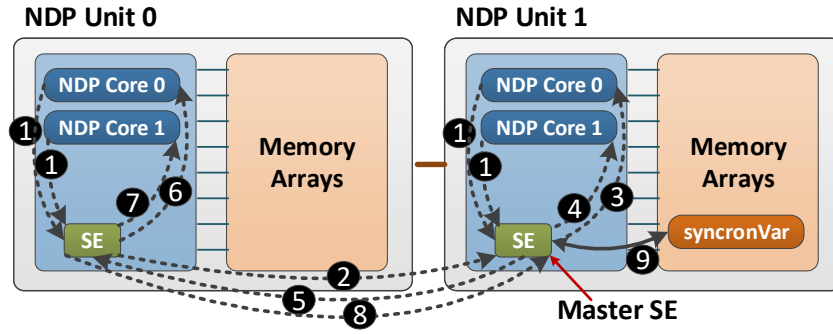


Figure 4.4: An example execution scenario for a lock requested by *all* NDP cores.

then serve successive local requests to the same variable until there are no other local requests. By using the proposed hierarchical communication protocol, the cores send local messages to their local SE, and the SE needs to send *only one aggregated* message, on behalf of all its local waiting cores, to the *Master SE*. As a result, we reduce the need for communication through the narrow, expensive links that connect different NDP units.

The *Master SE* first prioritizes the local waiting list, granting the lock to its own local NDP cores in sequence (e.g., to NDP Core 0 first ③, and to NDP Core 1 next ④ in Figure 4.4). At the end of the critical section, each local lock owner sends a lock release message to its SE in order to release the lock. When there are no other local requests, the *Master SE* transfers the control of the lock to the SE of another NDP unit based on its global waiting list ⑤. Then, the local SE grants the lock to its local NDP cores in sequence (e.g., ⑥, ⑦). After all local cores release the lock, the SE sends an *aggregated* global lock release message to the *Master SE* ⑧ and releases its ST entry. When the message arrives at the *Master SE*, if there are no other pending requests to the same variable, the *Master SE* releases its ST entry. In this example, SEs directly buffer the lock variable in their STs. If an ST is *full*, the *Master SE* globally coordinates synchronization by keeping track of all required information in main memory ⑨, via our proposed overflow management scheme (Section 4.4.3).

## 4.4 SynCron: Detailed Design

*SynCron* leverages the key observation that all synchronization primitives fundamentally communicate the same information, i.e., a waiting list of cores that participate in the synchronization process, and a condition to be met to notify one or more cores. Based on this observation, we design *SynCron* to cover the four most widely used synchronization primitives. Without loss of generality, we assume that each NDP core represents a hardware thread context with a unique ID. To support multiple hardware thread contexts per NDP core, the corresponding hardware structures of *SynCron* need to be augmented to include 1-bit per hardware thread context.

### 4.4.1 Programming Interface and ISA Extensions

*SynCron* provides lock, barrier, semaphore and condition variable synchronization primitives, supporting two types of barriers: within cores of the *same* NDP unit and within cores across different

NDP units of the system. *SynCron*'s programming interface (Table 4.2) implements the synchronization semantics with two new ISA instructions, which are *rich* and *general* enough to express all supported primitives. NDP cores use these instructions to assemble messages for synchronization requests, which are issued through the network to SEs.

---

### ***SynCron* Programming Interface**

---

```

syncronVar *create_syncvar ();
void destroy_syncvar (syncronVar *svar);
void lock_acquire (syncronVar *lock);
void lock_release (syncronVar *lock);
void barrier_wait_within_unit (syncronVar *bar, int initialCores);
void barrier_wait_across_units (syncronVar *bar, int initialCores);
void sem_wait (syncronVar *sem, int initialResources);
void sem_post (syncronVar *sem);
void cond_wait (syncronVar *cond, syncronVar *lock);
void cond_signal (syncronVar *cond);
void cond_broadcast (syncronVar *cond);

```

---

Table 4.2: *SynCron*'s Programming Interface (i.e., API).

***req\_sync addr, opcode, info***: This instruction creates a message and commits when a response message is received back. The *addr* register has the address of a synchronization variable, the *opcode* register has the message opcode of a particular semantic of a synchronization primitive (Table 4.3), and the *info* register has specific information needed for the primitive (*MessageInfo* in message encoding of Fig. 4.5).

***req\_async addr, opcode***: This instruction creates a message and after the message is issued to the network, the instruction commits. The registers *addr*, *opcode* have the same semantics as in *req\_sync* instruction.

## **Memory Consistency**

We design *SynCron* assuming a relaxed consistency memory model. The proposed ISA extensions act as memory fences. First, *req\_sync*, commits once a message (ACK) is received (from the local SE to the core), which ensures that all following instructions will be issued after *req\_sync* has been completed. Its semantics is similar to those of the SYNC and ACQUIRE operations of Weak Ordering (WO) [515] and Release Consistency (RC) [515] models, respectively. Second, *req\_async*, does not require a return message (ACK). It is issued once all previous instructions are completed. Its semantics is similar to that of the RELEASE operation of RC [515]. In the case of WO, *req\_sync* is sufficient. In the case of RC, the *req\_sync* instruction is used for acquire-type semantics, i.e., lock\_acquire, barrier\_wait, semaphore\_wait and condition\_variable\_wait, while the *req\_async* instruction is used for release-type semantics, i.e., lock\_release, semaphore\_post, condition\_variable\_signal, and condition\_variable\_broadcast.

## Message Encoding

Figure 4.5 describes the encoding of the message used for communication between NDP cores and the SE. Each message includes: (i) the 64-bit address of the synchronization variable, (ii) the message opcode that implements the semantics of the different synchronization primitives (6 bits cover all message opcodes), (iii) the unique ID number of the NDP core (6 bits are sufficient for our simulated NDP system in Section 4.5), and (iv) a 64-bit field (*MessageInfo*) that communicates specific information needed for each different synchronization primitive, i.e., the number of the cores that participate in a barrier, the initial value of a semaphore, the address of the lock associated with a condition variable.

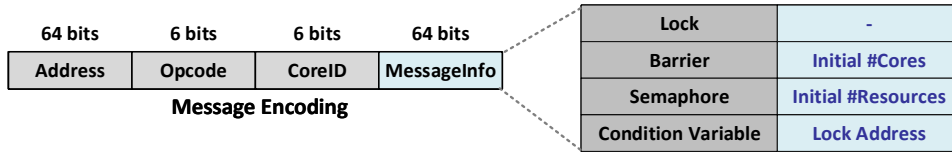


Figure 4.5: Message encoding of *SynCron*.

**Hierarchical Message Opcodes.** *SynCron* enables a hierarchical scheme, where the SEs of NDP units communicate with each other to coordinate synchronization at a global level. Therefore, we support two types of messages (Table 4.3): (i) *local*, which are used by NDP cores to communicate with their local SE, and (ii) *global*, which are used by SEs to communicate with the *Master SE*, and vice versa. Since we support two types of barriers (Table 4.2), we design two message opcodes for a *local* barrier\_wait message sent by an NDP core to its local SE: (i) *barrier\_wait\_local\_within\_unit* is used when cores of a single NDP unit participate in the barrier, and (ii) *barrier\_wait\_local\_across\_units* is used when cores from different NDP units participate in the barrier. In the latter case, if a *smaller* number of cores than the total *available* cores of the NDP system participate in the barrier, *SynCron* supports one-level communication: local SEs re-direct all messages (received from their local NDP cores) to the *Master SE*, which globally coordinates the barrier among *all* participating cores. This design choice is a trade-off between performance (*more remote messages*) and hardware/ISA complexity, since the number of participating cores of *each* NDP unit would need to be communicated to the hardware through additional registers in ISA, and message opcodes (*higher complexity*).

Primitives	<i>SynCron</i> Message Opcodes
<b>Locks</b>	lock_acquire_global, lock_acquire_local, lock_release_global
	lock_release_local, lock_grant_global, lock_grant_local
	lock_acquire_overflow, lock_release_overflow, lock_grant_overflow
<b>Barriers</b>	barrier_wait_global, barrier_wait_local_within_unit
	barrier_wait_local_across_units, barrier_depart_global, barrier_depart_local
	barrier_wait_overflow, barrier_departure_overflow
<b>Semaphores</b>	sem_wait_global, sem_wait_local, sem_grant_global
	sem_grant_local, sem_post_global, sem_post_local
	sem_wait_overflow, sem_grant_overflow, sem_post_overflow
<b>Condition Variables</b>	cond_wait_global, cond_wait_local, cond_signal_global
	cond_signal_local, cond_broad_global, cond_broad_local
	cond_grant_global, cond_grant_local, cond_wait_overflow
	cond_signal_overflow, cond_broad_overflow, cond_grant_overflow
<b>Other</b>	decrease_indexing_counter

Table 4.3: Message opcodes of *SynCron*.



### 4.4.2 Synchronization Engine (SE)

Each SE module (Figure 4.6) is integrated into the compute die of each NDP unit. An SE consists of *three* components:

#### Synchronization Processing Unit (SPU)

The SPU is the logic that handles the messages, updates the ST, and issues requests to memory as needed. The SPU includes the control unit, a buffer, and a few registers. The buffer is a small SRAM queue for temporarily storing messages that arrive at the SE. The control unit implements custom logic with simple logical bitwise operators (and, or, xor, zero) and multiplexers.

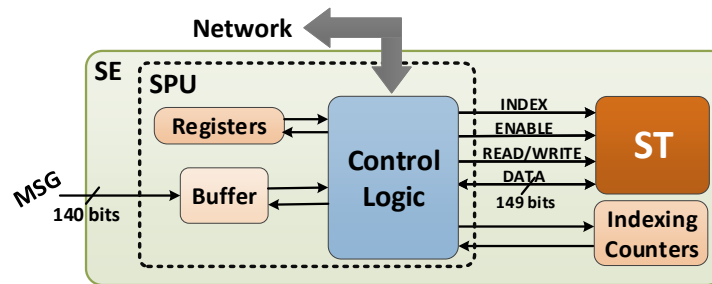


Figure 4.6: The Synchronization Engine (SE).

#### Synchronization Table (ST)

ST keeps track of all the information needed to coordinate synchronization. Each ST has 64 entries. Figure 4.7 shows an ST entry, which includes: (i) the 64-bit address of a synchronization variable, (ii) the global waiting list used by the *Master SE* for global synchronization among SEs, i.e., a hardware bit queue including one bit for each SE of the system, (iii) the local waiting list used by all SEs for synchronization among the NDP cores of an NDP unit, i.e., a hardware bit queue including one bit for each NDP core within the unit, (iv) the state of the ST entry, which can be either *free* or *occupied*, and (v) a 64-bit field (*TableInfo*) to track specific information needed for each synchronization primitive. For the lock primitive, the *TableInfo* field is used to indicate the lock owner that is either an SE of an NDP unit (*Global ID* represented by the most significant bits) or a *local* NDP core (*Local ID* represented by the least significant bits). We assume that all NDP cores of an NDP unit have a unique *local ID* within the NDP unit, while all SEs of the system have a unique *global ID* within the system. The number of bits in the global and local waiting lists of Figure 4.7 is specific for the configuration of our evaluated system (Section 4.5), which includes 16 NDP cores per NDP unit and 4 SEs (one per NDP unit), and has to be extended accordingly, if the system supports more NDP cores or SEs.

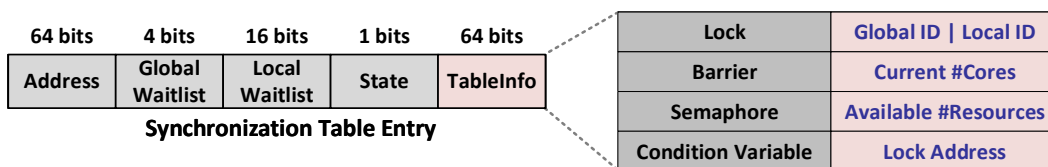


Figure 4.7: Synchronization Table (ST) entry.



## Indexing Counters

If an ST is full, i.e., all its entries are in *occupied* state, *SynCron* cannot keep track of information for a new synchronization variable in ST. We use the main memory as a fallback solution for such ST overflow (Section 4.4.3). The SE keeps track of *which* synchronization variables are currently serviced via main memory: similar to MiSAR [299], we include a small set of counters (*indexing counters*), 256 in current implementation, indexed by the least significant bits of the address of a synchronization variable, as extracted from the message that arrives at an SE. When an SE receives a message with acquire-type semantics for a synchronization variable and there is no corresponding entry in the *fully-occupied* ST, the indexing counter for that synchronization variable increases. When an SE receives a message with release-type semantics for a synchronization variable that is currently serviced using main memory, the corresponding indexing counter decreases. A synchronization variable is currently serviced via main memory, when the corresponding indexing counter is larger than zero. Note that different variables may alias to the same indexing counter. This aliasing does not affect correctness, but it does affect performance, since a variable may unnecessarily be serviced via main memory, while the ST is *not* full.

## Control Flow in SE

Figure 4.8 describes the control flow in SE. When an SE receives a message, it decodes the message **1** and accesses the ST **2a**. If there is an ST entry for the specific variable (depending on its address), the SE processes the waiting lists **3**, updates the ST **4a**, and encodes return message(s) **5**, if needed. If there is *not* an ST entry for the specific variable, the SE checks the value of the corresponding indexing counter **2b**: (i) if the indexing counter is zero *and* the ST is not full, the SE reserves a *new* ST entry and continues with step **3**, otherwise (ii) if the indexing counter is larger than zero *or* the ST is full, there is an overflow. In that case, if the SE is the *Master SE* for the specific variable, it reads the synchronization variable from *local* memory arrays **2c**, processes the waiting lists **3**, updates the variable in main memory **4b**, and encodes return message(s) **5**, if needed. If the SE is *not* the *Master SE* for the specific variable, it encodes an *overflow* message to the *Master SE* **2d** to handle overflow.

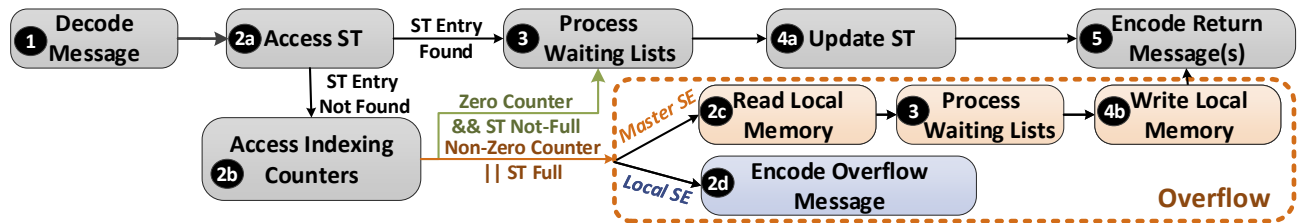


Figure 4.8: Control flow in SE.

### 4.4.3 Overflow Management

*SynCron* integrates a hardware-only overflow management scheme that provides very modest performance degradation (Section 4.6.7) and is programmer-transparent. To handle ST overflow cases,

we need to address two issues: (i) where to keep track of required information to coordinate synchronization, and (ii) how to coordinate ST overflow cases between SEs. For the former issue, we design a generic structure allocated in main memory. For the latter issue, we propose a hierarchical *overflow* communication protocol between SEs.

### SynCron's Synchronization Variable

We design a generic structure (Figure 4.9), called *synchronVar*, which is used to coordinate synchronization for all supported primitives in ST overflow cases. *synchronVar* is defined in the driver of the NDP system, which handles the allocation of the synchronization variables: programmers use *create\_syncvar()* (Table 4.2) to create a *new* synchronization variable, the driver allocates the bytes needed for *synchronVar* in main memory, and returns an opaque pointer that points to the address of the variable. Programmers should not de-reference the opaque pointer and its content can *only* be accessed via *SynCron*'s API (Table 4.2).

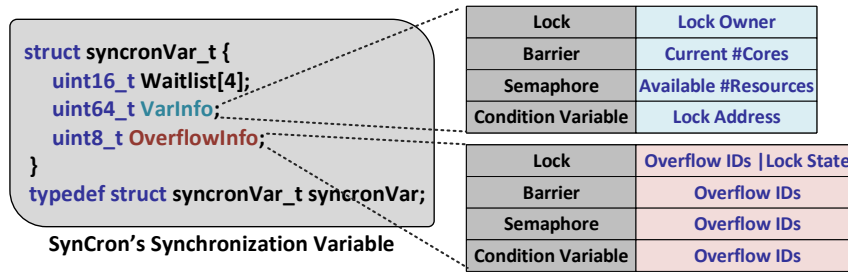


Figure 4.9: Synchronization variable of *SynCron* (*synchronVar*).

*synchronVar* structure includes one waiting list for each SE of the system, which has one bit for each NDP core within the NDP unit, and two additional fields (*VarInfo*, *OverflowInfo*) needed to hierarchically handle ST overflows for all primitives.

### Communication Protocol between SEs

To ensure correctness, *only* the *Master SE* updates the *synchronVar* variable: in ST overflow, the SPU of the *Master SE* issues read or write requests to its local memory to *globally* coordinate synchronization via the *synchronVar* variable. In our proposed hierarchical design, there are two overflow scenarios: (i) the ST of the *Master SE* overflows, and (ii) the ST of a local SE overflows or STs of multiple local SEs overflow.

**The ST of the *Master SE* overflows.** The other SEs of the system have *not* overflowed for a specific synchronization variable. Thus, they can still directly buffer this variable in their local STs, and serve their local cores themselves, implementing a hierarchical (two-level) communication with *Master SE*. The *Master SE* receives *global* messages from SEs, and serves a local SE of an NDP unit using *all* bits in the waiting list of the *synchronVar* variable associated with that local SE. Specifically, when it receives a *global* acquire-type message from a local SE, it sets *all* bits in the corresponding waiting list of the *synchronVar* variable. When it receives a *global* release-type message from a local SE, it resets *all* bits in the corresponding waiting list of the *synchronVar* variable.

**The ST of a local SE overflows.** In this scenario, there are local SEs that have overflowed for a specific variable, and local SEs that have *not* overflowed. Without loss of generality, we assume that only one SE of the system has overflowed. **The local SEs that have *not* overflowed** serve their local cores themselves via their STs, implementing a hierarchical (two-level) communication with *Master SE*. When the *Master SE* receives a *global* message from a local SE (that has *not* overflowed), it (i) sets (or resets) *all bits* in the waiting list of the *synchronVar* variable associated with that SE, and (ii) responds with a *global* message to the local SE, if needed.

**The overflowed SE** needs to notify the *Master SE* to handle *local* synchronization requests of NDP cores located at *another* NDP unit via main memory. We design *overflow* message opcodes (Table 4.3) to be sent from the local overflowed SE to the *Master SE* and back. The overflowed SE re-directs all messages (sent from its local NDP cores) for a specific variable to the *Master SE* using the *overflow* message opcodes, and both the overflowed SE and the *Master SE* increase their corresponding indexing counters to indicate that this variable is currently serviced via memory. When the *Master SE* receives an *overflow* message, it (i) sets (or resets) in the waiting list (associated with the overflowed SE) of the *synchronVar* variable, the bit that corresponds to the *local ID* of the NDP core within the NDP unit, (ii) sets (or resets) in the *OverflowInfo* field of the *synchronVar* variable the bit that corresponds to the *global ID* of the overflowed SE to keep track of *which* SE (or SEs) of the system has overflowed, and (iii) responds with an *overflow* message to that SE, if needed. The *local ID* of the NDP core, and the *global ID* of the overflowed SE are encoded in the *CoreID* field of the message (Figure 4.5). When all bits in the waiting lists of the *synchronVar* variable become zero (upon receiving a release-type message), the *Master SE* decrements the corresponding indexing counter. Then, it sends a *decrease\_index\_counter* message (Table 4.3) to the overflowed SE (based on the set bit that is tracked in the *OverflowInfo* field), which decrements its corresponding indexing counter.

#### 4.4.4 SynCron Enhancements

##### RMW Operations

It is straightforward to extend *SynCron* to support simple atomic *rmw* operations inside the SE (by adding a lightweight ALU). The *Master SE* could be responsible for executing atomic *rmw* operations on a variable depending on its address. We leave that for future work.

##### Lock Fairness

When local cores of an NDP unit repeatedly request a lock from their local SE, the SE repeatedly grants the lock within its unit, potentially causing unfairness and delay to other NDP units. To prevent this, an extra field of a local grant counter could be added to the ST entry. The counter increases every time the SE grants the lock to a local core. If the counter exceeds a predefined threshold, then when the SE receives a lock release, it transfers the lock to another SE (assuming other SEs request the lock). The host OS or the user could dynamically set this threshold via a dedicated register. We leave the exploration of such fairness mechanisms to future work.

#### 4.4.5 Comparison with Prior Work

*SynCron*'s design shares some of its design concepts with SSB [300], LCU [307], and MiSAR [299]. However, *SynCron* is more general, supporting the four most widely used synchronization primitives, and easy-to-use thanks to its high-level programming interface.

Table 4.4 qualitatively compares *SynCron* with these schemes. SSB and LCU support only lock semantics, thus they introduce two *ISA extensions* for a simple lock. MiSAR introduces seven ISA extensions to support three primitives and handle overflow scenarios. *SynCron* includes two ISA extensions for four *supported primitives*. A *spin-wait approach* performs consecutive synchronization retries, typically incurring high energy consumption. A *direct notification* scheme sends a direct message to only one waiting core when the synchronization variable becomes available, minimizing the traffic involved upon a release operation. SSB, LCU and MiSAR are tailored for *uniform* memory systems. In contrast, *SynCron* is the *only* hardware synchronization mechanism that targets NDP systems as well as *non-uniform* memory systems.

SSB and LCU handle *overflow* in hardware synchronization resources using a pre-allocated table in main memory, and if it overflows, they switch to software exception handlers (handled by the programmer), which typically incur large overheads (due to OS intervention) when overflows happen at a non-negligible frequency. To avoid falling back to main memory, which has high latency, and using expensive software exception handlers, MiSAR requires the programmer to handle overflow scenarios using alternative software synchronization libraries (e.g., pthread library provided by the OS). This approach can provide performance benefits in CPU systems, since alternative synchronization solutions can exploit low-cost accesses to caches and hardware cache coherence. However, in NDP systems alternative solutions would by default use main memory due to the absence of shared caches and hardware cache coherence support. Moreover, when overflow occurs, MiSAR's accelerator sends abort messages to all participating CPU cores notifying them to use the alternative solution, and when the cores finish synchronizing via the alternative solution, they notify MiSAR's accelerator to switch back to hardware synchronization. This scheme introduces additional hardware/ISA complexity, and communication between the cores and the accelerator, thus incurring high network traffic and communication costs, as we show in Section 4.6.7. In contrast, *SynCron* directly falls back to memory via a fully-integrated hardware-only overflow scheme, which provides graceful performance degradation (Section 4.6.7), and is completely transparent to the programmer: programmers *only* use *SynCron*'s high-level API, similarly to how software libraries are in charge of synchronization.

#### 4.4.6 Use of *SynCron* in Conventional Systems

The baseline NDP architecture [32, 34, 35, 212, 367] we assume in this work shares key design principles with conventional NUMA systems. However, unlike NDP systems, NUMA CPU systems (i) have a shared level of cache (within a NUMA socket and/or across NUMA sockets), (ii) run multiple multi-threaded applications, i.e., a high number of software threads executed in hardware thread contexts, and (iii) the OS migrates software threads between hardware thread contexts to improve system

	SSB [300]	LCU [307]	MiSAR [299]	<b>SynCron</b>
Supported Primitives	1	1	3	<b>4</b>
ISA Extensions	2	2	7	<b>2</b>
Spin-Wait Approach	yes	yes	no	<b>no</b>
Direct Notification	no	yes	yes	<b>yes</b>
Target System	uniform	uniform	uniform	<b>non-uniform</b>
Overflow Management	partially integrated	partially integrated	handled by programmer	<b>fully integrated</b>

Table 4.4: Comparison of *SynCron* with prior mechanisms.

performance. Therefore, although *SynCron* could be implemented in such commodity systems, our proposed hardware design would need extensions. First, *SynCron* could exploit the low-cost accesses to *shared* caches in conventional CPUs, e.g., including an additional level in *SynCron*’s hierarchical design to use the shared cache for efficient synchronization within a NUMA socket, and/or handling overflow scenarios by falling back to the low-latency cache instead of main memory. Second, *SynCron* needs to support use cases (ii) and (iii) listed above in such systems, i.e., including larger STs and waiting lists to satisfy the needs of multiple multithreaded applications, handling the OS thread migration scenarios across hardware thread contexts, and handling multiple synchronization requests sent from different software threads with the same hardware ID to SEs, when different software threads are executed on the same hardware thread context. We leave the optimization of *SynCron*’s design for conventional systems to future work.

## 4.5 Methodology

**Simulation Methodology.** We use an in-house simulator that integrates ZSim [516] and Ramulator [469]. We model 4 NDP units (Table 4.5), each with 16 in-order cores. The cores issue a memory operation after the previous one has completed, i.e., there are no overlapping operations issued by the same core. Any write operation is completed (and the latency is accounted for in our simulations) before executing the next instruction. To ensure memory consistency, compiler support [517] guarantees that there is no reordering around the *sync* instructions and a read is inserted after a write inside a critical section.

We evaluate three NDP configurations for different memory technologies, namely 2D, 2.5D, 3D NDP. The 2D NDP configuration uses a DDR4 memory model and resembles recent 2D NDP systems [157, 162, 338, 401]. In the 2.5D NDP configuration, each compute die of NDP units (16 NDP cores) is connected to an HBM stack via an interposer, similar to current GPUs [525, 526] and FPGAs [477, 527]. For the 3D NDP configuration, we use the HMC memory model, where the compute die of the NDP unit is located in the logic layer of the memory stack, as in prior works [21, 32, 34, 35]. Due to space limitations, we present detailed evaluation results for the 2.5D NDP configuration, and provide a sensitivity study for the different NDP configurations in Section 4.6.5.

<b>NDP Cores</b>	16 in-order cores @2.5 GHz per NDP unit
<b>L1 Data + Inst. Cache</b>	private, 16KB, 2-way, 4-cycle; 64 B line; 23/47 pJ per hit/miss [518]
<b>NDP Unit Local Network</b>	buffered crossbar network with packet flow control; 1-cycle arbiter; 1-cycle per hop [519]; 0.4 pJ/bit per hop [520]; M/D/1 model [521] for queueing latency;
<b>DRAM HBM</b>	4 stacks; 4GB HBM 1.0 [471, 472]; 500MHz with 8 channels; nRCDR/nRCDW/nRAS/nWR 7/6/17/8 ns [469, 522]; 7 pJ/bit [523]
<b>DRAM HMC</b>	4 stacks; 4GB HMC 2.1; 1250MHz; 32 vaults per stack; nRCD/nRAS/nWR 17/34/19 ns [469, 522]
<b>DRAM DDR4</b>	4 DIMMs; 4GB each DIMM DDR4 2400MHz; nRCD/nRAS/nWR 16/39/18 ns [469, 522]
<b>Interconnection Links Across NDP Units</b>	12.8GB/s per direction; 40 ns per cache line; 20-cycle; 4 pJ/bit
<b>Synchronization Engine</b>	SPU @1GHz clock frequency [524]; $8 \times$ 64-bit registers; buffer: 280B; ST: 1192B, 64 entries, 1-cycle [518]; indexing counters: 2304B, 256 entries (8 LSB of the address), 2-cycle [518]

Table 4.5: Configuration of our simulated system.

We model a crossbar network within each NDP unit, simulating queuing latency using the M/D/1 model [521]. We count in ZSim-Ramulator all events for caches, i.e., number of hits/misses, network, i.e., number of bits transferred inside/across NDP units, and memory, i.e., number of total memory accesses, and use CACTI [518] and parameters reported in prior works [367, 520, 523] to calculate energy. To estimate the latency in SE, we use CACTI for ST and indexing counters, and Aladdin [524] for the SPU with 1GHz at 40nm. Each message is served in 12 cycles, corresponding to the message (`barrier_depart_global`) that takes the longest time.

**Workloads.** We evaluate workloads with both (i) coarse-grained synchronization, i.e., including only a few synchronization variables to protect shared data, leading to cores highly contending for them (*high-contention*), and (ii) fine-grained synchronization, i.e., including a large number of synchronization variables, each of them protecting a small granularity of shared data, leading to cores not frequently contending for the same variables at the same time (*low-contention*). We use the term *synchronization intensity* to refer to the ratio of synchronization operations over other computation in the workload. As this ratio increases, synchronization latency affects the total execution time of the workload more.

We study three classes of applications (Table 4.6), all well suited for NDP. First, we evaluate pointer-chasing workloads, i.e., lock-based concurrent data structures from the ASCYLIB library [528], used as key-value sets. In ASCYLIB’s Binary Search Tree (BST) [529], the lock memory requests are only 0.1% of the total memory requests, so we also evaluate an external fine-grained locking BST from [334]. Data structures are initialized with a fixed size and statically partitioned across NDP units, except for BSTs, which are distributed randomly. In these benchmarks, each core performs a fixed number of operations. We use lookup operations for data structures that support it, deletion for the rest, and push and pop operations for stack and queue. Second, we evaluate graph applications with fine-grained synchronization from Crono [530, 531] (push version), where the output array has

read-write data. All real-world graphs [532] used are undirected and statically partitioned across NDP units, where the vertex data is equally distributed across cores. Third, we evaluate time series analysis [533], using SCRIMP, and *real* data sets from Matrix Profile [534]. We replicate the input data in each NDP unit and partition the output array (read-write data) across NDP units.

Data Structure	Configuration
Stack [528]	100K - 100% push
Queue [528, 535]	100K - 100% pop
Array Map [459, 528]	10 - 100% lookup
Priority Queue [528, 536, 537]	20K - 100% deleteMin
Skip List [528, 536]	5K - 100% deletion
Hash Table [487, 528]	1K - 100% lookup
Linked List [487, 528]	20K - 100% lookup
Binary Search Tree Fine-Grained (BST_FG) [334]	20K - 100% lookup
Binary Search Tree Drachsler (BST_Drachsler) [528, 529]	10K - 100% deletion

Real Application	Locks	Barriers	Real Application	Input Data Set
Breadth First Search ( <b>bfs</b> ) [530]	✓	✓	<b>bfs, cc, sssp, pr, tf, tc</b>	wikipedia
Connected Components ( <b>cc</b> ) [530]	✓	✓		-20051105 ( <b>wk</b> )
Single Source Shortest Paths ( <b>sssp</b> ) [530]	✓	✓		soc-LiveJournal1 ( <b>sl</b> )
Pagerank ( <b>pr</b> ) [530]	✓	✓		sx-stackoverflow ( <b>sx</b> )
Teenage Followers ( <b>tf</b> ) [531]	✓	-		com-Orkut ( <b>co</b> )
Triangle Counting ( <b>tc</b> ) [530]	✓	✓		air quality ( <b>air</b> )
Time Series Analysis ( <b>ts</b> ) [534]	✓	✓	<b>ts</b>	energy consumption ( <b>pow</b> )

Table 4.6: Summary of all workloads used in our evaluation.

**Comparison Points.** We compare *SynCron* with three schemes: (i) *Central*: a message-passing scheme that supports all primitives by extending the barrier primitive of Tesseract [32], i.e., one dedicated NDP core in the entire NDP system acts as server and coordinates synchronization among all NDP cores of the system by issuing memory requests to synchronization variables via its memory hierarchy, while the remaining client cores communicate with it via hardware message-passing; (ii) *Hier*: a hierarchical message-passing scheme that supports all primitives, similar to the barrier primitive of [212] (or hierarchical lock of [512]), i.e., one NDP core per NDP unit acts as server and coordinates synchronization by issuing memory requests to synchronization variables via its memory hierarchy (including caches), and communicates with other servers and local client cores (located at the same NDP unit with it) via hardware message-passing; (iii) *Ideal*: an ideal scheme with zero performance overhead for synchronization. In our evaluation, each NDP core runs one thread. For fair comparison, we use the same number of client cores, i.e., 15 per NDP unit, that execute the main workload for all schemes. For synchronization, we add one server core for the entire system in *Central*, one server core per NDP unit for *Hier*, and one SE per NDP unit for *SynCron*. For *SynCron*, we disable one core per NDP unit to match the same number of client cores as the previous schemes. Maintaining the same thread-level parallelism for executing the main kernel is consistent with prior works on message-passing synchronization [299, 512].

## 4.6 Evaluation

### 4.6.1 Performance

#### Synchronization Primitives

Figure 4.10 evaluates all supported primitives using 60 cores, varying the interval (in terms of instructions) between two synchronization points. We devise simple benchmarks, where cores repeatedly request a single synchronization variable. For lock, the critical section is empty, i.e., it does not include any instruction. For semaphore and condition variable, half of the cores execute `sem_wait/cond_wait`, while the rest execute `sem_post/cond_signal`, respectively. As the interval between synchronization points becomes smaller, *SynCron*'s performance benefit increases. For an interval of 200 instructions, *SynCron* outperforms *Central* and *Hier* by  $3.05\times$  and  $1.40\times$  respectively, averaged across all primitives. *SynCron* outperforms *Hier* due to directly buffering synchronization variables in low-latency STs, and achieves the highest benefits in the condition variable primitive (by  $1.61\times$ ), since this benchmark has higher synchronization intensity compared to the rest: cores coordinate for both the condition variable and the lock associated with it. When the interval between synchronization operations becomes larger, synchronization requests become less dominant in the main workload, and thus all schemes perform similarly. Overall, *SynCron* outperforms prior schemes for all different synchronization primitives.

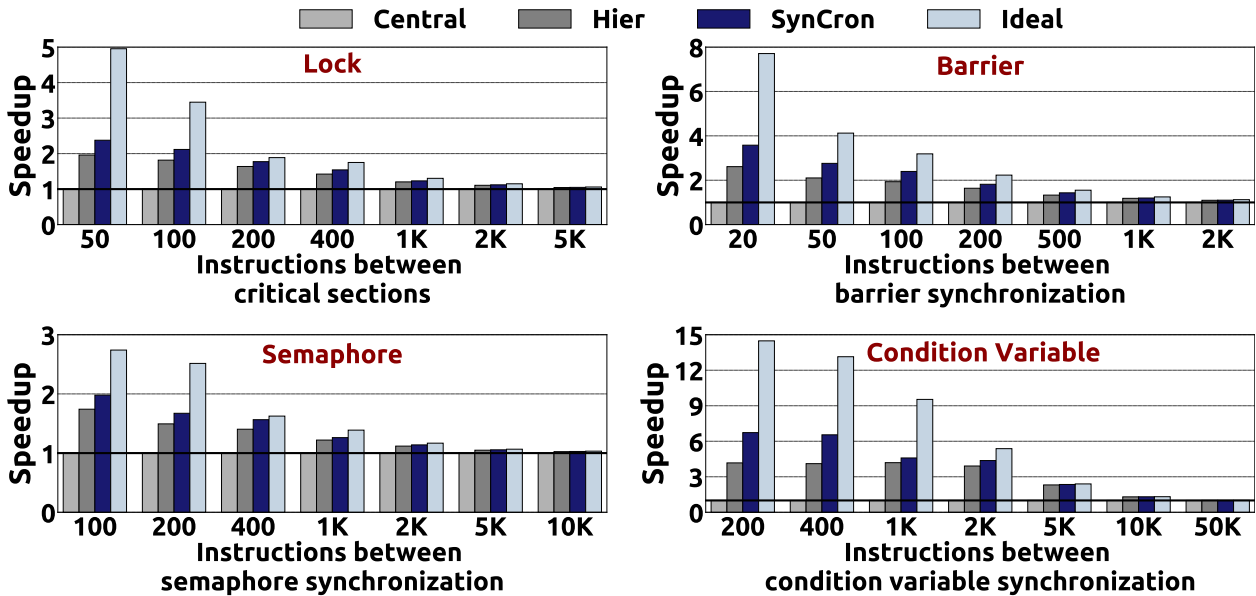


Figure 4.10: Speedup of different synchronization primitives.

#### Pointer-Chasing Data Structures

Figure 4.11 shows the throughput for all schemes in pointer-chasing varying the NDP cores in steps of 15, each time adding one NDP unit.

We observe four different patterns. First, *stack*, *queue*, *array map*, and *priority queue* incur high contention, as all cores heavily contend for a few variables. *Array map* has the lowest scalability due to a larger critical section. In high-contention scenarios, hierarchical schemes (*Hier*, *SynCron*)



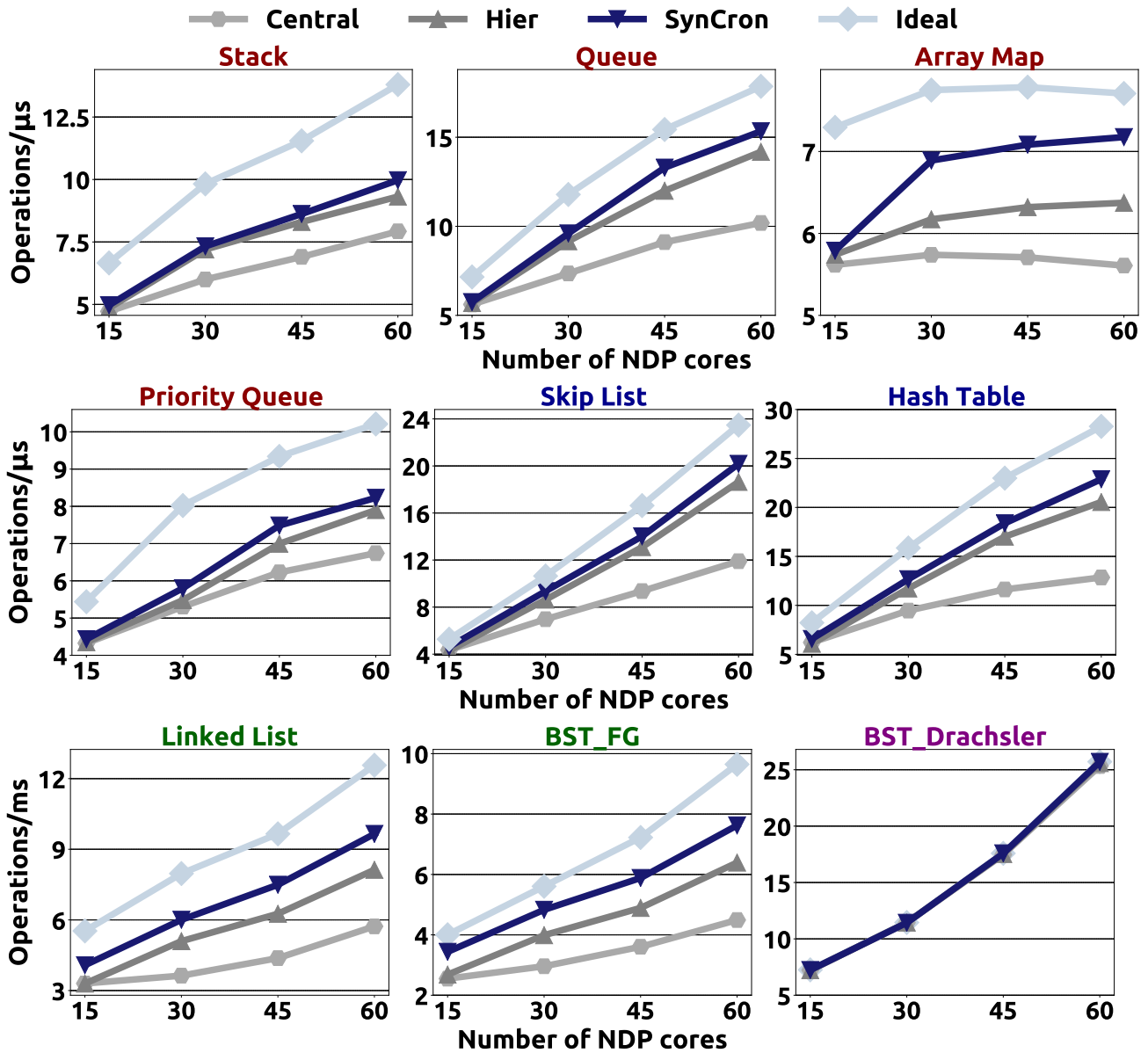


Figure 4.11: Throughput of pointer-chasing using data structures.

perform better by reducing the expensive traffic across NDP units. *SynCron* outperforms *Hier*, since the latency cost of using SEs that update small STs is lower than using NDP cores as servers that update larger caches. Second, *skip list* and *hash table* incur medium contention, as different cores may work on different parts of the data structure. For these data structures, hierarchical schemes perform better, as they minimize the expensive traffic, and multiple server cores concurrently serve requests to their local memory. *SynCron* retains most of the performance benefits of *Ideal*, incurring only 19.9% overhead with 60 cores, and outperforms *Hier* by 9.8%. Third, *linked list* and *BST\_FG* exhibit low contention and high synchronization demand, as each core requests multiple locks concurrently. These data structures cause higher synchronization-related traffic inside the network compared to *skip list* and *hash table*, and thus *SynCron* further outperforms *Hier* by  $1.19\times$  due to directly buffering synchronization variables in STs. Fourth, in *BST\_Drachsler* lock requests constitute only 0.1% of the total requests, and all schemes perform similarly. Overall, we conclude that *SynCron* achieves higher throughput than prior mechanisms under different scenarios with diverse conditions.

## Real Applications

Figure 4.12 shows the performance of all schemes with real applications using all NDP units, normalized to *Central*. Averaged across 26 application-input combinations, *SynCron* outperforms *Central* by  $1.47\times$  and *Hier* by  $1.23\times$ , and performs within 9.5% of *Ideal*.

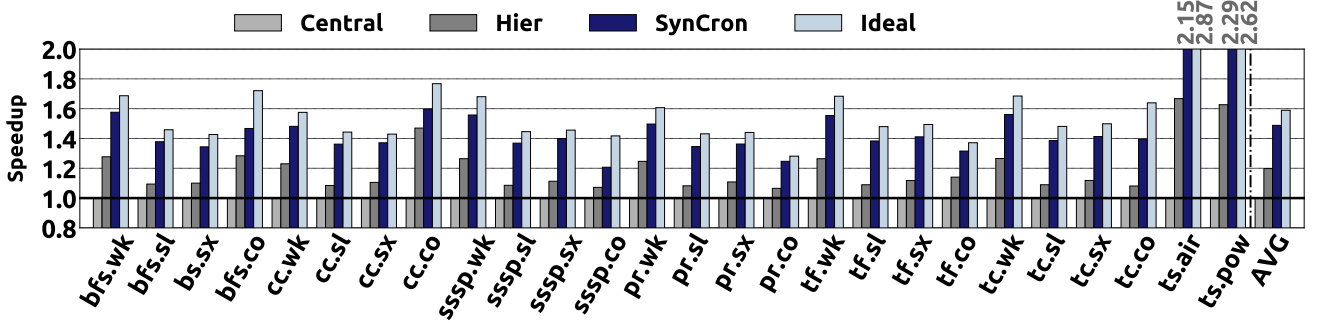


Figure 4.12: Speedup in real applications normalized to *Central*.

Our real applications exhibit low contention, as two cores rarely contend for the same synchronization variable, and high synchronization demand, as several synchronization variables are active during execution. We observe that *Hier* and *SynCron* increase parallelism, because the per-NDP-unit servers service different synchronization requests concurrently, and avoid remote synchronization messages across NDP units. Even though *Hier* performs  $1.19\times$  better than *Central*, on average, its performance is still  $1.33\times$  worse than *Ideal*. *SynCron* provides most of the performance benefits of *Ideal* (with only 9.5% overhead on average), and outperforms *Hier* due to directly buffering the synchronization variables in STs, thereby completely avoiding the memory accesses for synchronization requests. Specifically, we find that *time series analysis* has high synchronization intensity, since the ratio of synchronization over other computation of the workload is higher compared to graph workloads. For this application, *Hier* and *SynCron* outperform *Central* by  $1.64\times$  and  $2.22\times$ , as they serve multiple synchronization requests concurrently. *SynCron* further outperforms *Hier* by  $1.35\times$  due to directly buffering the synchronization variables in STs. We conclude that *SynCron* performs best across *all* real application-input combinations and approaches the *Ideal* scheme with no synchronization overhead.

**Scalability.** Figure 4.13 shows the scalability of real applications using *SynCron* from 1 to 4 NDP units. Due to space limitations, we present a subset of our workloads, but we report average values for all 26 application-input combinations. This also applies for all figures presented henceforth. Across all workloads, *SynCron* enables performance scaling by at least  $1.32\times$ , on average  $2.03\times$ , and up to  $3.03\times$ , when using 4 NDP units (60 NDP cores) over 1 NDP unit (15 NDP cores).

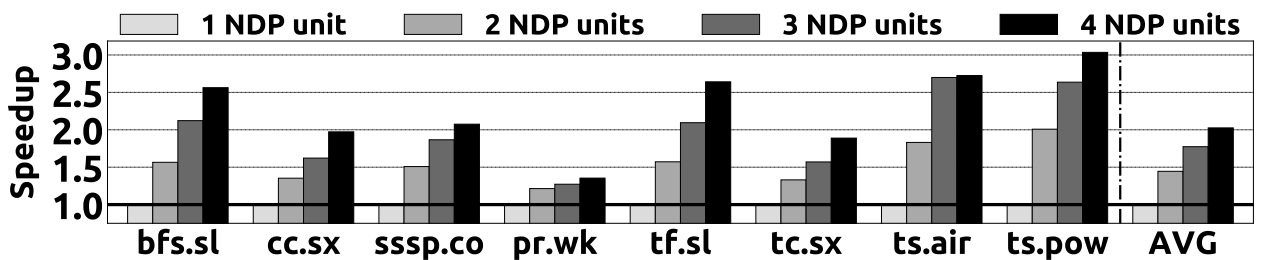


Figure 4.13: Scalability of real applications using *SynCron*.

### 4.6.2 Energy Consumption

Figure 4.14 shows the energy breakdown for cache, network, and memory in our real applications when using all cores. *SynCron* reduces the network and memory energy thanks to its hierarchical design and direct buffering. On average, *SynCron* reduces energy consumption by  $2.22\times$  over *Central* and  $1.94\times$  over *Hier*, and incurs only 6.2% energy overhead over *Ideal*.

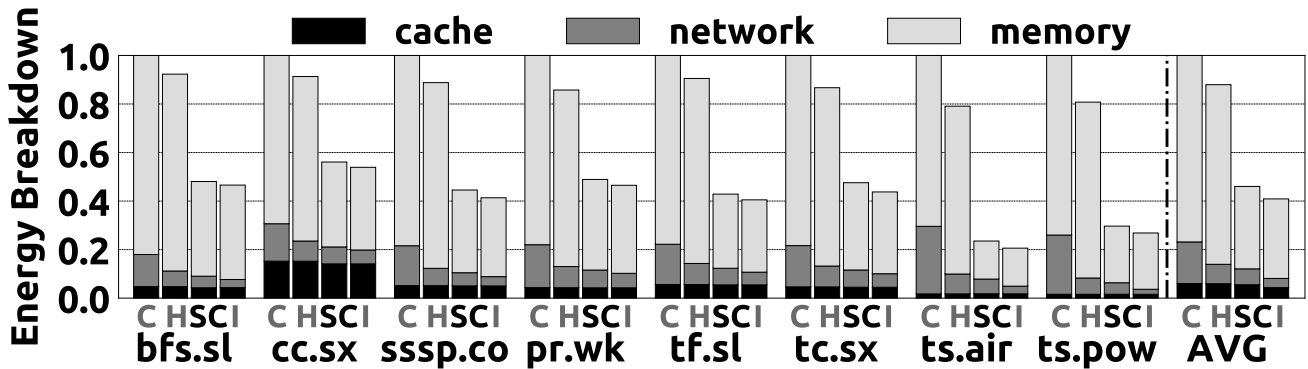


Figure 4.14: Energy breakdown in real applications for C: *Central*, H: *Hier*, SC: *SynCron* and I: *Ideal*.

We observe that 1) cache energy consumption constitutes a small portion of the total energy, since these applications have irregular access patterns. NDP cores that act as servers for *Central* and *Hier* increase the cache energy only by 5.1% and 4.8% over *Ideal*. 2) *Central* generates a larger amount of expensive traffic across NDP units compared to hierarchical schemes, resulting in  $2.68\times$  higher network energy over *SynCron*. *SynCron* also has less network energy ( $1.21\times$ ) than *Hier*, because it avoids transferring synchronization variables from memory to SEs due to directly buffering them. 3) *Hier* and *Central* have approximately the same memory energy consumption, because they issue a similar number of requests to memory. In contrast, *SynCron*'s memory energy consumption is similar to that of *Ideal*. We note that *SynCron* provides *higher* energy reductions in applications with high synchronization intensity, such as time series analysis, since it avoids a *higher* number of memory accesses for synchronization due to its direct buffering capability.

### 4.6.3 Data Movement

Figure 4.15 shows normalized data movement, i.e., bytes transferred between NDP cores and memory, for all schemes using four NDP units. *SynCron* reduces data movement across all workloads by  $2.08\times$  and  $2.04\times$  over *Central* and *Hier*, respectively, on average, and incurs only 13.8% more data movement than *Ideal*. *Central* generates high data movement across NDP units, particularly when running time series analysis that has high synchronization intensity. *Hier* reduces the traffic across NDP units; however, it may increase the traffic inside an NDP unit, occasionally leading to slightly higher total data movement (e.g., *ts.air*). This is because when an NDP core requests a synchronization variable that is physically located in another NDP unit, it first sends a message inside the NDP unit to its local server, which in turns sends a message to the global server. In contrast, *SynCron* reduces the traffic inside an NDP unit due to directly buffering synchronization variables, and across NDP units due to its hierarchical design.

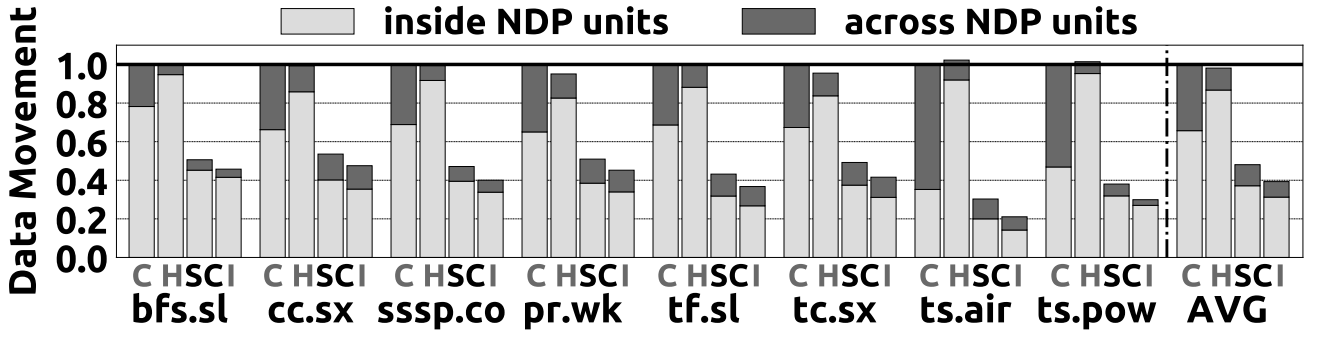


Figure 4.15: Data movement in real applications for C: *Central*, H: *Hier*, SC: *SynCron* and I: *Ideal*.

#### 4.6.4 Non-Uniformity of NDP Systems

##### High Contention

Hierarchical schemes provide high benefit under high contention, as they prioritize local requests inside each NDP unit. We study their performance benefit in stack and priority queue (Figure 4.16) when varying the transfer latency of the interconnection links used across four NDP units. *Central* is significantly affected by the interconnect latency across NDP units, as it is oblivious to the non-uniform nature of the NDP system. Observing *Ideal*, which reflects the actual behavior of the main workload, we notice that after a certain point (vertical line), the cost of remote memory accesses across NDP units become high enough to dominate performance. *SynCron* and *Hier* tend to follow the actual behavior of the workload, as local synchronization messages within NDP units are much less expensive than remote messages of *Central*. *SynCron* outperforms *Hier* by  $1.06\times$  and  $1.04\times$  for stack and priority queue. We conclude that *SynCron* is the best at hiding the latency of slow links across NDP units.

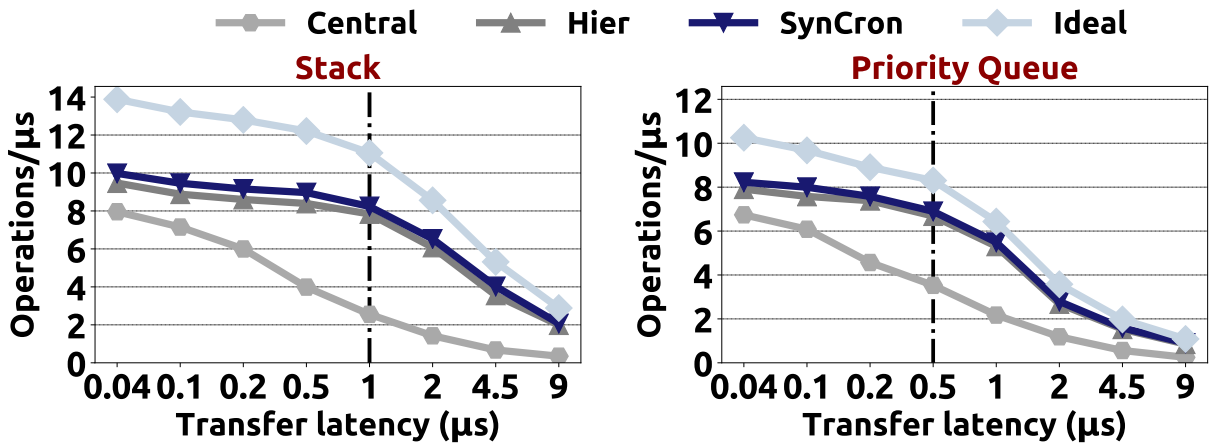


Figure 4.16: Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units.

##### Low Contention

We also study the effect of interconnection links used across the NDP units in a low-contention graph application (Figure 4.17). Observing *Ideal*, with 500 ns transfer latency per cache line, we note that the workload experiences  $2.46\times$  slowdown over the default latency of 40 ns, as 24.1% of its memory

accesses are to remote NDP units. As the transfer latency increases, *Central* incurs significant slowdown over *Ideal*, since all NDP cores of the system communicate with one single server, generating expensive traffic across NDP units. In contrast, the slowdown of hierarchical schemes over *Ideal* is smaller, as these schemes generate less remote traffic by distributing the synchronization requests across multiple local servers. *SynCron* outperforms *Hier* due to its direct buffering capabilities. Overall, *SynCron* outperforms prior high-performance schemes even when the network delay across NDP units is large.

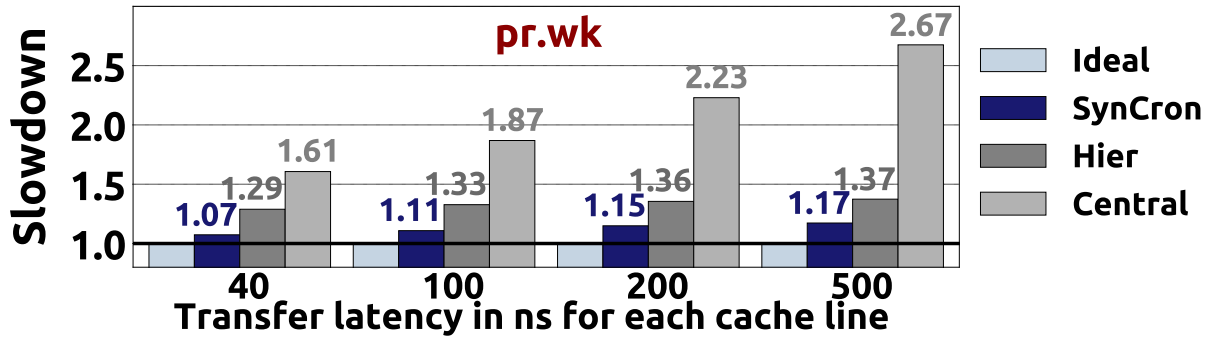


Figure 4.17: Performance sensitivity to the transfer latency of the interconnection links used to connect the NDP units. All data is normalized to *Ideal* (lower is better).

#### 4.6.5 Memory Technologies

We study three memory technologies, which provide different memory access latencies and bandwidth. We evaluate (i) 2.5D NDP using HBM, (ii) 3D NDP using HMC, and (iii) 2D NDP using DDR4. Figure 4.18 shows the performance of all schemes normalized to *Central* of each memory. The reported values show the speedup of *SynCron* over *Central* and *Hier*. *SynCron*'s benefit is independent of the memory used: its performance versus *Ideal* only slightly varies ( $\pm 1.4\%$ ) across different memory technologies, since STs never overflow. Moreover, *SynCron*'s performance improvement over prior schemes increases as the memory access latency becomes higher thanks to direct buffering, which avoids expensive memory accesses for synchronization. For example, in *ts.pow*, *SynCron* outperforms *Hier* by  $1.41\times$  and  $2.49\times$  with HBM and DDR4, respectively, as the latter incurs higher access latency. Overall, *SynCron* is orthogonal to the memory technology used.

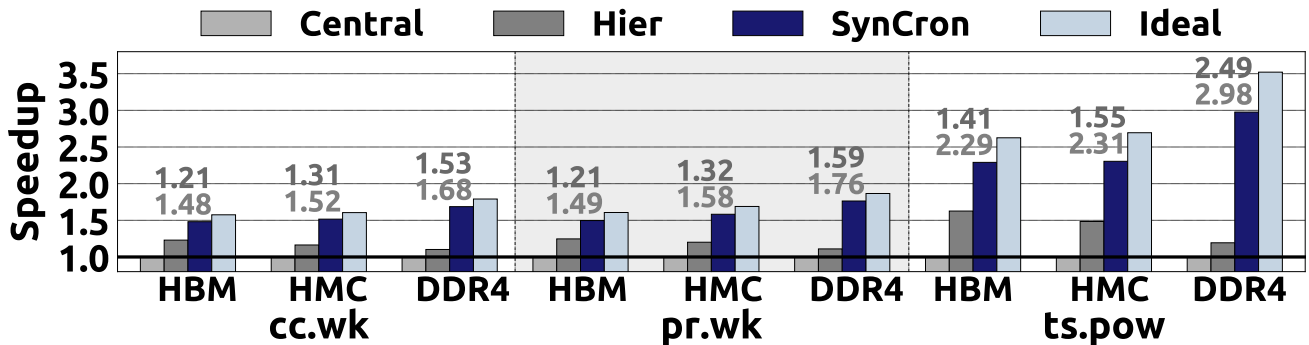


Figure 4.18: Speedup with different memory technologies.

#### 4.6.6 Effect of Data Placement

Figure 4.19 evaluates the effect of better data placement on *SynCron*'s benefits. We use Metis [538] to obtain a 4-way graph partitioning to minimize the crossing edges between the 4 NDP units. All data values are normalized to *Central* without Metis. For *SynCron*, we define ST occupancy as the average fraction of ST entries that are occupied in each cycle.

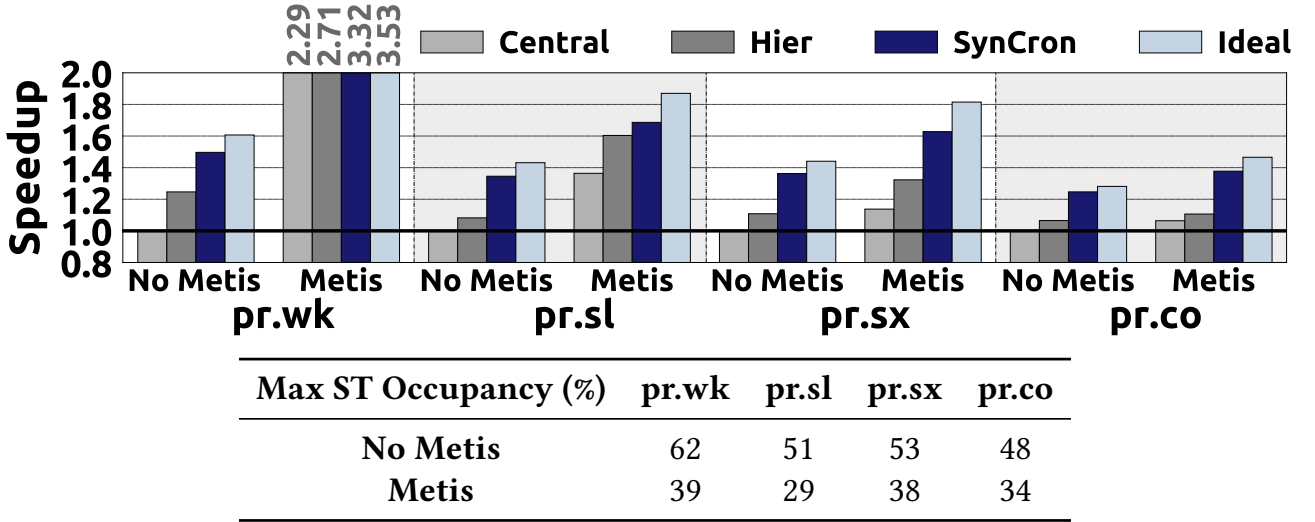


Figure 4.19: Performance sensitivity to a better graph partitioning and maximum ST occupancy of *SynCron*.

We make three observations. First, *Ideal*, which reflects the actual behavior of the main kernel (i.e., with zero synchronization overhead), improves performance by  $1.47\times$  across the four graphs. Second, with a better graph partitioning, *SynCron* still outperforms both *Central* and *Hier*. Third, we find that ST occupancy is lower with a better graph partitioning. When a local SE receives a request for a synchronization variable of another NDP unit, *both* the local SE and the *Master SE* reserve a new entry in their STs. With a better graph partitioning, NDP cores send requests to their local SE, which is also the *Master SE* for the requested variable. Thus, *only one* SE of the system reserves a new entry, resulting in a lower ST occupancy. We conclude that, with better data placement *SynCron* still performs the best while achieving even lower ST occupancy.

#### 4.6.7 *SynCron*'s Design Choices

##### Hierarchical Design

To demonstrate the effectiveness of *SynCron*'s hierarchical design in non-uniform NDP systems, we compare it with *SynCron*'s *flat* variant. Each core in *flat* directly sends all its synchronization requests to the *Master SE* of each variable. In contrast, each core in *SynCron* sends all its synchronization requests to the local SE. If the local SE is *not* the *Master SE* for the requested variable, the local SE sends a message across NDP units to the *Master SE*.

We evaluate three synchronization scenarios: (i) low-contention and synchronization non-intensive (e.g., graph applications), (ii) low-contention and synchronization-intensive (e.g., time series analysis), and (iii) high-contention (e.g., queue data structure).

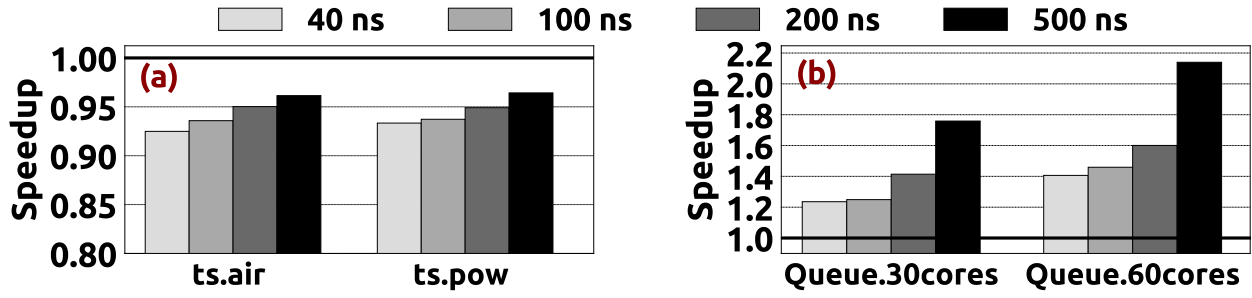


Figure 4.21: Speedup of *SynCron* normalized to *flat*, as we vary the transfer latency of the interconnection links used to connect NDP units, under (a) a low-contention and synchronization-intensive scenario using 4 NDP units, and (b) a high-contention scenario using 2 and 4 NDP units.

**Low-contention and synchronization non-intensive.** Figure 4.20 evaluates this scenario using several graph processing workloads with 40 ns link latency between NDP units. *SynCron* is 1.1% worse than *flat*, on average. We conclude that *SynCron* performs only *slightly* worse than *flat* for low-contention and synchronization non-intensive scenarios.

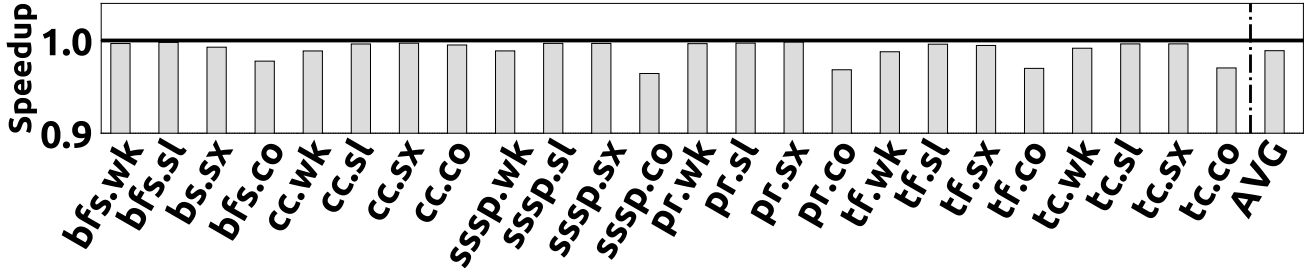


Figure 4.20: Speedup of *SynCron* normalized to *flat* with 40 ns link latency between NDP units, under a low-contention and synchronization non-intensive scenario.

**Low-contention and synchronization-intensive.** Figure 4.21a evaluates this scenario using time series analysis with four different link latency values between NDP units. *SynCron* performs 7.3% worse than *flat* with a 40 ns inter-NDP-unit latency. With a 500 ns inter-NDP-unit latency, *SynCron* performs *only* 3.6% worse than *flat*, since remote traffic has a larger impact on the total execution time. We conclude that *SynCron* performs modestly worse than *flat*, and *SynCron*'s slowdown decreases as non-uniformity, i.e., the latency between NDP units, increases.

**High-contention.** Figure 4.21b evaluates this scenario using a queue data structure with four different link latency values between NDP units, for 30 and 60 NDP cores. *SynCron* with 30 NDP cores outperforms *flat* from  $1.23\times$  to  $1.76\times$ , as the inter-NDP-unit latency increases from 40 ns to 500 ns (i.e., with increasing non-uniformity in the system). In a scenario with high non-uniformity in the system and large number of contended cores, e.g., using a 500 ns inter-NDP-unit latency and 60 NDP cores, *SynCron*'s benefit increases to a  $2.14\times$  speedup over *flat*. We conclude that *SynCron* performs *significantly* better than *flat* under high-contention.

Overall, we conclude that in *non-uniform, distributed* NDP systems, *only a hierarchical* hardware synchronization design can achieve high performance under *all* various scenarios.

## ST Size

We show the effectiveness of the proposed 64-entry ST (per NDP unit) using real applications. Table 4.7 shows the measured occupancy across all STs. Figure 4.22 shows the performance sensitivity to ST size. In graph applications, the average ST occupancy is low (2.8%), and the 64-entry ST never overflows: maximum occupancy is 63% (*cc.wk*). In contrast, time series analysis has higher ST occupancy (reaching up to 89% in *ts.pow*) due to the high synchronization intensity, but there are no ST overflows. Even a 48-entry ST overflows for only 0.01% of synchronization requests, and incurs 2.1% slowdown over a 64-entry ST. We conclude that the proposed 64-entry ST meets the needs of applications that have high synchronization intensity.

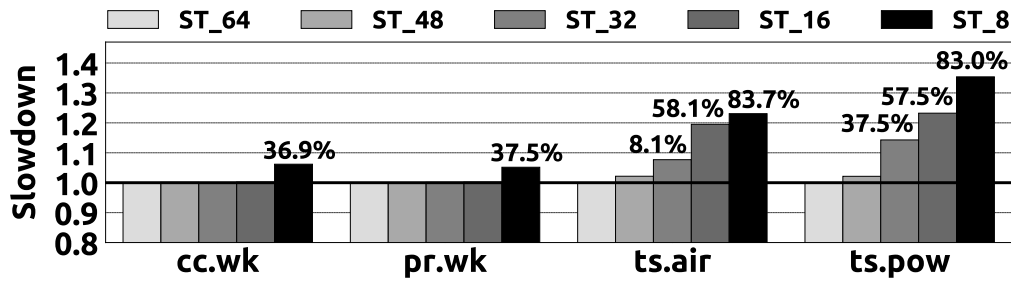


Figure 4.22: Slowdown with varying ST size (normalized to 64-entry ST). Numbers on top of bars show the percentage of overflowed requests.

## Overflow Management

The linked list and BST\_FG data structures are the *only* cases where the proposed 64-entry ST overflows, when using 60 cores, for 3.1% and 30.5% of the requests, respectively. This is because each core requests at least two locks *at the same time* during the execution. Note that these synthetic benchmarks represent extreme scenarios, where all cores repeatedly perform key-value operations.

ST Occupancy	Max (%)	Avg (%)	ST Occupancy	Max (%)	Avg (%)
bfs.wk	51	1.33	pr.sl	51	2.27
bfs.sl	59	1.49	pr.sx	53	2.46
bfs.sx	51	3.24	pr.co	48	4.72
bfs.co	55	6.09	tf.wk	62	1.44
cc.wk	63	1.27	tf.sl	53	2.21
cc.sl	61	2.16	tf.sx	50	2.99
cc.sx	48	2.43	tf.co	48	4.61
cc.co	46	4.53	tc.wk	62	1.26
sssp.wk	62	1.18	tc.sl	48	2.08
sssp.sl	54	2.08	tc.sx	50	2.77
sssp.sx	50	2.20	tc.co	51	4.52
sssp.co	48	5.23	ts.air	84	44.20
pr.wk	62	4.27	ts.pow	89	43.51

Table 4.7: ST occupancy in real applications.



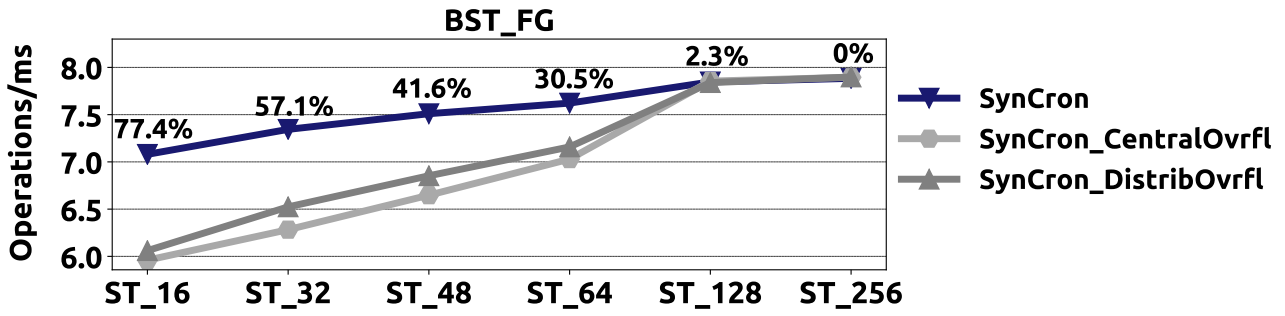


Figure 4.23: Throughput achieved by BST\_FG using different overflow schemes and varying the ST size. The reported numbers show to the percentage of overflowed requests.

Figure 4.23 compares BST\_FG’s performance with *SynCron*’s integrated overflow scheme versus with a non-integrated scheme as in MiSAR. When overflow occurs, MiSAR’s accelerator aborts all participating cores notifying them to use an alternative synchronization library, and when the cores finish synchronizing via an alternative solution, they notify MiSAR’s accelerator to switch back to hardware synchronization. We adapt this scheme to *SynCron* for comparison purposes: when an ST overflows, SEs send abort messages to NDP cores with a hierarchical protocol, notifying them to use an alternative synchronization solution, and after finishing synchronization they notify SEs to decrease their indexing counters and switch to hardware. We evaluate two alternative solutions: (i) *SynCron\_CentralOvrfl*, where one dedicated NDP core handles all synchronization variables, and (ii) *SynCron\_DistribOvrfl*, where one NDP core per NDP unit handles variables located in the same NDP unit. With 30.5% overflowed requests (i.e., with a 64-entry ST), *SynCron\_CentralOvrfl* and *SynCron\_DistribOvrfl* incur 12.3% and 10.4% performance slowdown compared to with *no* ST overflow, due to high network traffic and communication costs between NDP cores and SEs. In contrast, *SynCron* affects performance by only 3.2% compared to with *no* ST overflow. We conclude that *SynCron*’s integrated hardware-only overflow scheme enables very small performance overhead.

#### 4.6.8 *SynCron*’s Area and Power Overhead

Table 4.8 compares an SE with the ARM Cortex A7 core [539]. We estimate the SPU using Aladdin [524], and the ST and indexing counters using CACTI [518]. We conclude that our proposed hardware unit incurs very modest area and power costs to be integrated into the compute die of an NDP unit.

	SE (Synchronization Engine)	ARM Cortex A7 [539]
Technology	40nm	28nm
Area	SPU: 0.0141mm <sup>2</sup> , ST: 0.0112mm <sup>2</sup>	32KB L1 Cache
	Indexing Counters: 0.0208mm <sup>2</sup>	
	<b>Total:</b> 0.0461mm <sup>2</sup>	<b>Total:</b> 0.45mm <sup>2</sup>
Power	2.7 mW	100mW

Table 4.8: Comparison of SE with a simple general-purpose in-order core, ARM Cortex A7.

## 4.7 Recommendations

This section presents our key takeaways in the form of recommendations for software and hardware designers.

**Recommendation #1.** *Provide hardware synchronization support for NDP architectures.* Figures 4.10, 4.11 and 4.12 demonstrate that *SynCron* significantly outperforms software-based synchronization schemes, e.g., *Central* and *Hier*, across various contention scenarios and workload demands. In addition, Tables 4.7 and 4.8 show that *SynCron* has modest area and power costs for NDP architectures. In contrast to commodity CPU and GPU systems that run multiple software threads executed at each hardware thread context, NDP architectures [21–25, 27–29, 32–37, 158, 159, 208, 212, 213, 216, 265–267, 338, 366–368, 473–477] typically support a only fixed number of hardware thread contexts, and thus in such computing platforms synchronization can be effectively implemented in hardware with low cost. Therefore, we suggest that hardware designers of NDP architectures provide low-cost synchronization mechanisms implemented in *hardware*.

**Recommendation #2.** *Design hierarchical, non-uniform aware synchronization schemes for non-uniform NDP systems.* NDP systems are typically non-uniform, distributed architectures, in which inter-unit communication is more expensive (both in performance and energy) than intra-unit communication [28, 29, 32, 34, 35, 37, 212, 366]. Our evaluations presented in Figures 4.16 and 4.17 show that the hierarchical schemes, i.e., *Hier* and *SynCron*, provide significant performance benefits over *Central*, since *Central* is oblivious to the non-uniform nature of NDP systems. Under high-contention scenarios (Figure 4.16), the hierarchical (non-uniform aware) schemes achieve high system performance by minimizing the expensive traffic across NDP units of the system. Under low-contention scenarios (Figure 4.17), the hierarchical schemes provide high system performance, because they (i) generate less remote traffic by distributing the synchronization requests across multiple local synchronization units, and (ii) increase parallelism, since the per-NDP-unit synchronization units service different synchronization requests concurrently. To this end, we recommend that hardware architects design non-uniform aware synchronization mechanisms for NDP systems.

**Recommendation #3.** *Design effective data placement schemes of the input data and the associated synchronization variables across multiple NDP units of the NDP system.* In many real-world applications (e.g., graph processing applications), the large input data set given (e.g., real-world graphs with a large number of vertices and edges) is shared among multiple cores, and thus a fine-grained synchronization scheme (i.e., including a large number of synchronization variables, each of them protects a small granularity of shared data) is typically used. Figure 4.19 demonstrates that a better graph partitioning in graph processing workloads significantly improves performance of the main kernel and reduces the synchronization costs among NDP cores. Specifically, with a better graph partitioning *SynCron* (i) reduces the remote synchronization messages sent across the NDP units of the system through the expensive interconnection links, and (ii) has lower ST occupancy, thus having lower ST sizes (with lower area and power costs) can be sufficient to meet the synchronization needs of real-world applications without *never* overflowing. Therefore, we suggest that software engineers of real-world applications with fine-grained synchronization schemes design intelligent data

placement schemes of the input data and the associated synchronization variables across multiple NDP units of NDP architectures to achieve high system performance and minimize synchronization costs.

## 4.8 Related Work

To our knowledge, our work is the first one to (i) comprehensively analyze and evaluate synchronization primitives in NDP systems, and (ii) propose an end-to-end hardware-based synchronization mechanism for efficient execution of such primitives. We briefly discuss prior work.

**Synchronization on NDP.** Ahn et al. [32] include a message-passing barrier similar to our *Central* baseline. Gao et al. [212] implement a hierarchical tree-based barrier for HMC [468], where cores first synchronize inside the vault, then across vaults, and finally across HMC stacks. Section 4.6.1 shows that *SynCron* outperforms such schemes. Gao et al. [212] also provide remote atomics at the vault controllers of HMC. However, synchronization using remote atomics creates high global traffic and hotspots [153, 370, 371, 381, 382].

**Synchronization on CPUs.** A range of hardware synchronization mechanisms have been proposed for commodity CPU systems [301–306]. These are not suitable for NDP systems because they either (i) rely on the underlying cache coherence system [302, 306], (ii) are tailored for the 2D-mesh network topology to connect all cores [301, 303], or (iii) use transmission-line technology [304] or on-chip wireless technology [305]. Callbacks [540] includes a directory cache structure close to the LLC of a CPU system built on self-invalidation coherence protocols [309–314]. Although it has low area cost, it would be oblivious to the non-uniformity of NDP, thereby incurring high performance overheads under high contention (Section 4.6.7). Callbacks improves performance of spin-wait in hardware, on top of which high-level primitives (locks/barriers) are implemented in software. In contrast, *SynCron* directly supports high-level primitives in hardware, and is tailored to all salient characteristics of NDP systems.

The closest works to ours are SSB [300], LCU [307], and MiSAR [299]. SSB, a shared memory scheme, includes a small buffer attached to each controller of LLC to provide lock semantics for a given data address. LCU, a message-passing scheme, incorporates a control unit into each core and a reservation table into each memory controller to provide reader-writer locks. MiSAR is a message-passing synchronization accelerator distributed at each LLC slice of tile-based many-core chips. These schemes provide efficient synchronization for CPU systems *without* relying on hardware coherence protocols. As shown in Table 4.4, compared to these works, *SynCron* is a more effective, general and easy-to-use solution for NDP systems. These works have two major shortcomings. First, they are designed for *uniform* architectures, and would incur high performance overheads in *non-uniform, distributed* NDP systems under high-contention scenarios, similarly to *flat* in Figure 4.21b. Second, SSB and LCU handle overflow cases using software exception handlers that typically incur large performance overheads, while MiSAR’s overflow scheme would incur high performance degradation due to high network traffic and communication costs between the cores and the synchronization accelerator (Section 4.6.7). In contrast, *SynCron* is a non-uniformity aware, hardware-only, end-to-

end solution designed to handle key characteristics of NDP systems.

**Synchronization on GPUs.** GPUs support remote atomic units at the shared cache and hardware barriers among threads of the same block [541], while inter-block barrier synchronization is inefficiently implemented via the host CPU [541]. The closest work to ours is HQL [370], which modifies the tag arrays of L1 and L2 caches to support the lock primitive. This scheme incurs high area cost [371], and is tailored to the GPU architecture that includes a shared L2 cache, while most NDP systems do *not* have shared caches.

**Synchronization on MPPs.** The Cray T3D/T3E [372, 373], SGI Origin [315], and AMOs [374] include remote atomics at the memory controller, while NYU Ultracomputer [317] provides *fetch&and* remote atomics in each network switch. As discussed in Section 4.2, synchronization via remote atomics incurs high performance overheads due to high global traffic [153, 370, 371, 381]. Cray T3E supports a barrier using physical wires, but it is designed specifically for 3D torus interconnect. Tera MTA [316], HEP [375, 376], J- and M-machines [377, 378], and Alewife [542] provide synchronization using hardware bits (*full/empty* bits) as tags in *each memory word*. This scheme can incur high area cost [307]. QOLB [379] associates one cache line for every lock to track a pointer to the next waiting core, and one cache line for local spinning using bits (*syncbits*). QOLB is built on the underlying cache coherence protocol. Similarly, DASH [380] keeps a queue of waiting cores for a lock in the directory used for coherence to notify caches when the lock is released. CM5 [308] supports remote atomics and a barrier among cores via a dedicated physical control network (organized as a binary tree), which would incur high hardware cost to be supported in NDP systems.

## 4.9 Summary

*SynCron* is the first end-to-end synchronization solution for NDP systems. *SynCron* avoids the need for complex coherence protocols and expensive *rmw* operations, incurs very modest hardware cost, generally supports many synchronization primitives and is easy-to-use. Our evaluations show that it outperforms prior designs under various conditions, providing high performance both under high-contention (due to reduction of expensive traffic across NDP units) and low-contention scenarios (due to direct buffering of synchronization variables and high execution parallelism). We conclude that *SynCron* is an efficient synchronization mechanism for NDP systems, and hope that this work encourages further comprehensive studies of the synchronization problem in heterogeneous systems, including NDP systems.

# CHAPTER 5

## *SparseP*

### 5.1 Overview

Sparse Matrix Vector Multiplication (SpMV) is a fundamental linear algebra kernel for important applications from the scientific computing, machine learning, and graph analytics domains. In commodity systems, it has been repeatedly reported to achieve only a small fraction of the peak performance [18, 103, 111, 120, 132, 146, 291, 294, 320–323] due to its algorithmic nature, the employed compressed matrix storage format, and the sparsity pattern of the input matrix. SpMV performs indirect memory references as a result of storing the matrix in a compressed format, and irregular memory accesses to the input vector due to sparsity. The matrices involved are very sparse, i.e., the vast majority of elements are zeros [18, 103, 150, 286, 289–293]. For example, the matrices that represent Facebook’s and YouTube’s network connectivity contain 0.0003% [286, 289] and 2.31% [286, 290] non-zero elements, respectively. Therefore, in processor-centric systems, SpMV is a memory-bandwidth-bound kernel for the majority of real sparse matrices, and is bottlenecked by data movement between memory and processors [17, 18, 42, 44, 103, 111, 120, 146, 161, 162, 272, 291, 294, 320–323, 383, 384].

One promising way to alleviate the data movement bottleneck is the Processing-In-Memory (PIM) paradigm [5, 21, 22, 25, 27, 32, 34, 36, 76, 157–164, 174, 175, 180, 181, 186, 190–193, 199, 203, 204, 207, 208, 210–213, 215, 218–221, 265, 267, 324, 326, 339–341, 368, 369, 389, 392–398, 478, 489, 543–578]. PIM moves computation close to application data by equipping memory chips with processing capabilities [160, 545]. Prior works [5, 21, 22, 28, 29, 32–35, 37, 76, 141, 208, 212, 213, 216, 218, 266, 267, 369, 546, 546, 550, 579–581] propose PIM architectures wherein a processor logic layer is tightly integrated with DRAM memory layers using 2.5D/3D-stacking technologies [467, 468, 472]. Nonetheless, the 2.5D/3D integration itself might not always be able to provide significantly higher memory bandwidth for processors than standard DRAM [339, 340]. To provide even higher bandwidth for the in-memory processors, *near-bank* PIM designs have been explored [157, 161, 162, 338–341, 385–398]. *Near-bank* PIM designs tightly couple a PIM core with each DRAM bank, exploiting bank-level parallelism to expose high on-chip memory bandwidth of standard DRAM to processors. Moreover, manufacturers of near-bank PIM architectures avoid disturbing the key components (i.e., subarray and bank) of commodity DRAM to provide a cost-efficient and practical way for silicon materialization. Two *real* near-bank PIM architectures are Samsung’s FIMDRAM [340, 341] and the UPMEM PIM system [157, 161, 162, 399].

Most near-bank PIM architectures [157, 161, 162, 338–341, 385–390] support several PIM-enabled memory chips connected to a host CPU via memory channels. Each memory chip comprises multiple PIM cores, which are low-area and low-power cores with relatively low computation capability [161, 162], and each of them is located close to a DRAM bank [157, 161, 162, 338–341, 385–390]. Each PIM core can access data located on their local DRAM banks, and typically there is no direct communication channel among PIM cores. Overall, near-bank PIM architectures provide high levels of parallelism and very large memory bandwidth, thereby being a very promising computing platform to accelerate memory-bound kernels. Recent works leverage near-bank PIM architectures to provide high performance and energy benefits on bioinformatics [161, 162, 400, 401], skyline computation [402], compression [403] and neural network [161, 162, 340, 385, 387] kernels. A recent study [161, 162] provides PrIM benchmarks [404], which are a collection of 16 kernels for evaluating near-bank PIM architectures, like the UPMEM PIM system. However, there is *no* prior work to thoroughly study the widely used, memory-bound SpMV kernel on a real PIM system.

Our work is the first to efficiently map the SpMV execution kernel on near-bank PIM systems, and understand its performance implications on a real PIM system. Specifically, our **goal** in this work is twofold: (i) design efficient SpMV algorithms to accelerate this kernel in current and future PIM systems, while covering a wide variety of sparse matrices with diverse sparsity patterns, and (ii) provide an extensive characterization analysis of the widely used SpMV kernel on a real PIM architecture. To this end, we provide a wide variety of SpMV implementations for real PIM architectures, and conduct a rigorous experimental analysis of SpMV kernels in the UPMEM PIM system, the first publicly-available real-world PIM architecture.

We present the *SparseP* library [6] that includes 25 SpMV kernels for real PIM systems, supporting various (1) data types, (2) data partitioning techniques of the sparse matrix to PIM-enabled memory, (3) compressed matrix formats, (4) load balancing schemes across PIM cores, (5) load balancing schemes across threads of a multithreaded PIM core, and (6) synchronization approaches among

threads within PIM core. We support a wide range of data types, i.e., 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float and 64-bit float data types to cover a wide variety of real-world applications that employ SpMV as their underlying kernel. We design two types of well-crafted data partitioning techniques: (i) the 1D partitioning technique to perform the complete SpMV computation only using PIM cores, and (ii) the 2D partitioning technique to strive a balance between computation and data transfer costs to PIM-enabled memory. In the 1D partitioning technique, the matrix is horizontally partitioned across PIM cores, and the *whole* input vector is copied into the DRAM bank of *each* PIM core, while PIM cores directly compute the elements of the final output vector. In the 2D partitioning technique, the matrix is split in 2D tiles, the number of which is equal to the number of PIM cores, and a *subset* of the elements of the input vector is copied into the DRAM bank of each PIM core. However, in the 2D partitioning technique, PIM cores create a large number of partial results for the elements of the output vector which are gathered and merged by the host CPU cores to assemble the final output vector. We support the most popular compressed matrix formats, i.e., CSR [582, 583], COO [583, 584], BCSR [585], BCOO [583], and for each compressed format we implement various load balancing schemes across PIM cores to provide efficient SpMV execution for a wide variety of sparse matrices with diverse sparsity patterns. Finally, we design several load balancing schemes and synchronization approaches among parallel threads within a PIM core to cover a variety of real PIM systems that provide multithreaded PIM cores.

We conduct an extensive characterization analysis of *SparseP* kernels on the UPMEM PIM system [157, 161, 162, 338] analyzing the SpMV execution using (1) one single multithreaded PIM core, (2) thousands of PIM cores, and (3) comparing it with that achieved on conventional processor-centric CPU and GPU systems. First, we characterize the limits of a single multithreaded PIM core, and show that (i) high operation imbalance across threads of a PIM core can impose high overhead in the core pipeline, and (ii) fine-grained synchronization approaches to increase parallelism cannot outperform a coarse-grained approach, if PIM hardware serializes accesses to the local DRAM bank. Second, we analyze the end-to-end SpMV execution of 1D and 2D partitioning techniques using thousands of PIM cores. Our study indicates that the performance (i) of the 1D partitioning technique is limited by data transfer costs to *broadcast* the whole input vector into *each* DRAM bank of PIM cores, and (ii) of the 2D partitioning technique is limited by data transfer costs to *gather* partial results for the elements of the output vector from PIM-enabled memory to the host CPU. Such data transfers incur high overheads, because they take place via the narrow memory bus. In addition, our detailed study across a wide variety of compressed matrix formats and sparse matrices with diverse sparsity patterns demonstrates that (i) the compressed matrix format determines the data partitioning strategy across DRAM banks of PIM-enabled memory, thereby affecting the computation balance across PIM cores with corresponding performance implications, and (ii) there is *no one-size-fits-all* solution. The load balancing scheme across PIM cores (and across threads within a PIM core) and data partitioning technique that provides the best-performing SpMV execution depends on the characteristics of the input matrix and the underlying PIM hardware. Finally, we compare the SpMV execution on a state-of-the-art UPMEM PIM system with 2528 PIM cores to state-of-the-art CPU and GPU systems, and observe that SpMV on the UPMEM PIM system achieves a much higher fraction of the machine's peak

performance compared to that on the state-of-the-art CPU and GPU systems. Our extensive evaluation provides programming recommendations for software designers, and suggestions and hints for hardware and system designers of future PIM systems.

Our most significant recommendations for PIM software designers are:

1. Design algorithms that provide high load balance across threads of PIM core in terms of computations, loop control iterations, synchronization points and memory accesses.
2. Design compressed data structures that can be effectively partitioned across DRAM banks, with the goal of providing high computation balance across PIM cores.
3. Design *adaptive* algorithms that trade off computation balance across PIM cores for lower data transfer costs to PIM-enabled memory, and adapt their configuration to the particular patterns of each input given, as well as the characteristics of the PIM hardware.

Our most significant suggestions for PIM hardware and system designers are:

1. Provide low-cost synchronization support and hardware support to enable concurrent memory accesses by multiple threads to the local DRAM bank to increase parallelism in a multithreaded PIM core.
2. Optimize the broadcast collective operation in data transfers from main memory to PIM-enabled memory to minimize overheads of copying the input data into all DRAM banks in the PIM system.
3. Optimize the gather collective operation *at DRAM bank granularity* for data transfers from PIM-enabled memory to the host CPU to minimize overheads of retrieving the output results.
4. Design high-speed communication channels and optimized libraries for data transfers to/from thousands of DRAM banks of PIM-enabled memory.

Our *SparseP* software package is freely and publicly available [6] to enable further research on SpMV in current and future PIM systems. The main contributions of this work are as follows:

- We present *SparseP*, the first open-source SpMV software package for real PIM architectures. *SparseP* includes 25 SpMV kernels, supporting the four most widely used compressed matrix formats and a wide range of data types. *SparseP* is publicly available at [6], and can be useful for researchers to improve multiple aspects of future PIM hardware and software.
- We perform the first comprehensive study of the widely used SpMV kernel on the UPMEM PIM architecture, the first real commercial PIM architecture. We analyze performance implications of SpMV PIM execution using a wide variety of (1) compressed matrix formats, (2) data types, (3) data partitioning and load balancing techniques, and (4) 26 sparse matrices with diverse sparsity patterns.
- We compare the performance and energy of SpMV on the state-of-the-art UPMEM PIM system with 2528 PIM cores to state-of-the-art CPU and GPU systems. SpMV execution achieves less than 1% of the peak performance on processor-centric CPU and GPU systems, while it achieves on average 51.7% of the peak performance on the UPMEM PIM system, thus better leveraging the computation capabilities of underlying hardware. The UPMEM PIM system also provides high energy efficiency on the SpMV kernel.



## 5.2 Background and Motivation

### 5.2.1 Sparse Matrix Vector Multiplication (SpMV)

The SpMV kernel multiplies a sparse matrix of size  $M \times N$  with a dense input vector of size  $1 \times N$  to compute an output vector of size  $M \times 1$ . The SpMV kernel is widely used in a variety of applications including graph processing [1, 286, 586, 587], neural networks [275, 588–590], machine learning [271, 591–595], and high performance computing [42, 111, 386, 596–599]. These applications involve matrices with very high sparsity [18, 103, 150, 286, 289–293], i.e., a large fraction of zero elements. Thus, using a compression scheme is a straightforward approach to avoid unnecessarily storing zero elements and performing computations on them. For general sparse matrices, the most widely used storage format is the Compressed Sparse Row (CSR) format [582, 583]. Figure 5.1 presents an example of a compressed matrix using the CSR format (left), and the CSR-based SpMV execution (right), assuming an input vector  $x$  and an output vector  $y$ .

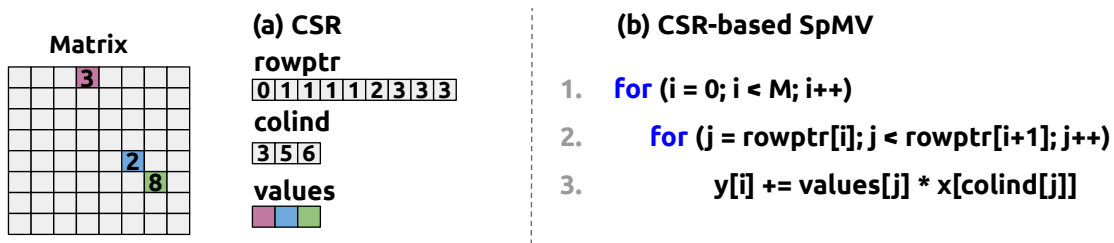


Figure 5.1: (a) CSR representation of a sparse matrix. (b) CSR-based SpMV implementation.

### Compressed Matrix Storage Formats

Several prior works [17, 116, 117, 120, 132, 384, 582–585, 600–613] propose compressed storage formats for sparse matrices, which are typically of two types [286]. The first approach is to design general purpose compressed formats, such as CSR [582, 583], CSR5 [601], COO [583, 584], BCSR [585], and BCOO [583]. Such encodings are general in applicability and are highly-efficient in storage. The second approach is to leverage a certain known structure in a given type of sparse matrix. For example, the DIA format [603] is effective in matrices where the non-zero elements are concentrated along the diagonals of the matrix. Such encodings aim to improve performance of sparse matrix computations by specializing to particular matrix patterns, but they sacrifice generality. In this work, we explore with the four most widely used *general* compressed formats (Figure 5.2), which we describe in more detail next.

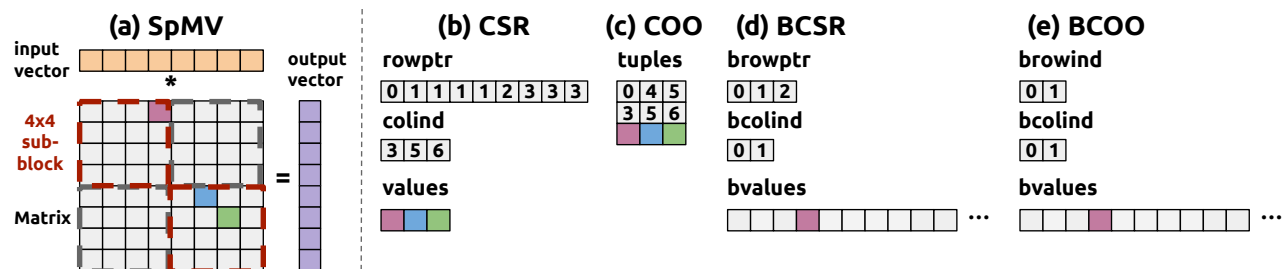


Figure 5.2: (a) SpMV with a dense matrix representation, and (b) CSR, (c) COO, (d) BCSR, (e) BCOO formats.

**Compressed Sparse Row (CSR) [582,583].** The CSR format (Figure 5.2b) sequentially stores values in a row-wise order. A column index array (`colind[]`) and a value array (`values[]`) store the column index and value of each non-zero element, respectively. An array, named `rowptr[]`, stores the location of the first non-zero element of each row within the `values[]` array. The values of an adjacent pair of the `rowptr[]` array, i.e., `rowptr[i, i+1]`, represent a slice of the `colind[]` and `values[]` arrays. The corresponding slice of the `colind[]` and `values[]` arrays stores the column indices and the values of the non-zero elements, respectively, for the  $i$ -th row of the original matrix.

**Coordinate Format (COO) [583,584].** The COO format (Figure 5.2c) stores the non-zero elements as a series of tuples (`tuples[]` array). Each tuple includes the row index, column index, and value of the non-zero element.

**Block Compressed Sparse Row (BCSR) [585].** The BCSR format (Figure 5.2d) is a block representation of CSR. Instead of storing and indexing single non-zero elements, BCSR stores and indexes  $r \times c$  sub-blocks with at least one non-zero element. The original matrix is split into  $r \times c$  sub-blocks. Figure 5.2d shows an example of BCSR assuming  $4 \times 4$  sub-blocks. The original matrix of Figure 5.2a is split into four sub-blocks, and two of them (highlighted with red color) contain at least one non-zero element. The `bvalues[]` array stores the values of all the *non-zero sub-blocks* of the original matrix. Each non-zero sub-block is stored in the `bvalues[]` array with a dense representation, i.e., padding with zero values when needed. The `bcolind[]` array stores the block-column index of each non-zero sub-block. The `browptr[]` array stores the location of the first non-zero sub-block of each block row within the `bcolind[]` array, assuming a block row represents  $r$  consecutive rows of the original matrix, where  $r$  is the vertical dimension of the sub-block.

**Block Coordinate Format (BCOO) [583].** The BCOO format is the block counterpart of COO. The `browind[]`, `bcolind[]` and `bvalues[]` arrays store the row indices, column indices and values of the non-zero sub-blocks, respectively. Figure 5.2e shows an example of BCOO, assuming  $4 \times 4$  sub-blocks.

## SpMV in Processor-Centric Systems

Many prior works [18, 103, 111, 120, 132, 139, 292, 294, 320–323] generally show that SpMV performs poorly on commodity CPU and GPU systems, and achieves a small fraction of the peak performance (e.g., 10% of the peak performance [322]) due to its algorithmic nature, the employed compressed matrix storage format and the sparsity pattern of the matrix.

The SpMV kernel is highly bottlenecked by the memory subsystem in processor-centric CPU and GPU systems due to three reasons. First, due to its algorithmic nature there is *no* temporal locality in the input matrix. Unlike traditional algebra kernels like Matrix Matrix Multiplication or LU decomposition, the elements of the matrix in SpMV are used only *once* [291, 294]. Second, due to the sparsity of the matrix, the matrix is stored in a compressed format (e.g., CSR) to avoid unnecessary computations and data accesses. Specifically, the non-zero elements of the matrix are stored contiguously in memory, while additional data structures assist in the proper traversal of the matrix, i.e., to discover

the positions of the non-zero elements. For example, CSR uses the `rowptr[]` and `colind[]` arrays to discover the positions of the non-zero elements of the matrix. These additional data structures cause additional memory access operations, memory bandwidth pressure and contention with other requests in the memory subsystem. Third, due to the sparsity of the input matrix, SpMV causes irregular memory accesses to the elements of the input vector  $x$ . The memory accesses to the elements of the input vector are input driven, i.e., they follow the sparsity pattern of the input matrix. This irregularity results to poor data locality on the elements of the input vector and expensive data accesses, because it increases the average access latency due to a high number of cache misses on commodity systems with deep cache hierarchies [291, 294]. As a result, memory-centric near-bank PIM systems constitute a better fit for the widely used SpMV kernel, because they provide high levels of parallelism, large aggregate memory bandwidth and low memory access latency [161, 162, 338–340].

### 5.2.2 Near-Bank PIM Systems

Figure 5.3 shows the baseline organization of a near-bank PIM system that we assume in this work. The PIM system consists of a host CPU, standard DRAM memory modules, and PIM-enabled memory modules. PIM-enabled modules are connected to the host CPU using one or more memory channels, and include multiple PIM chips. A PIM chip (Figure 5.3 right) tightly integrates a low-area PIM core with a DRAM bank. We assume that each PIM core can additionally include a small private instruction memory and a small data (scratchpad or cache) memory. PIM cores can access data located on their local DRAM bank, and typically there is no direct communication channel among PIM cores. The DRAM banks of PIM chips are accessible by the host CPU for copying input data and retrieving results via the memory bus.

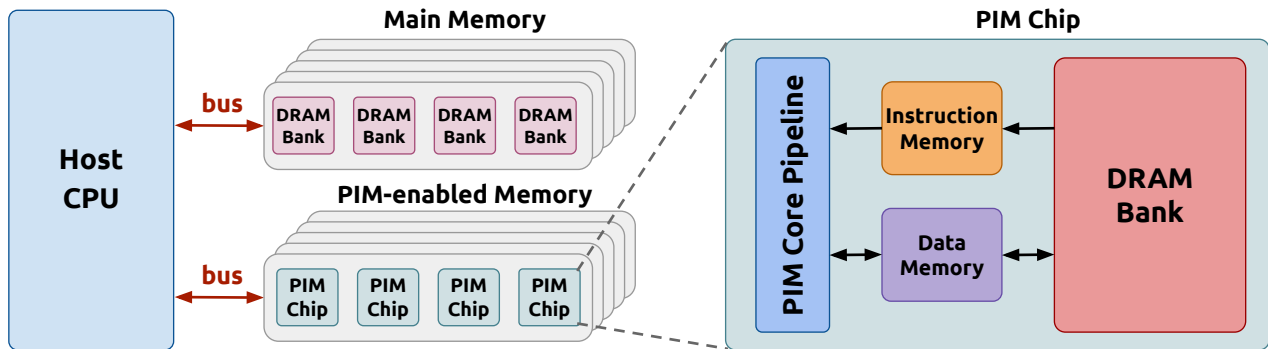


Figure 5.3: High-level organization of a near-bank PIM architecture.

#### The UPMEM PIM Architecture

The UPMEM PIM system [157, 161, 162] includes the host CPU with standard main memory, and UPMEM PIM modules. An UPMEM PIM module is a standard DDR4-2400 DIMM [614] with 2 ranks. Each rank contains 64 PIM cores, which are called DRAM Processing Units (DPUs). In the current UPMEM PIM system, there are 20 double-rank PIM DIMMs with 2560 DPUs.<sup>1</sup>

<sup>1</sup>There are thirty two faulty DPUs in the system where we run our experiments. They cannot be used and do not affect the correctness of our results, but take away from the system's full computational power of 2560 DPUs.

**DPU Architecture and Interface.** Each DPU has exclusive access to a 24-KB instruction memory, called **IRAM**, a 64-KB scratchpad memory, called **WRAM**, and a 64-MB DRAM bank, called **MRAM**. A DPU is a multithreaded in-order 32-bit RISC core that can potentially reach 500 MHz [338]. The DPU has 24 hardware threads, each of which has 24 32-bit general purpose registers. The DPU pipeline has 14 stages, and supports a single cycle 8x8-bit multiplier. Multiplications on 64-bit integers, 32-bit floats and 64-bit floats are not supported in hardware, and require longer routines with a large number of operations [161, 162, 338]. Threads share the IRAM and WRAM, and can access the MRAM by executing transactions at 64-bit granularity via a DMA engine, i.e., data can be accessed from/to MRAM as a multiple of 8 bytes, up to 2048 bytes. MRAM transactions are serialized in the DMA engine. The ISA provides DMA instructions to move instructions from MRAM to IRAM, or data between MRAM and WRAM. The DPU accesses the WRAM through 8-, 16-, 32- and 64-bit load/store instructions. DPUs use the *Single Program Multiple Data* programming model, where software threads, called **tasklets**, execute the same code, but operate in different pieces of data, and can execute different control-flow paths during runtime. Tasklets can synchronize using mutexes, barriers, handshakes and semaphores provided by the UPMEM runtime library.

**CPU-DPU Data Transfers.** Standard main memory and PIM-enabled memory have different data layouts. The UPMEM SDK [615] has a transposition library to execute necessary data shuffling when moving data between main memory and MRAM banks of PIM-enabled memory modules via a programmer-transparent way. The CPU-DPU and DPU-CPU data transfers can be performed in parallel, i.e., concurrently across multiple MRAM banks, with the limitation that *the transfer sizes from/to all MRAM banks need to be the same*. The UPMEM SDK provides two options: (i) perform parallel transfers to all MRAM banks of all ranks, or (ii) iterate over each rank to perform parallel transfers to MRAM banks of the same rank, and serialize data transfers across ranks.

## 5.3 The *SparseP* Library

This section describes the parallelization techniques that we explore for SpMV on real PIM architectures, and presents the SpMV implementations of our *SparseP* package. Section 5.3.1 describes SpMV execution on a real PIM system. Section 5.3.2 presents an overview of the data partitioning techniques that we explore. Section 5.3.3 and Section 5.3.4 describe in detail the parallelization techniques across PIM cores, and across threads within a PIM core, respectively. Section 5.3.5 describes the kernel implementation for all compressed matrix storage formats.

### 5.3.1 SpMV Execution on a PIM System

Figure 5.4 shows the SpMV execution on a real PIM system, which is broken down in four steps: (1) the time to load the input vector into DRAM banks of PIM-enabled memory (**load**), (2) the time to execute the SpMV kernel on PIM cores (**kernel**), (3) the time to retrieve from DRAM banks to the host CPU results for the output vector (**retrieve**), and (4) the time to merge partial results and assemble the final output vector on the host CPU (**merge**). In our analysis, we omit the time

to load the matrix into PIM-enabled memory, since this step can typically be hidden in real-world applications (it can be overlapped with other computation performed by the application or amortized if the application performs multiple SpMV iterations on the same matrix).

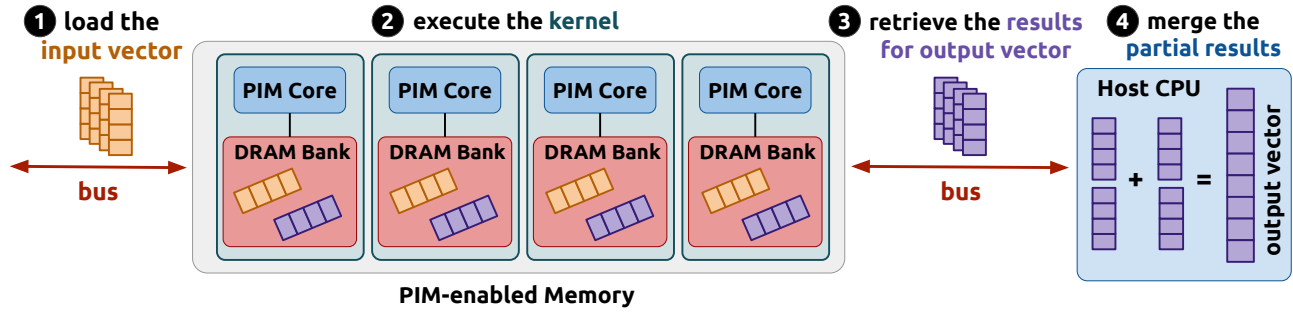


Figure 5.4: Execution of the SpMV kernel on a real PIM system.

### 5.3.2 Overview of Data Partitioning Techniques

To parallelize the SpMV kernel, we implement well-crafted data partitioning schemes to split the matrix across multiple DRAM banks of PIM cores. *SparseP* supports two general types of data partitioning techniques, shown in Figure 5.5.

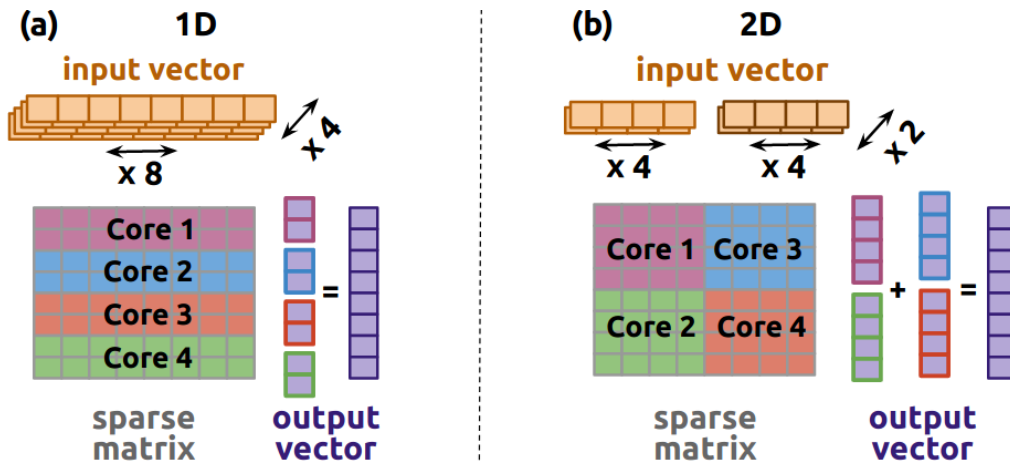


Figure 5.5: Data partitioning techniques of the *SparseP* package.

First, we provide an 1D partitioning technique (Figure 5.5a), where the matrix is horizontally partitioned across PIM cores, and the whole input vector is copied into the DRAM bank of each PIM core. With the 1D partitioning technique, almost the entire SpMV computation is performed using only PIM cores, since the merge step in the host CPU is negligible: a very small number of partial results is created, i.e., only for a few rows that are split across neighboring PIM cores. Thus, the number of partial elements of the output vector is at most equal to the number of PIM cores used. Second, we provide a 2D partitioning technique (Figure 5.5b), where the matrix is partitioned into 2D tiles, the number of which is equal to the number of PIM cores. With the 2D partitioning technique, we aim to strive a balance between computation and data transfer costs, since only a subset of the elements of the input vector is copied into the DRAM bank of each PIM core. However, PIM cores assigned to tiles that horizontally overlap, i.e., tiles that share the same rows of the original

matrix (rows that are split across multiple tiles), produce *many* partial results for the elements of the output vector. These partial results are transferred to the host CPU, and merged by CPU cores, which assemble the final output vector. In the *SparseP* library, the merge step performed by the CPU cores is parallelized using the OpenMP API [616].

In both data partitioning schemes, matrices are stored in a row-sorted way, i.e., the non-zero elements are sorted in increasing order of their row indices. Therefore, each PIM core computes results for a *continuous* subset of elements of the output vector. This way we minimize data transfer costs, since we only transfer necessary data to the host CPU, i.e., *the values* of the elements of the output vector produced at PIM cores. If each PIM core instead computed results for a *non-continuous* subset of elements of the output vector, an additional array *per core*, which would store *the indices* of the *non-continuous* elements within the output vector, would need to be transferred to the host CPU, causing additional data transfer overheads.

### 5.3.3 Parallelization Techniques Across PIM Cores

To parallelize SpMV across multiple PIM cores *SparseP* supports various parallelization schemes for both 1D and 2D partitioning techniques.

#### 1D Partitioning Technique

To efficiently parallelize SpMV across multiple PIM cores via the 1D partitioning technique, *SparseP* provides various load balancing schemes for each supported compressed matrix format. Figure 5.6 presents an example of parallelizing SpMV across multiple PIM cores using load balancing schemes for the CSR and COO formats. For the CSR and COO formats, we balance either the rows, such that each PIM core processes almost the same number of rows, or the non-zero elements, such that each PIM core processes almost the same number of non-zero elements. In the CSR format, since the matrix is stored in row-order, i.e., the `rowptr[ ]` array stores the index pointers of the non-zero elements of *each row*, and thus balancing the non-zero elements across PIM cores is performed at row granularity. In the COO format, the matrix is stored in non-zero order using the `tuples[ ]` array, and thus balancing the non-zero elements can be performed either at row granularity, or by splitting a row across two neighboring PIM cores to provide a near-perfect non-zero element balance across cores. In the latter case, as mentioned, a small number of partial results for the output vector is merged by the host CPU: if the row is split between two neighboring PIM cores at most one element needs to be accumulated at the host CPU cores.

Figure 5.7 presents an example of parallelizing SpMV across multiple PIM cores using load balancing schemes of the BCSR and BCOO formats. In Figure 5.7, the cells of the matrix represent sub-blocks of size 4x4: the *grey* cells represent sub-blocks that do not have *any* non-zero element, and the *colored* cells represent sub-blocks that have *k* non-zero elements, where *k* is the number shown inside the colored cell. In the BCSR and BCOO formats, since the matrix is stored in sub-blocks of non-zero elements, we balance either the blocks, such that each PIM core processes almost the same number of blocks, or the non-zero elements, such that each PIM core processes almost the same number of

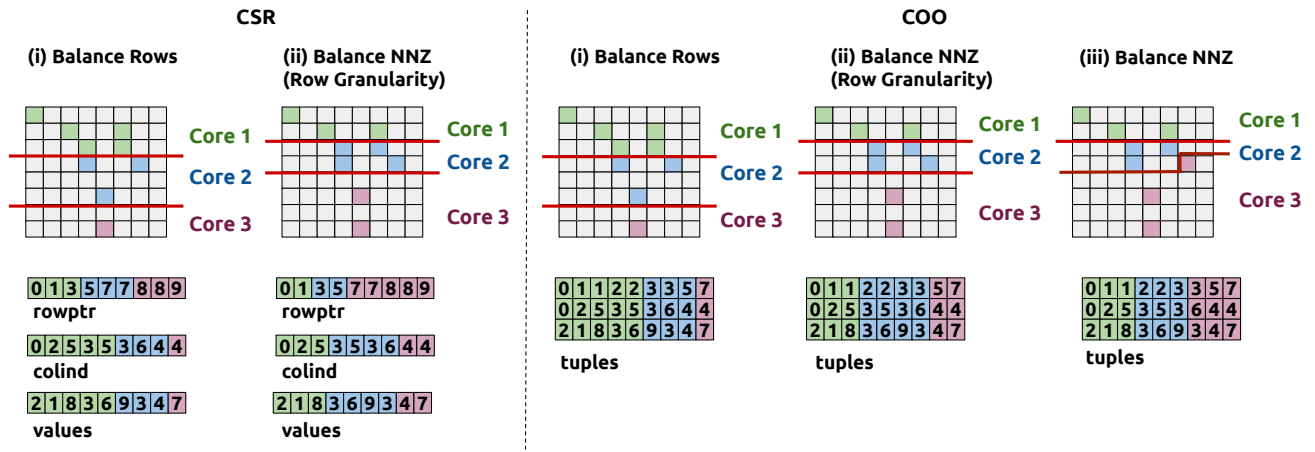


Figure 5.6: Load balancing schemes across PIM cores for the CSR (left) and COO (right) formats with the 1D partitioning technique. The colored cells of the matrix represent non-zero elements.

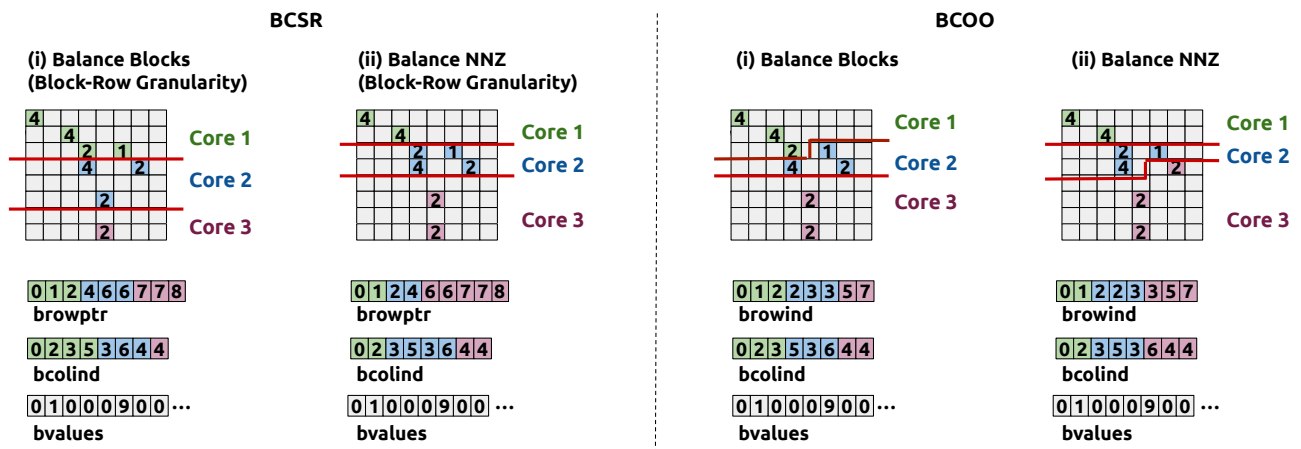


Figure 5.7: Load balancing schemes across PIM cores for the BCSR (left) and BCOO (left) formats with the 1D partitioning technique. The cells of the matrix represent sub-blocks of size 4x4. The colored cells of the matrix represent non-zero sub-blocks, and the number inside a colored cell describes the number of non-zero elements of the corresponding sub-block.

non-zero elements. Similarly to CSR, in the BCSR format, the matrix is stored in block-row-order, i.e., the `browptr[ ]` array stores the index pointers of the non-zero blocks of *each block row* (recall that a block row represents  $r$  consecutive rows of the original matrix, where  $r$  is the vertical dimension of the sub-block), and thus balancing the blocks or the non-zero elements across cores is limited to be performed at block-row granularity. In the BCOO format, given that a block-row might be split across two PIM cores, a small number of partial results for the output vector is merged by the host CPU: between two neighboring PIM cores at most block size  $r$  elements ( $r$  is the vertical dimension of the block size) might need to be accumulated at the host CPU cores.

## 2D Partitioning Technique

*SparseP* includes three 2D partitioning techniques, shown in Figure 5.8:

1. **equally-sized** (Figure 5.8a): The 2D tiles are statically created to have the same height and width. This way the subsets of the elements for the input and output vectors have the same sizes across all PIM cores.



2. **equally-wide** (Figure 5.8b): The 2D tiles have the same width and variable height. This way the subset of the elements for the input vector has the same size across PIM cores, while the subset of the elements for the output vector varies across PIM cores. We balance the non-zero elements across the tiles of the *same* vertical partition, such that we can provide high non-zero element balance across PIM cores assigned to the same vertical partition.
3. **variable-sized** (Figure 5.8c): The 2D tiles have both variable width and height. We balance the non-zero elements both across the vertical partitions and across the tiles of the *same* vertical partition. This way we can provide high non-zero element balance across all PIM cores.

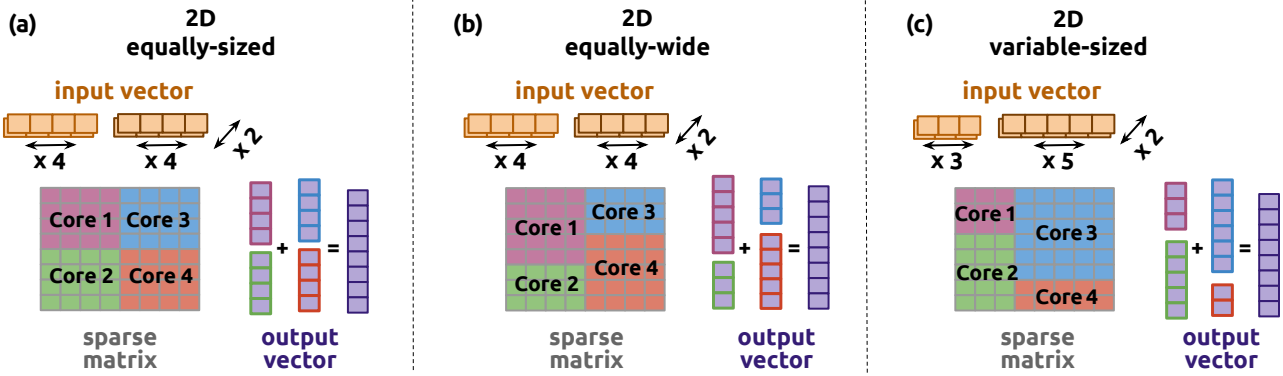


Figure 5.8: The 2D partitioning techniques of *SparseP* package assuming 4 PIM cores and 2 vertical partitions.

*SparseP* provides various load balancing schemes across PIM cores in the *equally-wide* and *variable-sized* techniques. In the *equally-wide* technique, for the CSR and COO formats, we balance the non-zero elements across the tiles of the same vertical partition. Load balancing in the CSR format is performed at row-granularity, i.e., splitting the `rowptr[]` array across PIM cores. For the BCSR and BCOO formats, we balance either the blocks or the non-zero elements across the tiles of the same vertical partition. Load balancing in the BCSR format is performed at block-row granularity, i.e., splitting the `browptr[]` array across PIM cores. In the *variable-sized* technique, we first balance the non-zero elements across the vertical partitions, such that the vertical partitions include the same number of non-zero elements. Then, across the tiles of the same vertical partition, we balance the non-zero elements for the CSR (at row-granularity) and COO formats, and either the blocks or the non-zero elements for the BCSR (at block-row granularity) and BCOO formats.

Table 5.1 summarizes the parallelization approaches across PIM cores. Please also see Appendix 8.3 for all SpMV kernels provided by the *SparseP* software package. All kernels support a wide range of data types, i.e., 8-bit integer (**int8**), 16-bit integer (**int16**), 32-bit integer (**int32**), 64-bit integer (**int64**), 32-bit float (**fp32**), and 64-bit float (**fp64**) data types.

### 5.3.4 Parallelization Techniques Across Threads within a PIM Core

PIM cores can support multiple hardware threads to exploit high memory bank bandwidth [161, 162]. To parallelize SpMV across multiple threads within a multithreaded PIM core *SparseP* supports various load balancing schemes for each compressed matrix format, and three synchronization approaches to ensure correctness among threads of a PIM core.



Partitioning Technique	Compressed Format	Load Balancing Across PIM Cores
1D	CSR	rows ( <b>CSR.row</b> ) nnz* ( <b>CSR.nnz</b> )
	COO	rows ( <b>COO.row</b> ) nnz* ( <b>COO.nnz-rgrn</b> ) nnz ( <b>COO.nnz</b> )
	BCSR	blocks <sup>†</sup> ( <b>BCSR.block</b> ) nnz <sup>†</sup> ( <b>BCSR.nnz</b> )
	BCOO	blocks ( <b>BCOO.block</b> ) nnz ( <b>BCOO.nnz</b> )
2D <i>equally-sized</i>	CSR ( <b>DCSR</b> )	-
	COO ( <b>DCOO</b> )	-
	BCSR ( <b>DBC</b> CSR)	-
	BCOO ( <b>DBC</b> OO)	-
2D <i>equally-wide</i>	CSR ( <b>R</b> BD <b>C</b> SR)	nnz*
	COO ( <b>R</b> BD <b>C</b> OO)	nnz
	BCSR	blocks <sup>†</sup> ( <b>R</b> BD <b>B</b> C <b>S</b> R) nnz <sup>†</sup>
	BCOO	blocks ( <b>R</b> BD <b>B</b> C <b>O</b> O) nnz
2D <i>variable-sized</i>	CSR ( <b>B</b> D <b>C</b> SR)	nnz*
	COO ( <b>B</b> D <b>C</b> OO)	nnz
	BCSR	blocks <sup>†</sup> ( <b>B</b> D <b>B</b> C <b>S</b> R) nnz <sup>†</sup>
	BCOO	blocks ( <b>B</b> D <b>B</b> C <b>O</b> O) nnz

Table 5.1: Parallelization techniques across PIM cores of the *SparseP* library. \*: row-granularity, <sup>†</sup>: block-row-granularity

### Load Balancing Approaches

In a similar way as explained in Figure 5.6, for the CSR and COO formats, we balance either the rows, such that each thread processes almost the same number of rows, or the non-zero elements, such that each thread processes almost the same number of non-zero elements. In the CSR format, matrix is stored in row-order, and thus load balancing across threads is performed at row granularity. In the UPMEM PIM system, elements of the output vector are accessed at 64-bit granularity in DRAM memory. Thus, when balancing is performed at row granularity, we assign rows to threads in chunks of  $8/\text{sizeof}(\text{data\_type})$  to ensure 8-byte alignment on the elements of the output vector. In the COO format, balancing the non-zero elements can be performed either at row granularity or by splitting the row between threads, i.e., providing an almost perfect non-zero balance across threads. In the latter case, synchronization among threads for write accesses on the elements of the output vector can be implemented with three synchronization approaches described in Section 5.3.4.

For the BCSR and BCOO formats, we balance either the blocks, such that each thread processes almost the same number of blocks, or the non-zero elements, such that each thread processes almost

the same number of non-zero elements. In the BCSR format, the matrix is stored in block-row order, and thus load balancing across threads is performed at block row granularity. For both formats, the block sizes are *configurable* in *SparseP*. In our evaluation, we use block sizes of 4x4, since these are the most common dimensions to cover various sparse matrices [18, 320, 617]. In the UPMEM PIM architecture, elements of the output vector are accessed at 64-bit granularity. Therefore, for the BCSR format, with an 8-bit integer data type and small block sizes (4x4 or smaller), threads use synchronization primitives to ensure correctness when writing the elements of the output vector. This is because different threads may write to the same 64-bit-aligned DRAM memory location. Synchronization among threads for writes to the elements of the output vector is necessary for all configurations of the BCOO format, and can be implemented with three approaches described next.

### Synchronization Approaches

*SparseP* provides three synchronization approaches.

1. **Coarse-Grained Locking (lb-cg)**. One global mutex protects the elements of the entire output vector.
2. **Fine-Grained Locking (lb-fg)**. Multiple mutexes protect the elements of the output vector. *SparseP* associates mutexes to the elements of the output vector in a round-robin manner. The UPMEM API supports up to 56 mutexes [615]. In our evaluation, we use 32 mutexes such that we can find the corresponding mutex for a particular element of the output vector only with a shift operation on the MRAM address, avoiding costly division operations.
3. **Lock-Free (lf)**. Since the formats are row-sorted or block-row-sorted, race conditions in the elements of the output vector arise *only in a few elements*, i.e., either when a row (or a block row for BCSR/BCOO) is split across threads, or when continuous elements of the output vector processed by different threads belong to the same 64-bit-aligned DRAM location in the UPMEM PIM system. In our proposed lock-free approach, threads temporarily store partial results for these few elements in the data (scratchpad) memory (i.e., WRAM in the UPMEM PIM system), and later one single thread merges the partial results, and writes the final result for the corresponding element of the output vector to the DRAM bank.

Table 5.2 summarizes the parallelization techniques across threads of a PIM core. All kernels support a wide range of data types, i.e., 8-bit integer (**int8**), 16-bit integer (**int16**), 32-bit integer (**int32**), 64-bit integer (**int64**), 32-bit float (**fp32**), and 64-bit float (**fp64**) data types.

### 5.3.5 Kernel Implementation

We briefly describe the *SparseP* implementations for all compressed matrix formats, i.e., the way that threads access data involved in the kernel from/to the local DRAM bank. The SpMV kernels include three types of data structures: (i) the arrays that store the non-zero elements, i.e., the values (`values[ ]`) and the positions of the non-zero elements (`rowptr[ ]`, `colind[ ]` for CSR, `tuples[ ]` for COO, `browptr[ ]`, `bcolind[ ]` for BCSR, `browind[ ]`, `bcolind[ ]` for BCOO),

Compressed Format	Load Balancing Across Threads	Synchronization Approach
CSR	rows ( <b>CSR.row</b> ) nnz* ( <b>CSR.nnz</b> )	- -
COO	rows ( <b>COO.row</b> ) nnz* ( <b>COO.nnz-rgrn</b> ) nnz ( <b>COO.nnz</b> )	- - lb-cg / lb-fg / lf
BCSR	blocks <sup>†</sup> ( <b>BCSR.block</b> ) nnz <sup>†</sup> ( <b>BCSR.nnz</b> )	lb-cg / lb-fg (only for int8 and small block sizes) lb-cg / lb-fg (only for int8 and small block sizes)
BCOO	blocks ( <b>BCOO.block</b> ) nnz ( <b>BCOO.nnz</b> )	lb-cg / lb-fg / lf lb-cg / lb-fg / lf

Table 5.2: Parallelization schemes across threads of a PIM core. \*: row-granularity, <sup>†</sup>: block-row-granularity

(ii) the array that stores the elements of the input vector, and (iii) the array that stores the partial results created for the elements of the output vector.

First, SpMV performs streaming memory accesses to the arrays that store the non-zero elements and their positions. Therefore, to exploit spatial locality and immense bandwidth in data (scratchpad or cache) memory, each thread reads the non-zero elements by fetching large chunks of bytes in a coarse-grained manner from DRAM to data memory (i.e., WRAM in the UPMEM PIM system). Then, it accesses elements through data memory in a fine-grained manner. In the UPMEM PIM system, we fetch chunks of 256-byte data to discover the non-zero elements, as suggested by the UPMEM API [615], since 256-byte transfer sizes highly exploit the available local bandwidth of DRAM bank [161, 162]. For the BCSR and BCOO formats, only for the array that stores the values of the non-zero elements (i.e., `bvalues[]`), we fetch from DRAM to data memory block size chunks, i.e., chunks of  $r \times c \times \text{sizeof}(\text{data\_type})$  bytes, assuming that the matrix is stored in blocks of size  $r \times c$ .

Second, SpMV causes irregular memory accesses to the elements of the input vector. Specifically, the accesses to the elements of the input vector are input-driven, i.e., they are determined by the column positions (column indexes) of the non-zero elements of each particular matrix. Given that matrices involved in SpMV are very sparse [18, 103, 150, 286, 289–293], i.e., the column indexes of the non-zero elements significantly vary, memory accesses to the input vector incur poor data locality. Thus, in our SpMV implementations, threads of a PIM core directly access elements of the input vector through DRAM bank at fine-granularity [161, 162, 615], i.e., using the smallest possible granularity: for the CSR and COO formats at 64-bit granularity, and for the BCSR and BCOO formats at the granularity of  $c \times \text{sizeof}(\text{data\_type})$  bytes, where  $c$  is the horizontal dimension of the block size.

Third, regarding the output vector, threads temporarily store partial results for the same elements of the output vector in data (scratchpad or cache) memory to exploit data locality, until all the non-zero elements of the *same* row or the *same* block row have been traversed (recall matrices are stored in a row-sorted way). Then, the produced results are written to DRAM bank at fine-granularity [161, 162, 615]: for the CSR and COO formats at 64-bit granularity, and for the BCSR and BCOO formats at the granularity of  $r \times \text{sizeof}(\text{data\_type})$  bytes, where  $r$  is the vertical dimension of the block size.

## 5.4 Evaluation Methodology

We conduct our evaluation on an UPMEM PIM system that includes a 2-socket Intel Xeon Silver 4110 CPU [618] at 2.10 GHz (host CPU), standard main memory (DDR4-2400) [614] of 128 GB, and 20 UPMEM PIM DIMMs with 160 GB PIM-capable memory and 2560 DPUs.<sup>2</sup>

First, we evaluate SpMV execution using one single DPU and multiple tasklets (Section 5.5). Table 5.3 shows our evaluated small matrices that fit in the 64 MB DRAM memory of a single DPU. The evaluated matrices vary in sparsity (i.e.,  $\text{NNZ} / (\text{rows} \times \text{columns})$ ), standard deviation of non-zero elements among rows (NNZ-r-std) and columns (NNZ-c-std). The highlighted matrices in Table 5.3 with red color exhibit block pattern [17, 18], i.e., they include *a lot* of dense sub-blocks (almost all their non-zero elements fit in dense sub-blocks).

Matrix Name	Sparsity	NNZ-r-std	NNZ-c-std
delaunay_n13	7.32e-04	1.343	1.343
wing_nodal	1.26e-03	2.861	2.861
raefsky4	3.396e-03	15.956	15.956
pkustk08	0.006542	61.537	61.537

Table 5.3: Small Matrix Dataset.

Second, we evaluate SpMV execution using *multiple* DPUs of the UPMEM PIM system (Section 5.6). We evaluate SpMV execution using both 1D (Section 5.6.1) and 2D (Section 5.6.2) partitioning techniques, and compare them (Section 5.6.3) using a wide variety of sparse matrices with diverse sparsity patterns. We select 22 representative sparse matrices from the Sparse Suite Collection [532], the characteristics of which are shown in Table 5.4. As the values of the last two metrics increase (i.e., NNZ-r-std and NNZ-c-std), the matrix becomes very irregular [108, 109], and is referred to as *scale-free* matrix. In our evaluation, we refer to all matrices between hgc to bns matrices of Table 5.4 as *regular* matrices. The matrices in which NNZ-r-std is larger than 25, i.e., all matrices between wbs to ask in Table 5.4, we refer to as *scale-free* matrices. Please see Appendix 8.4 for a complete description of our dataset of large sparse matrices.

Third, we compare the performance and energy consumption of SpMV execution on the UPMEM PIM system to those on the Intel Xeon Silver 4110 CPU [618] and the NVIDIA Tesla V100 GPU [619] (Section 5.7).

In Section 5.8, we summarize our key takeaways and provide programming recommendations for software designers, and suggestions and hints for hardware and system designers of future PIM systems.

<sup>2</sup>There are thirty two faulty DPUs in the system where we run our experiments. They cannot be used and do not affect the correctness of our results, but take away from the system's full computational power of 2560 DPUs.

Matrix Name	Sparsity	NNZ-r-std	NNZ-c-std
hugetric-00020 ( <b>hgc</b> )	4.21e-07	0.031	0.031
mc2depi ( <b>mc2</b> )	7.59e-06	0.076	0.076
parabolic_fem ( <b>pfm</b> )	1.33e-05	0.153	0.153
roadNet-TX ( <b>rtn</b> )	1.98e-06	1.037	1.037
rajat31 ( <b>rjt</b> )	9.24e-07	1.106	1.106
<b>af_shell1 (ash)</b>	6.90e-05	1.275	1.275
delaunay_n19 ( <b>del</b> )	1.14e-05	1.338	1.338
thermomech_dK ( <b>tdk</b> )	6.81e-05	1.431	1.431
memchip ( <b>mem</b> )	2.02e-06	2.062	1.173
amazon0601 ( <b>amz</b> )	2.08e-05	2.79	15.29
FEM_3D_thermal2 ( <b>fth</b> )	1.59e-04	4.481	4.481
web-Google ( <b>wbg</b> )	6.08e-06	6.557	38.366
<b>ldoor (ldr)</b>	5.13e-05	11.951	11.951
poisson3Db ( <b>psb</b> )	3.24e-04	14.712	14.712
<b>boneS10 (bns)</b>	6.63e-05	20.374	20.374
webbase-1M ( <b>wbs</b> )	3.106e-06	25.345	36.890
in-2004 ( <b>in</b> )	8.846e-06	37.230	144.062
<b>pkustk14 (pks)</b>	6.428e-04	46.508	46.508
com-Youtube ( <b>cmb</b> )	4.639e-06	50.754	50.754
as-Skitter ( <b>skt</b> )	7.71e-06	136.861	136.861
sx-stackoverflow ( <b>sxw</b> )	5.352e-06	137.849	65.367
ASIC_680k ( <b>ask</b> )	8.303e-06	659.807	659.807

Table 5.4: Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern. The highlighted matrices with red color exhibit block pattern [17, 18].

## 5.5 Analysis of SpMV Execution on One DPU

This section characterizes SpMV performance with various load balancing schemes and compressed matrix formats using multiple tasklets in a single DPU. Section 5.5.1 compares load balancing schemes of each compressed matrix format, and Section 5.5.2 compares the scalability of various compressed matrix formats.

### 5.5.1 Load Balancing Schemes Across Tasklets of One DPU

We compare the parallelization schemes of each compressed matrix format supported by *SparseP* library (presented in Table 5.2) across multiple threads of a multithreaded PIM core. Figure 5.9 compares the load balancing schemes of each compressed matrix format using 16 tasklets in a single DPU. For the BCSR and BCOO formats, we omit results for the fine-grained locking approach, since it performs similarly with the coarse-grained locking approach: as we explain in Appendix 8.1.1, fine-grained locking does not increase parallelism over coarse-grained, since in the UPMEM PIM hardware, DRAM memory accesses of the critical section are serialized in the DMA engine of the DPU [161, 162, 615].

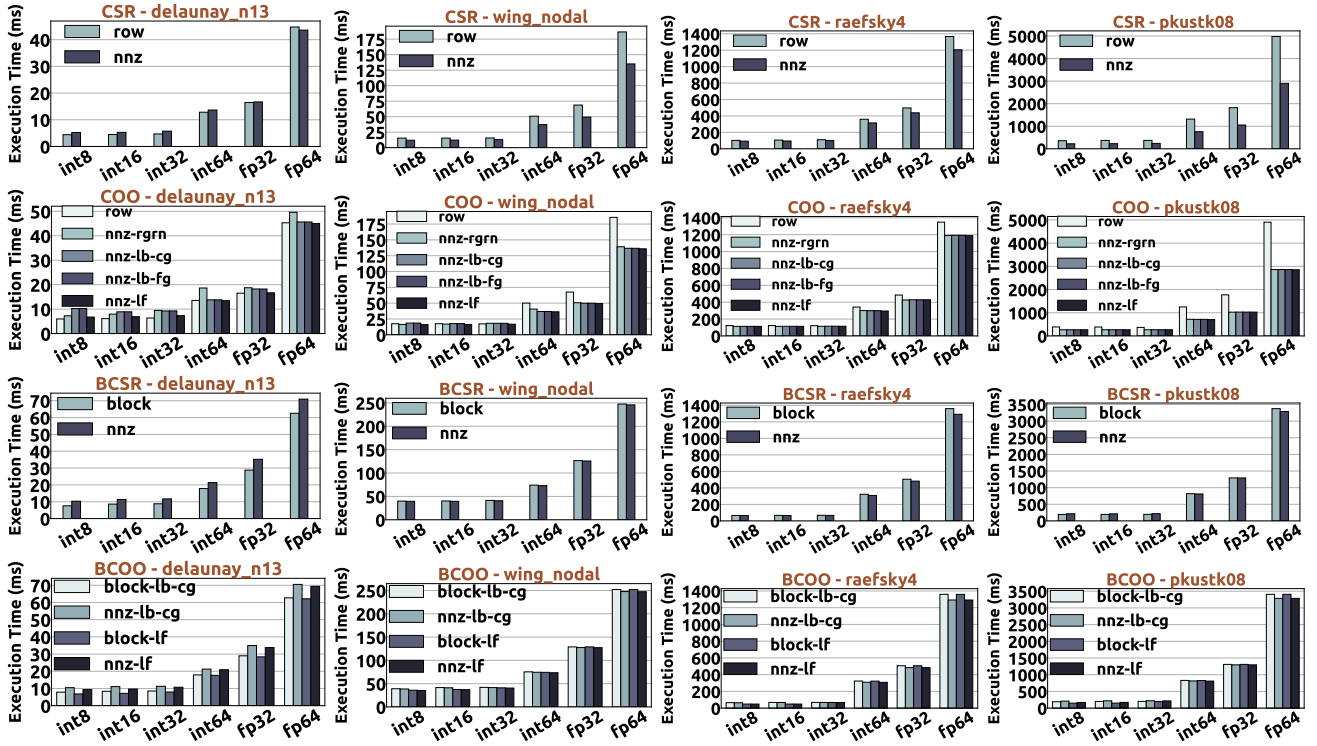


Figure 5.9: Execution time achieved by various load balancing schemes of each compressed matrix format using 16 tasklets of a single DPU.

We draw four findings from Figure 5.9. First, we find that SpMV execution using `int8`, `int16`, and `int32` data types achieves similar execution times across them. This is because the multiplication operation of these data types is sufficiently supported by hardware [162]. In contrast, execution time sharply increases when using more heavyweight data types, i.e., `int64` and floating point data types, in which multiplication is emulated in software using the 8x8-bit multiplier of the DPU [161,162,615].

Second, we observe that balancing the non-zero elements across tasklets typically outperforms balancing the rows for the CSR/COO formats or blocks for the BCSR/BCOO formats, since the non-zero element multiplications are computationally very expensive and can significantly affect load balance across tasklets. However, in `delaunay_n13` matrix, balancing the non-zero elements causes high row/block imbalance across tasklets, since one tasklet processes a significantly higher number of rows/blocks over the rest, thereby causing high operation imbalance across tasklets within the DPU core pipeline. As a result, balancing the rows/blocks outperforms balancing the non-zero elements due to the particular pattern of `delaunay_n13` matrix. In addition, performance benefits of balancing the blocks over balancing the non-zero elements are significant in the BCSR/BCOO formats, because they operate at block granularity and incur high loop control costs.

Third, we observe that the lock-free approach (`COO.nnz-lf`) outperforms the lock-based approaches (`COO.nnz-lb-cg` and `COO.nnz-lb-fg`) in `delaunay_n13` matrix, especially in data types where the multiplication operation is supported directly in hardware. In `delaunay_n13` matrix, one tasklet processes a much larger number of rows than the rest, i.e., it performs a much larger number of critical sections than the rest. In other words, one tasklet performs a much larger number of lock acquisitions/releases and memory instructions than the rest. Thus, lock-based approaches cause high operation imbalance in the DPU core pipeline with significant performance

costs. Instead, lock-free and lock-based approaches in the BCOO format perform similarly, since lock acquisition/release costs can be hidden due to BCOO's higher loop control costs and larger critical sections. Overall, based on the second and the third findings, we conclude that in matrices and formats, where the load balancing and/or the synchronization scheme used cause *high* disparity in the number of non-zero elements/blocks/rows processed across tasklets or the number of lock acquisitions/lock releases/memory accesses performed across tasklets, the DPU core pipeline can incur significant performance overheads.

#### OBSERVATION 1:

*High operation imbalance* in computation, control, synchronization, or memory instructions executed by multiple threads of a PIM core can cause *high performance overheads* in the compute-bound and area-limited PIM cores.

Fourth, we find that the fine-grained locking approach (COO . nnz - 1b - fg) performs similarly with the coarse-grained locking approach (COO . nnz - 1b - cg). This is because the critical section includes memory accesses to the local DRAM bank, which, in the UPMEM PIM hardware, are serialized in the DMA engine of the DPU. Therefore, fine-grained locking does not increase execution parallelism over coarse-grained locking, since concurrent accesses to MRAM bank are not supported in the UPMEM PIM hardware. Fine-grained locking does not improve performance over coarse-grained locking, also when using block-based formats (e.g., BCSR/BCOO formats), as we demonstrate in Appendix 8.1.1. Therefore, we recommend PIM hardware designers to provide lightweight synchronization mechanisms [5] for PIM cores, and/or enable concurrent accesses to local DRAM memory, e.g., supporting sub-array level parallelism [180, 489, 552, 557, 560, 620–622] or multiple DRAM banks per PIM core.

#### OBSERVATION 2:

*Fine-grained* locking approaches to parallelizing critical sections that perform memory accesses to different DRAM memory locations cannot improve performance over *coarse-grained* locking, when the PIM hardware does not support *concurrent accesses to a DRAM bank*.

### 5.5.2 Analysis of Compressed Matrix Formats on One DPU

We compare the scalability and the performance achieved by various compressed matrix formats. Figure 5.10 compares the supported compressed formats for the int8 (top graphs) and fp64 (bottom graphs) data types when balancing the non-zero elements across tasklets of a DPU.

We draw three findings. First, we find that even though a DPU supports 24 tasklets, SpMV execution typically scales up to 16 tasklets, since the DPU pipeline is fully utilized. In `de1aunay_n13` matrix, CSR . nnz scales up to 24 tasklets. In this matrix, when using 16 tasklets, performance of the CSR . nnz scheme is limited by memory accesses: *only* one tasklet processes  $6 \times$  more rows than the



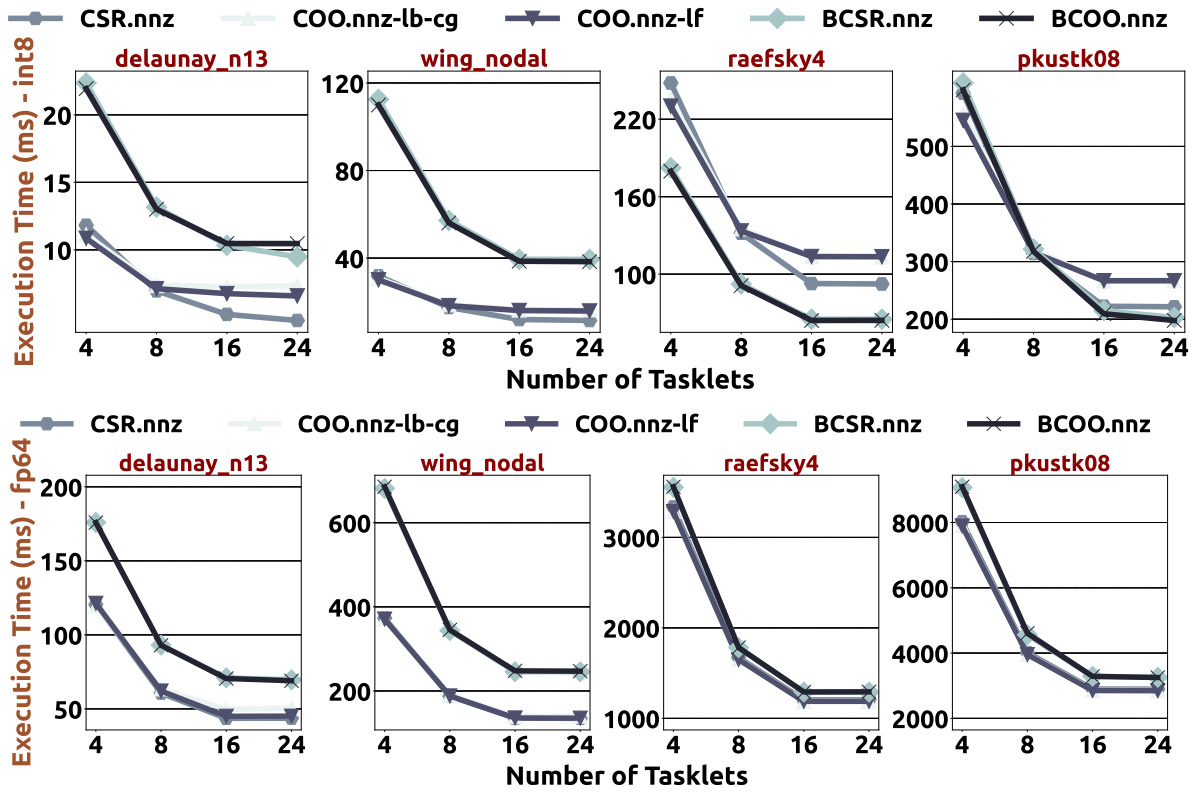


Figure 5.10: Scalability of all compressed formats for the int8 (top graphs) and fp64 (bottom graphs) data types as the number of tasklets of a single DPU increases.

rest, i.e., it performs  $6 \times$  more memory accesses to fetch elements from the `rowptr[]` array. Thus, as we increase the number of tasklets from 16 to 24, the disparity in the number of rows across tasklets decreases, and the performance of the `CSR.nnz` scheme improves. Second, we observe that for the data types with hardware-supported multiplication operation (e.g., int8 data type), CSR achieves the highest scalability, since it provides a better balance between memory access and computation. In contrast, in the floating point data types (e.g., fp64 data type), the DPU is significantly bottlenecked by the expensive software-emulated multiplication operations, and thus all formats scale similarly. Third, we observe that the BCSR and BCOO formats outperform the CSR and COO formats in matrices that exhibit block pattern (i.e., `raefsky4` and `pkustk08` matrices), only when multiplication is supported by hardware (e.g., int8 data type). This is because they exploit spatial and temporal locality in data memory (i.e., WRAM) in the accesses of the elements of the input vector. Instead, in the fp64 data type, performance is severely bottlenecked by computation, thus the BCSR/BCOO formats perform worse than the CSR/COO formats, since they incur higher indexing costs to discover the positions of the non-zero elements [286, 617].

### OBSERVATION 3:

Block-based formats (e.g., BCSR/BCOO) can provide high performance gains over non-block-based formats (e.g., CSR/COO) in matrices that exhibit block pattern, if the multiplication operation is supported by hardware. Otherwise, the state-of-the-art CSR and COO formats can provide high performance and scalability.



## 5.6 Analysis of SpMV Execution on Multiple DPUs

This section analyzes SpMV execution using multiple DPUs in the UPMEM PIM system using the large matrix data set of Table 5.4.

Section 5.6.1 evaluates the 1D partitioning schemes. Section 5.6.1 evaluates the actual kernel time of SpMV by comparing (a) all load balancing schemes of each compressed matrix format, and (b) the performance of all compressed matrix formats. Section 5.6.1 characterizes end-to-end SpMV execution time of the 1D partitioning technique including the data transfer costs for the input and output vectors.

Section 5.6.2 evaluates the 2D partitioning techniques. Section 5.6.2 presents three characterization studies on (a) performing fine-grained data transfers to transfer the elements of the input and output vectors to/from PIM-enabled memory, (b) the scalability of 2D partitioning techniques to thousands of DPUs, and (c) the number of vertical partitions to perform on the matrix. Section 5.6.2 compares the end-to-end performance of all compressed matrix formats for each of the three types of 2D partitioning techniques. Section 5.6.2 compares the best-performing SpMV implementations of all three types of 2D partitioning techniques.

Section 5.6.3 compares the best-performing (on average across all matrices and data types) SpMV implementations of the 1D and 2D partitioning techniques.

### 5.6.1 Analysis of SpMV Execution Using 1D Partitioning Techniques

We evaluate the 1D partitioning schemes highlighted in bold in Table 5.1. Specifically, for `COO.nnz`, we present the coarse-grained locking (`COO.nnz-lb`) and lock-free (`COO.nnz-lf`) approaches, since the fine-grained locking approach performs similarly with the coarse-grained locking approach, as shown in the previous section (Section 5.5.1). Similarly, for the `BCSR(int8 data type)` and `BCOO` formats, we present only the coarse-grained locking approach, since all synchronization approaches perform similarly (Section 5.5.1). Finally, in all experiments presented henceforth, we use 16 tasklets and load-balance the non-zero elements across tasklets within the DPU, since this load balancing scheme provides the highest performance benefits on average across all matrices and data types, according to our evaluation shown in Section 5.5.

#### Analysis of Kernel Time

We compare the `kernel` time of SpMV achieved by various load balancing schemes for each particular compressed matrix format, and then we compare the `kernel` time of the compressed matrix formats.

**Analysis of Load Balancing Schemes Across DPUs.** Figure 5.11 compares load balancing techniques for each compressed matrix format using 2048 DPUs and the `int32` data type.

We draw four findings. First, we observe that `CSR.nnz` and `COO.nnz-rgn`, i.e., balancing the non-zero elements across DPUs (at row granularity), either outperform or perform similarly to `CSR.row` and `COO.row`, respectively, i.e., balancing the rows across DPUs, except for `hgc` and

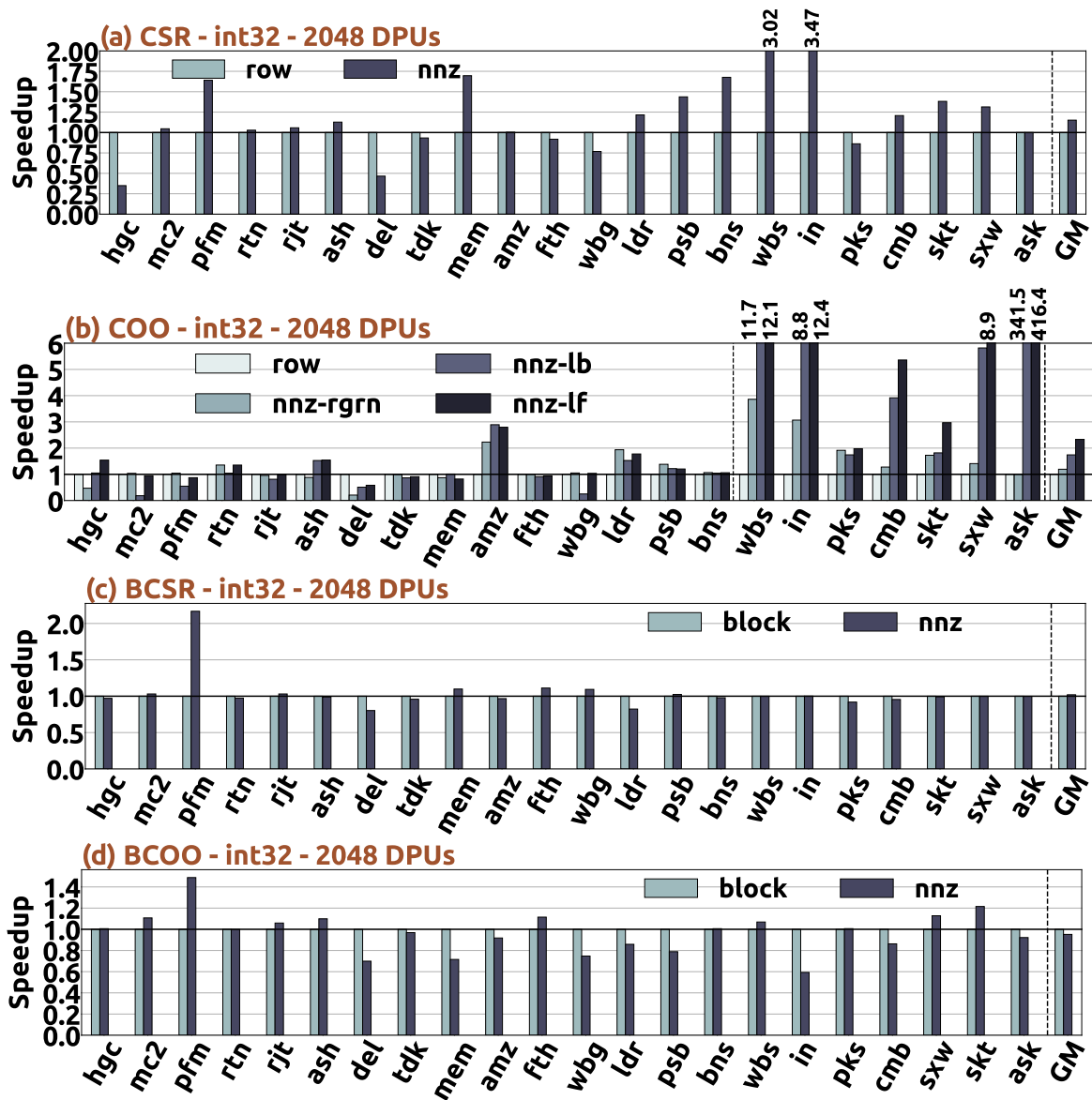


Figure 5.11: Performance comparison of load balancing techniques for each particular compressed format using 2048 DPUs and the int32 data type.

del matrices. In these two matrices, CSR.nnz and COO.nnz-rowgrn incur a high disparity in rows assigned to DPUs, i.e., only one DPU processes  $4\times$  and  $11\times$  more rows than the rest, for hgc and del matrices, respectively. This in turn creates a high disparity in the elements of the output vector processed across DPUs, causing performance to be limited by the DPU that processes the largest number of rows. Thus, we find that adaptive load balancing approaches and selection methods based on the characteristics of each input matrix need to be developed to achieve high performance across all matrices.

#### OBSERVATION 4:

*Adaptive load balancing schemes and selection methods for the balancing scheme on rows/blocks/non-zero elements based on the characteristics of each input matrix need to be developed to provide best performance across all matrices.*

Second, we find that `COO.nnz-lb` and `COO.nnz-lf`, which provide an almost perfect non-zero element balance across DPUs, significantly outperform `COO.row` and `COO.nnz-rgn` in *scale-free* matrices (i.e., from `wbs` to `ask` matrices) by on average  $6.73\times$ . Scale-free matrices have only a few rows, that include a much larger number of non-zero elements compared to the remaining rows of the matrix. Therefore, perfectly balancing the non-zero elements across DPUs provides high performance gains.

**OBSERVATION 5:**

*Perfectly balancing the non-zero elements across PIM cores can provide significant performance benefits in highly irregular, scale-free matrices.*

Third, we find that the lock-free `COO.nnz-lf` scheme outperforms the lock-based `COO.nnz-lb` scheme by  $1.34\times$  on average, and provides high performance benefits when there is a high row imbalance across tasklets within the DPU. When one tasklet processes a much larger number of rows versus the rest, it executes a much larger number of critical sections. As a result, the core pipeline incurs high imbalance in lock acquisitions/releases, causing the lock-based approach to incur high performance overheads in relatively compute-bound DPUs [161, 162].

**OBSERVATION 6:**

*Lock-free approaches can provide high performance benefits over lock-based approaches in PIM architectures, because they minimize synchronization overheads in PIM cores.*

Finally, in the BCSR and BCOO formats, balancing the blocks across DPUs performs similarly (on average across all matrices) to balancing the non-zero elements across DPUs.

To further investigate the performance of the various load balancing schemes, Figure 5.12 compares them using all the data types. We present the geometric mean of all matrices using 2048 DPUs. In the CSR and COO formats, balancing the non-zero elements across DPUs on average outperforms balancing the rows across DPUs by  $1.18\times$  and  $1.20\times$ , respectively. We observe that in the COO format almost perfectly balancing the non-zero elements across DPUs provides significant performance benefits ( $2.55\times$ , averaged across all the data types), compared to balancing the rows, especially when multiplication is not supported by hardware (e.g., for the floating point data types). In contrast, in the BCSR and BCOO formats, balancing the blocks across DPUs performs only slightly better (on average  $2.7\%$  across all the data types) than balancing the non-zero elements.

**Comparison of Compressed Matrix Formats.** Figures 5.13 and 5.14 compare the throughput (in GOperations per second) and the performance, respectively, achieved by various compressed formats using 2048 DPUs and the `int32` data type. For the CSR and COO formats, we select balancing the non-zero elements across DPUs, and for the BCSR and BCOO formats, we select balancing the blocks across DPUs, since these are the best-performing schemes for each format averaged across all matrices and data types (Figure 5.12).

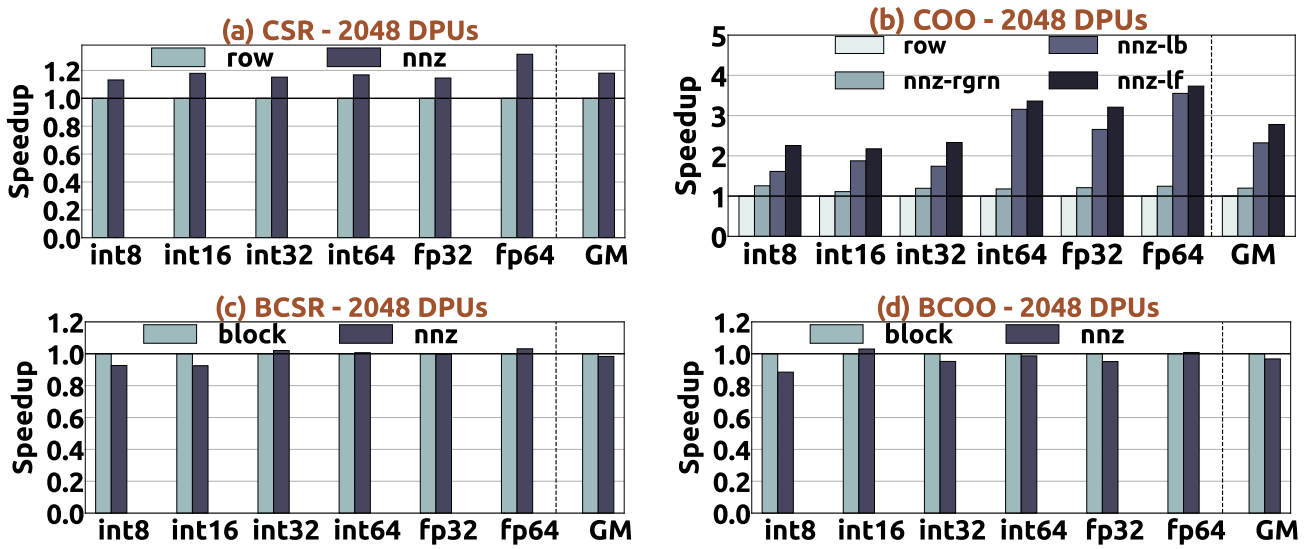


Figure 5.12: Performance comparison of load balancing techniques for each data type using 2048 DPUs.

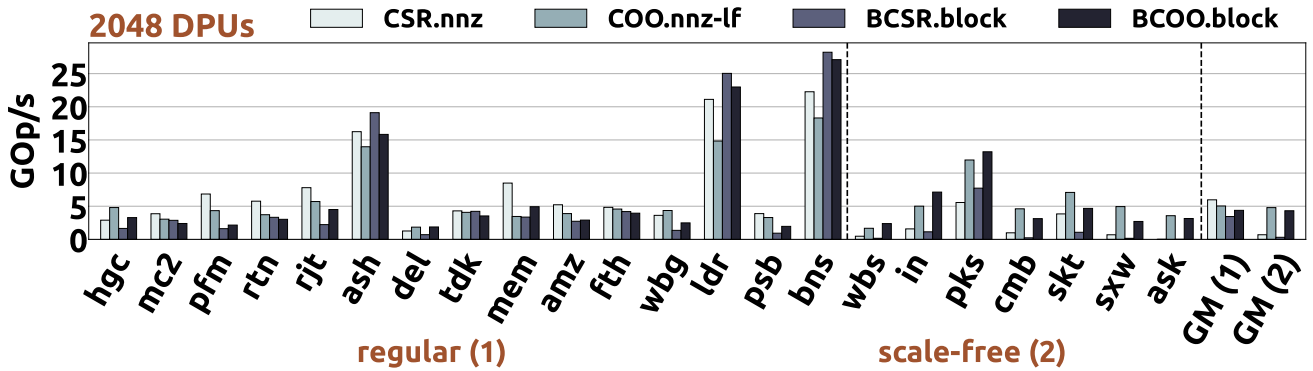


Figure 5.13: Throughput of various compressed formats using 2048 DPUs and the int32 data type.

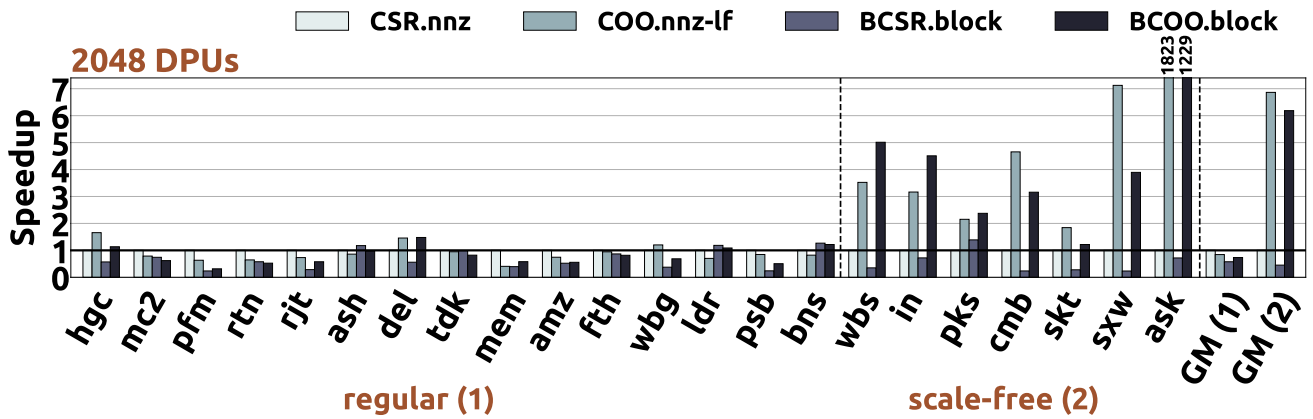


Figure 5.14: Performance comparison of various compressed formats using 2048 DPUs and the int32 data type. Performance is normalized to that of CSR . nnz.

We draw four findings. First, matrices that exhibit block pattern (almost all non-zero elements of the matrix fit in dense sub-blocks), i.e., ash, ldr, bns, pks matrices, have the highest throughput, since they leverage higher data locality compared to matrices with non-block pattern. Second, in scale-free matrices, the COO and BCOO formats significantly outperform the CSR and BCSR formats by  $6.94\times$  and  $13.90\times$ , respectively. This is because they provide better non-zero element balance

across DPUs. In the CSR and BCSR formats, the non-zero element balance is limited to be performed at row and block-row granularity, respectively, causing performance to be limited by the DPU that processes the largest number of non-zero elements. Third, we observe that the BCOO format can outperform the CSR format even in *non-blocked* scale-free matrices. Fourth, we find that when the CSR and BCSR formats provide sufficient non-zero element balance across DPUs, i.e., in many regular matrices such as `rtn`, `tdk`, `amz`, and `fth`, they can outperform the COO and BCOO formats, respectively.

#### OBSERVATION 7:

In *scale-free* matrices, the COO and BCOO formats significantly outperform the CSR and BCSR formats, because they provide higher non-zero element balance across PIM cores.

### Analysis of End-To-End SpMV Execution

Figure 5.15 shows the end-to-end execution time of 1D-partitioned kernels using 2048 DPUs and the `int32` data type. The times are broken down into (i) the time for CPU to DPU transfer to load the input vector into DRAM banks (`load`), (ii) the kernel time on DPUs (`kernel`), (iii) the time for DPU to CPU transfer to retrieve the results for the output vector (`retrieve`), and (iv) the time to merge partial results on the host CPU cores (`merge`).

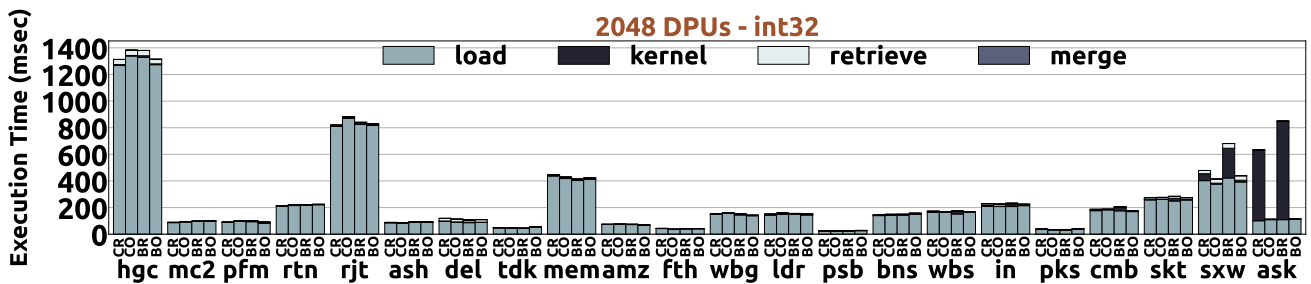


Figure 5.15: Total execution time when using 2048 DPUs and the `int32` data type for CR: CSR . `nnz`, CO: COO . `nnz - 1f`, BR: BCSR . `block` and BO: BCOO . `block` kernels.

We draw four findings. First, the `load` data transfers constitute more than 90% of the total execution time, because the input vector is replicated and broadcast into each DPU, causing a large number of bytes to be transferred through the narrow off-chip memory bus. An exception is in the CSR and BCSR formats for `sxw`, `ask` matrices, which include one very dense row, and thus `kernel` time is highly bottlenecked by one DPU that processes a significantly larger number of non-zero elements than the rest. Second, the `kernel` time constitutes on average only 4.3% of the total execution time, since SpMV is effectively parallelized to thousands of DPUs. Third, the `retrieve` data transfers constitute on average 3.4% of the total execution time, because the output vector is split across DPUs. Fourth, the `merge` time on the host CPU is negligible (less than 1% of the total execution time), since only a few partial results for the elements of the output vector are merged by the host CPU cores in the 1D partitioning techniques.

**OBSERVATION 8:**

The end-to-end performance of the 1D partitioning techniques is severely bottlenecked by the data transfer costs to replicate and broadcast the whole input vector into *each* DRAM bank of PIM cores, which takes place through the narrow off-chip memory bus.

To further investigate on the costs to the load input vector into all DRAM banks of PIM-enabled memory, we present in Figure 5.16 the total execution time achieved by COO . nnz - 1 f when varying (a) the data type using 2048 DPUs (normalized to the experiment for the int8 data type), and (b) the number of DPUs for the int32 data type (normalized to 64 DPUs).

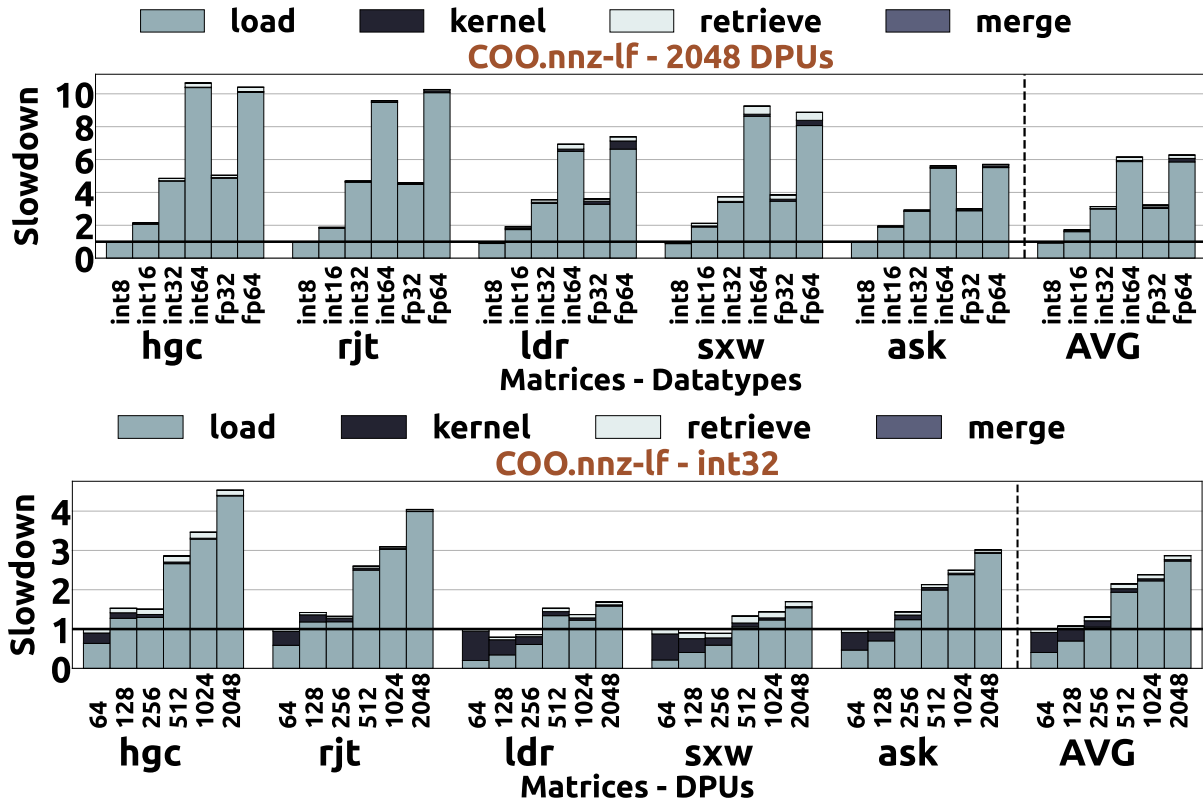


Figure 5.16: End-to-end execution time breakdown achieved by COO . nnz - 1 f when varying (a) the data type using 2048 DPUs (normalized to the experiment for the int8 data type), and (b) the number of DPUs for the int32 data type (normalized to 64 DPUs).

We draw two conclusions. First, the load data transfer costs increase proportionally to the number of bytes of the data type, and still dominate performance even for the data type with the smallest memory footprint (int8). Second, the load data transfer costs and the associated memory footprint for the input vector increase proportionally to the number of DPUs used, and thus the best end-to-end performance is achieved using only a small portion of the available DPUs on the system.

**OBSERVATION 9:**

SpMV execution of the 1D-partitioned schemes cannot scale up to a large number of PIM cores due to high data transfer overheads to copy the input vector into *each* DRAM bank of PIM-enabled memory.

## 5.6.2 Analysis of SpMV Execution Using 2D Partitioning Techniques

We evaluate the 2D-partitioned kernels highlighted in bold in Table 5.1. Specifically, for the COO format we use the lock-free approach, and for the BCSR (in the int8 data type) and BCOO formats we use the coarse-grained locking approach. In the *equally-wide* and *variable-sized* techniques, for the BCSR and BCOO formats we balance the blocks across DPUs of the same vertical partition, since doing so performs slightly better than balancing the non-zero elements, as explained in Section 5.6.1. In all experiments, we balance the non-zero elements across 16 tasklets within a single DPU.

### Sensitivity Studies on 2D Partitioning Techniques

We present three characterization studies on the 2D partitioning techniques. First, we evaluate the performance of fine-grained data transfers from/to PIM-enabled memory for the input and output vectors. Second, we evaluate the scalability of the 2D partitioning techniques to thousands of DPUs. Finally, we explore performance implications on the number of vertical partitions used in the 2D-partitioned kernels.

**Analysis of Fine-Grained Data Transfers.** The UPMEM API [615] has the limitation that *the transfer sizes from/to all DRAM banks involved in the same parallel transfer need to be the same*. The UPMEM API provides *parallel data transfers* either to all DPUs of all ranks (henceforth referred to as *coarse-grained* transfers), or at rank granularity, i.e., to 64 DPUs of the same rank (henceforth referred to as *fine-grained* transfers). In the first case, parallel data transfers are performed to all DPUs used at once, padding with empty bytes at the granularity of *all* DPUs used, e.g., 2048 DPUs in Figure 5.17. In the latter case, programmers iterate over the ranks of PIM-enabled DIMMs, and for *each* rank perform parallel data transfers to the 64 DPUs of the same rank padding with empty bytes at the granularity of 64 DPUs.

In SpMV execution, for the *equally-wide* and *variable-sized* techniques the heights and widths of 2D tiles vary, and thus padding with empty bytes is necessary for the `load` and `retrieve` data transfers of the elements of the input and output vector, respectively. Figure 5.17 compares coarse-grained data transfers, i.e., performing parallel transfers to all 2048 DPUs at once, with fine-grained data transfers, i.e., iterating over the ranks and for each rank performing parallel transfers to the 64 DPUs of the same rank. We evaluate both the *equally-wide* and *variable-sized* techniques using the COO format and with 2 and 32 vertical partitions. Please see Appendix 8.1.2 for all matrices.

We draw two findings. First, when the number of vertical partitions is small, e.g., 2 vertical partitions, the disparity in widths across tiles in the *variable-sized* scheme is low. Thus, BT only slightly outperforms BY by 1% on average, since in BY *only* a small amount of padding is added on the `load` data transfers of the input vector. In contrast, the disparity in heights across tiles in the *equally-wide* and *variable-sized* schemes is high. Thus, RY and BY significantly outperform RC and BC by an average of  $1.68\times$  and  $1.60\times$ , respectively. This is because fine-grained transfers to retrieve the elements of the output vector significantly decrease the amount of bytes transferred from PIM-enabled memory to host CPU over coarse-grained transfers. Second, when the number of vertical partitions is large, e.g., 32 vertical partitions, the disparity in heights across tiles in the

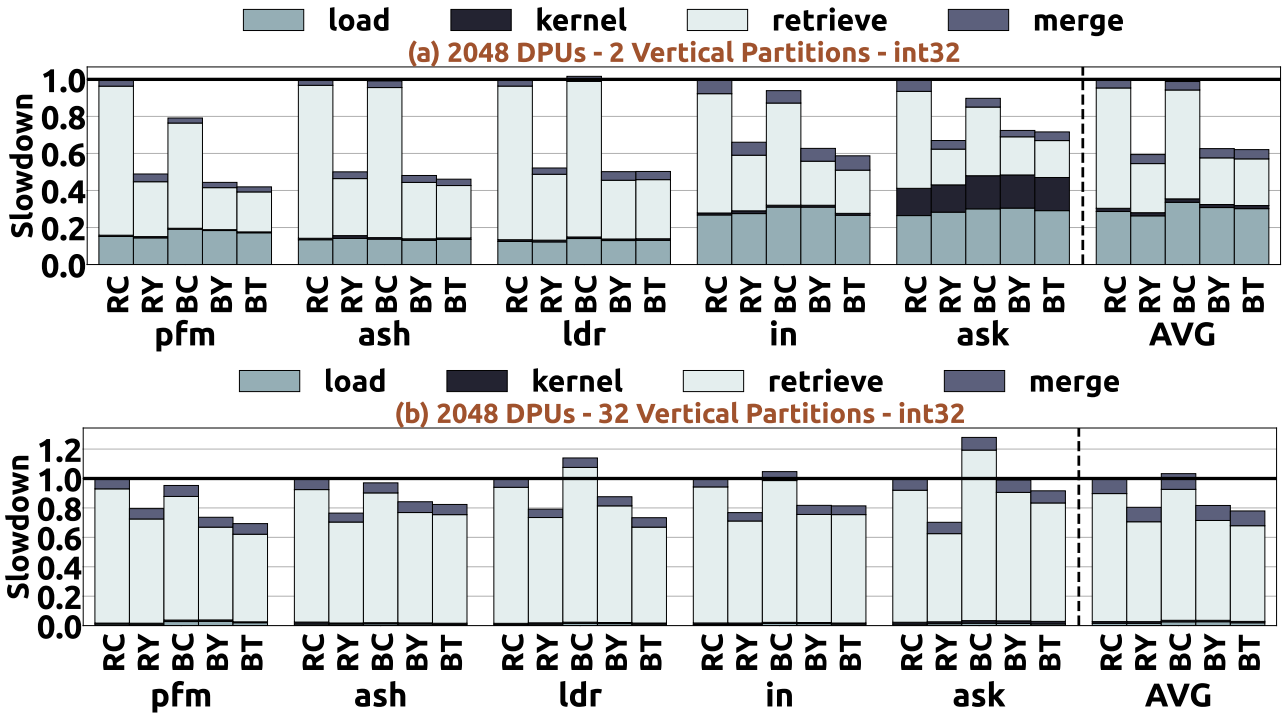


Figure 5.17: Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 (left) and 32 (right) vertical partitions. Performance is normalized to that of the RC scheme.

*equally-wide* and *variable-sized* schemes is lower compared to when the number of vertical partitions is small. Thus, RY and BY provide smaller performance benefits over RC and BC (on average  $1.24\times$  and  $1.22\times$ , respectively), respectively, compared to a small number of vertical partitions. In contrast, the disparity in heights across tiles in the *equally-wide* and *variable-sized* schemes is higher compared to when the number of vertical partitions is small. Thus, BT outperforms BY by 4.7% on average. Overall, we conclude that fine-grained data transfers (i.e., at rank granularity in the UPMEM PIM system) can significantly improve performance in the *equally-wide* and *variable-sized* schemes.

#### OBSERVATION 10:

*Fine-grained parallel transfers in the equally-wide and variable-sized 2D partitioning techniques, i.e., minimizing the amount of padding with empty bytes in parallel data transfers to/from PIM-enabled memory, can provide large performance gains.*

**Scalability of the 2D Partitioning Techniques.** We analyze scalability with the number of DPUs for the 2D partitioning techniques. Figures 5.18, 5.19 and 5.20 compare the performance of the *equally-sized*, *equally-wide* and *variable-sized* schemes, respectively, using the COO format and the int32 data type, as the number of DPUs increases.

We draw two findings. First, the *equally-sized* scheme (i.e., DCOO) achieves high scalability with a large number of vertical partitions. The kernel time of *equally-sized* scheme is mainly limited by the DPU (or a few DPUs) that processes the largest number of non-zero elements. With a large num-



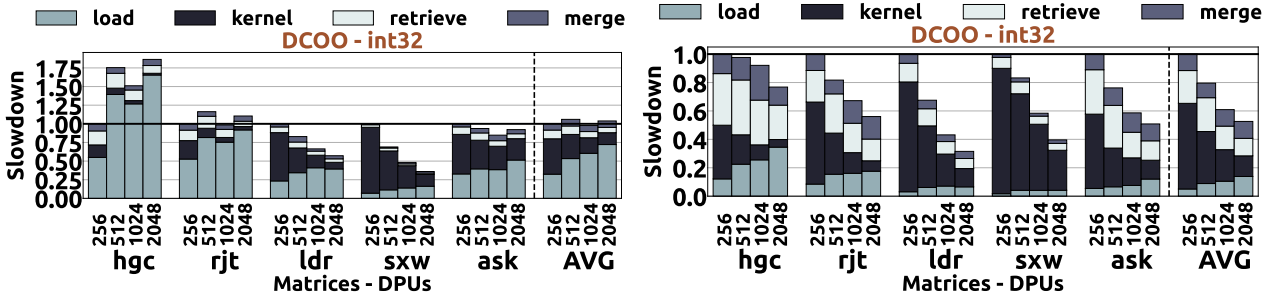


Figure 5.18: Execution time breakdown of *equally-sized* partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

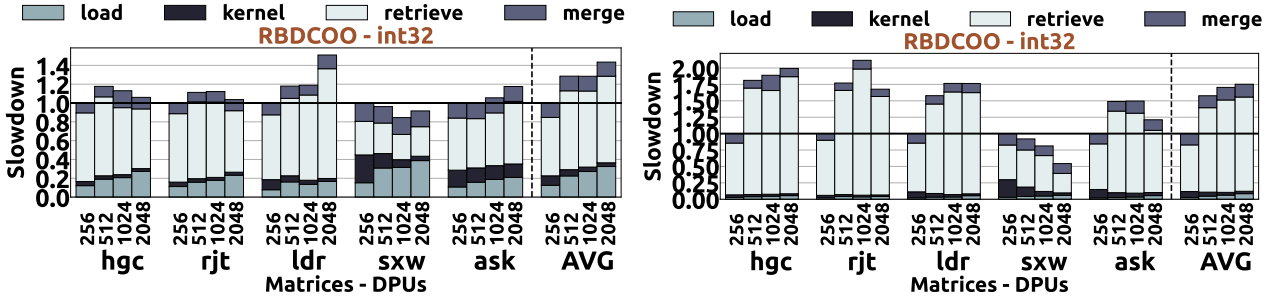


Figure 5.19: Execution time breakdown of *equally-wide* partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

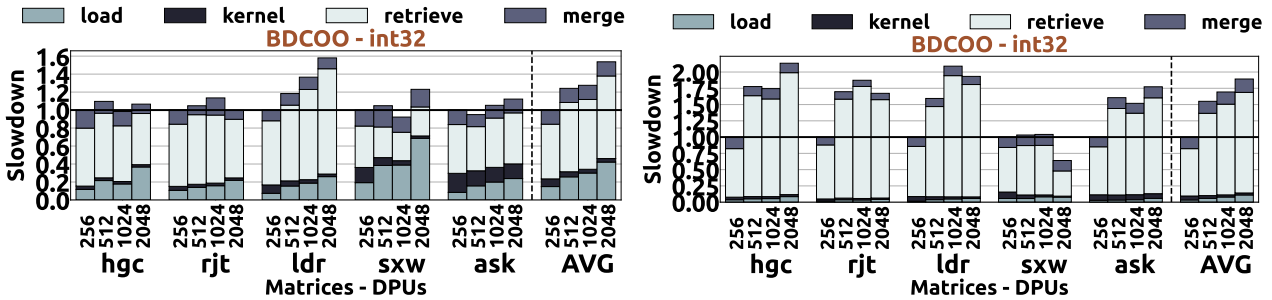


Figure 5.20: Execution time breakdown of *variable-sized* partitioning technique of the COO format using 4 (left) and 16 (right) vertical partitions when varying the number of DPUs used for the int32 data type. Performance is normalized to that with 256 DPUs.

ber of *static* vertical partitions, the non-zero element disparity across DPUs is high, i.e., the `kernel` time is highly bottlenecked by the DPU that processes the largest number of non-zero elements. As a result, increasing the number of DPUs improves performance by decreasing the `kernel` time via better non-zero element balance across DPUs.

#### OBSERVATION 11:

The `kernel` time in the *equally-sized* schemes is limited by the PIM core (or a few PIM cores) assigned to the 2D tile with the largest number of non-zero elements.

Second, we observe that the *equally-wide* and *variable-sized* schemes (i.e., RBDCOO and BDCOO) are severely bottlenecked by `retrieve` data transfer costs (a large number of partial results is created on PIM cores), and thus they are difficult to scale up to thousands of DPUs. Moreover, when

the number of vertical partitions is high, the disparity in heights of the tiles is high. Thus, as the number of DPUs increases, the amount of padding needed in `retrieve` data transfers becomes very large, causing significant performance degradation.

#### OBSERVATION 12:

The scalability of the *equally-wide* and *variable-sized* schemes to a large number of PIM cores is severely limited by large data transfer overheads to retrieve partial results for the elements of the output vector from the DRAM banks of PIM-enabled memory to the host CPU via the narrow memory bus.

**Effect of the Number of Vertical Partitions.** In all experiments presented henceforth, we perform fine-grained data transfers (at rank granularity, i.e., 64 DPUs in the UPMEM PIM system) in the 2D partitioning schemes. Figure 5.21 evaluates performance implications on the number of vertical partitions performed in 2D-partitioned kernels. We use the COO format and vary the number of vertical partitions from 1 to 32, in steps of multiple of 2. We draw four findings.

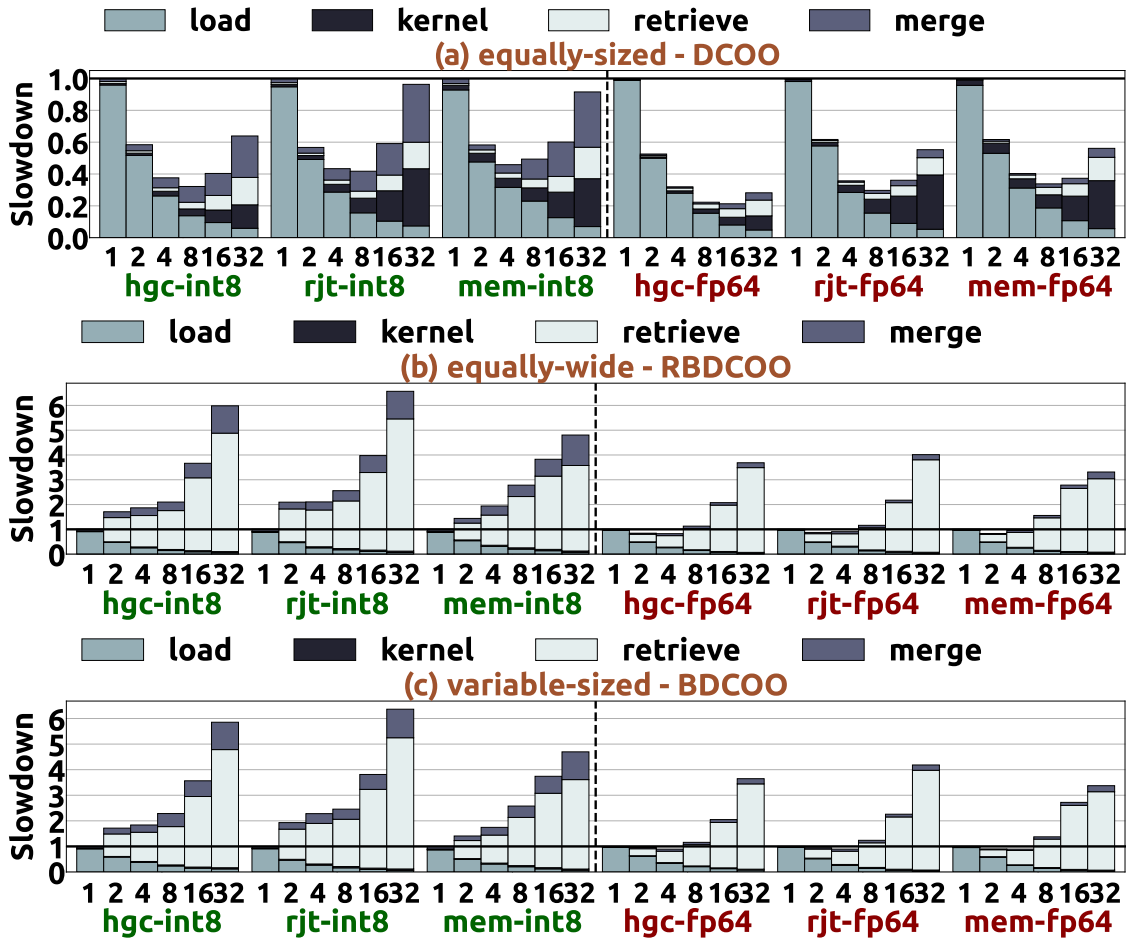


Figure 5.21: Execution time breakdown of 2D partitioning schemes using the COO format and 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int8 and fp64 data types. Performance is normalized to the performance of the experiment with 1 vertical partition.

First, in the *equally-sized* scheme, as the number of vertical partitions increases, kernel time increases, if there is *no* dense row in the matrix. This is because the disparity in the non-zero elements

across 2D tiles increases as the number of vertical partitions increases. Thus, performance is limited by one DPU or a few DPUs that process the largest number of non-zero elements.

**OBSERVATION 13:**

As the number of vertical partitions increases, the *equally-sized* 2D partitioning scheme typically increases the non-zero element disparity across PIM cores (unless there is one dense row on the matrix), thereby increasing the kernel time.

Second, as the number of vertical partitions increases, `retrieve` data transfer costs and `merge` time increase. This is because the partial results created for the output vector increase proportionally with the number of vertical partitions. The performance overheads of `retrieve` data transfer costs are highly affected by the characteristics of the underlying hardware (e.g., the bandwidth provided on I/O channels of the memory bus between host CPU and PIM-enabled DIMMs). Similarly, the performance cost of the `merge` step depends on the hardware characteristics of the host CPU (e.g., the number of the CPU cores, the available hardware threads, microarchitecture of CPU cores). We refer the reader to Appendix 8.1.3 for a comparison of SpMV execution using two different UPMEM PIM systems with different hardware characteristics (Table 8.1).

Third, we find that in the *equally-wide* and *variable-sized* schemes, there is high disparity in heights of 2D tiles, and as a result on the number of partial results created across DPUs. Even with fine-grained parallel `retrieve` data transfers at rank granularity, the amount of padding needed in the *equally-wide* and *variable-sized* schemes is at 88.6% and 88.0%, respectively, causing high bottlenecks in the narrow memory bus. Therefore, in PIM systems that do not support very fine-grained parallel transfers to gather results from PIM-enabled memory to the host CPU *at DRAM bank granularity*, execution is highly limited by the amount of padding performed in `retrieve` data transfers, which can be very large in irregular workloads [1,4,18,42,103,150,161,162,286,289–293,326,546,587] like the SpMV kernel.

**OBSERVATION 14:**

The *equally-wide* and *variable-sized* 2D partitioning schemes require fine-grained parallel transfers *at DRAM bank granularity* to be supported by the PIM system, i.e., zero padding in *parallel retrieve* data transfers from PIM-enabled memory to the host CPU, to achieve high performance.

Fourth, we find that the number of vertical partitions that provides the best performance depends on the sparsity pattern of the input matrix, the data type, and the underlying hardware parameters (e.g., number of PIM cores, off-chip memory bus bandwidth, transfer latency costs between main memory and PIM-enabled memory, characteristics and microarchitecture of the host CPU cores that perform the `merge` step). For example, with the `int8` data type, DCOO performs best for `hgc` and `mem` matrices with 8 and 4 vertical partitions, respectively. Instead, with the `fp64` data type, DCOO performs best for `hgc` and `mem` matrices with 16 and 8 vertical partitions, respectively. We refer the

reader to Appendix 8.1.3 for a characterization study on the number of vertical partitions to perform in the 2D-partitioned kernels using two UPMEM PIM systems with different hardware characteristics. As we demonstrate in Appendix 8.1.3, the number of vertical partitions that provides best performance on SpMV varies across the two different UPMEM PIM platforms. In this work, we leave for future work the exploration of selection methods for the number of vertical partitions that provide best SpMV execution. Overall, based on our analysis we conclude that the parallelization scheme that achieves the best performance in SpMV depends on both the input sparse matrix and the hardware characteristics of the PIM system.

#### OBSERVATION 15:

There is *no one-size-fits-all* parallelization approach for SpMV in PIM systems, since the performance of each parallelization scheme depends on the characteristics of the input matrix and the underlying PIM hardware.

### Analysis of Compressed Formats

We compare the performance achieved by various compressed matrix formats for each of the three types of the 2D partitioning technique. The goal of this experiment is to find the best-performing compressed format for each 2D partitioning technique. Figures 5.22, 5.23, and 5.24 compare the performance of compressed matrix formats for the *equally-sized*, *equally-wide* and *variable-sized* 2D partitioning techniques, respectively. We use 2048 DPUs and the int32 data type having 4 vertical partitions. See Appendix 8.1.4 for the complete evaluation on all large sparse matrices.

We draw two findings. First, as already explained, `kernel` time of the *equally-sized* scheme is limited by the DPU (or a few DPUs) assigned to the 2D tile with the largest number of non-zero elements. In scale-free matrices (e.g., `in` and `ask`), the disparity in the non-zero elements across 2D tiles is higher than in regular matrices (e.g., `pfg` and `bns`), causing `kernel` time to be a larger portion of the total execution time. Second, we find that the CSR and BCSR formats perform worse than the COO and BCOO formats, especially in the *equally-wide* and *variable-sized* schemes, due to higher `kernel` times. In the CSR and BCSR formats, data partitioning across DPUs and/or across

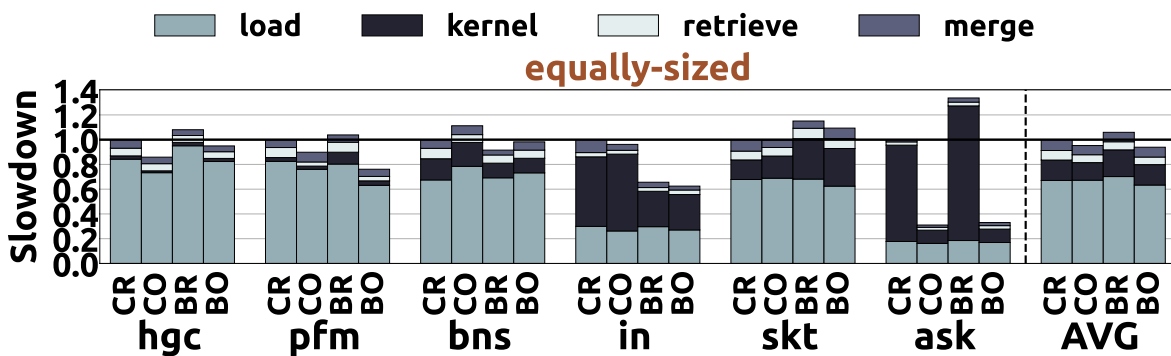


Figure 5.22: End-to-end execution time breakdown of the *equally-sized* 2D partitioning technique for CR: DCSR, CO: DCOO, BR: DBCSR and BO: DBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR.

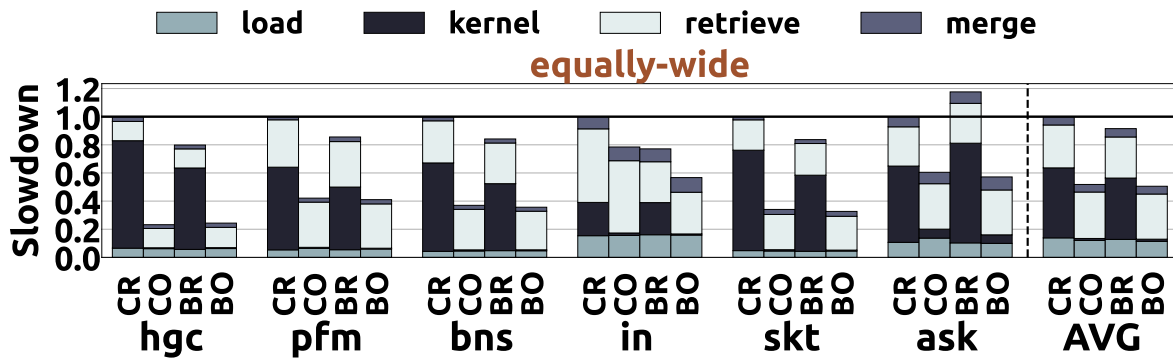


Figure 5.23: End-to-end execution time breakdown of the *equally-wide* 2D partitioning technique for CR: RBDCSR, CO: RBDCOO, BR: RDBCSR and BO: RDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR.

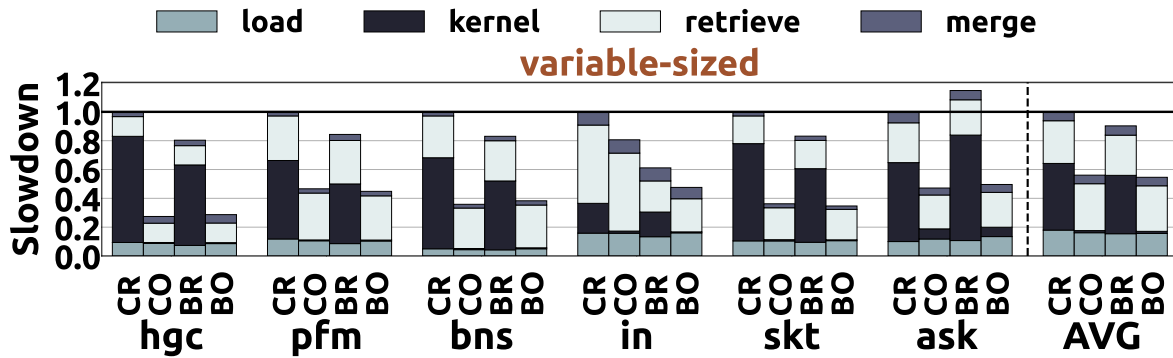


Figure 5.24: End-to-end execution time breakdown of the *variable-sized* 2D partitioning technique for CR: BDCSR, CO: BDCOO, BR: BDBCSR and BO: BDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR.

tasklets within a DPU is performed at row and block-row granularity, respectively. Thus, the CSR and BCSR formats can cause higher non-zero element imbalance across processing units compared to the COO and BCOO formats. Overall, the COO and BCOO formats outperform the CSR and BCSR formats by  $1.59 \times$  and  $1.53 \times$  (averaged across all three types of 2D partitioning techniques), respectively.

#### OBSERVATION 16:

The compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory. Thus, it affects the load balance across PIM cores with corresponding performance implications. Overall, the COO and BCOO formats outperform the CSR and BCSR formats, because they provide higher non-zero element balance across PIM cores.

### Comparison of 2D Partitioning Techniques

We compare the best-performing SpMV implementations of all 2D partitioning schemes, i.e., using the COO and BCOO formats. Figures 5.25 and 5.26 compare the throughput (in GOperations per second) and the performance, respectively, of DCOO, DBCOO, RBDCOO, RDBCOO, BDCOO, BDBCOO schemes using 2048 DPUs and the int32 data type. For each implementation, we vary the number of vertical partitions from 2 to 32, in steps of multiple of 2, and select the best-performing execution

throughput.

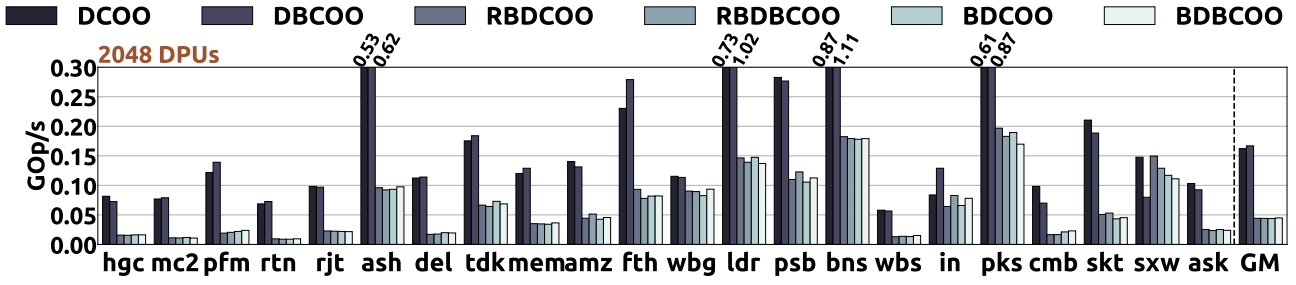


Figure 5.25: Throughput of 2D partitioning techniques using the COO and BCOO formats, 2048 DPUs and the int32 type.

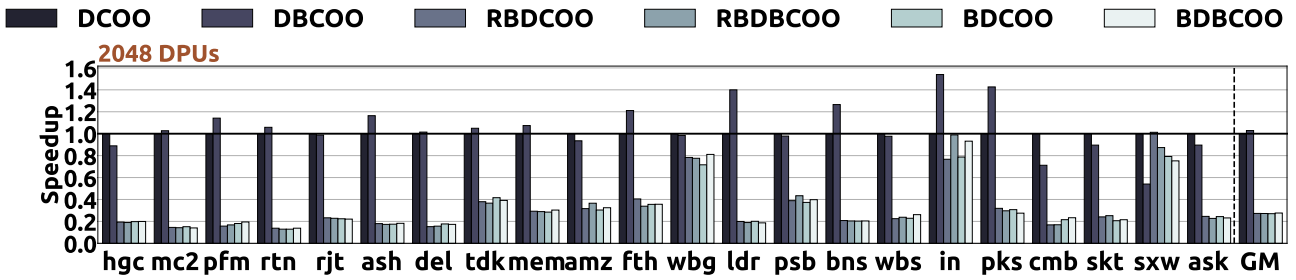


Figure 5.26: Performance comparison of 2D partitioning techniques using the COO and BCOO formats, 2048 DPUs and the int32 type. Performance is normalized to that of DCOO.

We draw two conclusions. First, similarly to 1D-partitioned kernels, matrices that exhibit block pattern (e.g., *ash*, *ldr*, *bns*, *pks*) have the highest throughput (Figure 5.25). Second, the *equally-wide* and *variable-sized* schemes perform similarly, i.e., their performance varies only by  $\pm 1.1\%$  on average. Even though the *variable-sized* technique can improve the non-zero element balance across DPUs, and thus kernel time, compared to the *equally-wide* technique, the total execution time does not improve. In the UPMEM PIM system, performance of both techniques is severely bottlenecked by data transfer overheads due to a large amount of padding needed to retrieve results from PIM-enabled memory to the host CPU. Third, we find that the *equally-sized* technique outperforms the *equally-wide* and *variable-sized* techniques by  $3.71\times$  on average, because it achieves lower data transfer overheads. The *equally-wide* and *variable-sized* techniques provide near-perfect non-zero element balance across DPUs, but they significantly increase the retrieve data transfer costs due to the large amount of padding with empty bytes performed. As a result, we recommend software designers to explore *relaxed* load balancing schemes, i.e., schemes that trade off computation balance across PIM cores for lower amounts of data transfer.

### 5.6.3 Comparison of 1D and 2D Partitioning Techniques

We compare the throughput (in GOperations per second) and the performance of the best-performing 1D- and 2D-partitioned kernels in Figures 5.27 and 5.28, respectively. For 1D partitioning, we use the lock-free COO (`COO.nnz-1f`) and coarse-grained locking BCOO (`BCOO.block`) kernels. For each matrix, we vary the number of DPUs from 64 to 2528, and select the best-performing end-to-end execution throughput. For 2D partitioning, we use the *equally-sized* COO (DCOO) and BCOO

(DBCOO) kernels with 2528 DPUs. For each matrix, we vary the number of vertical partitions from 2 to 32 (in steps of multiple of 2), and select the best-performing end-to-end execution throughput. The numbers shown over each bar of Figure 5.27 present the number of DPUs that provide the best-performing end-to-end execution throughput for each input-scheme combination. Please see Appendix 8.1.5 for a performance comparison of the best-performing SpMV kernels on two UPMEM PIM systems with different hardware characteristics.

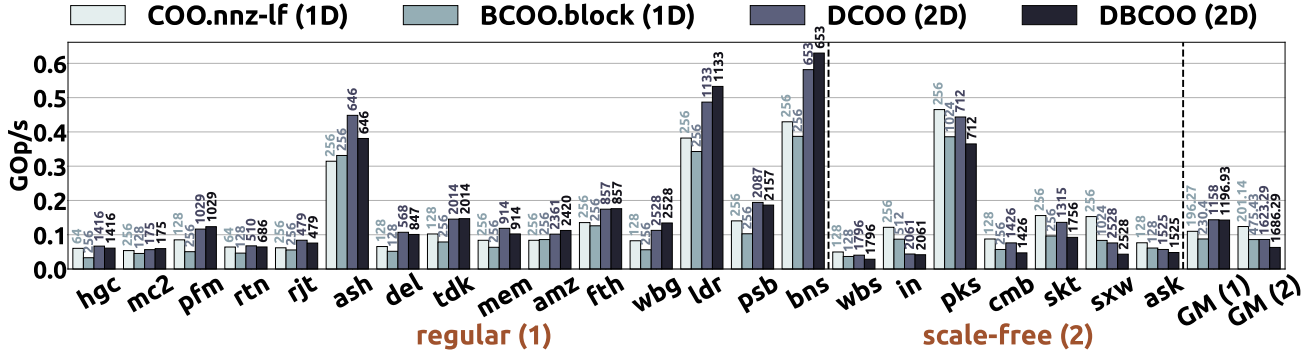


Figure 5.27: Throughput of the best-performing 1D- and 2D-partitioned kernels for the fp32 data type.

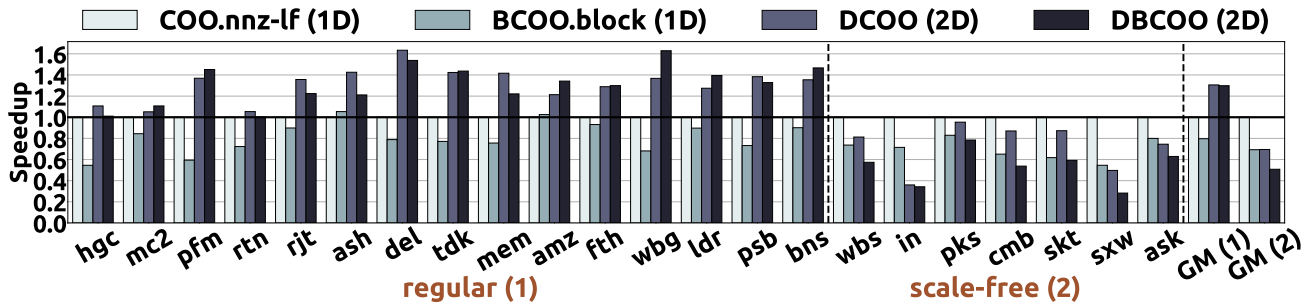


Figure 5.28: Performance comparison of the best-performing 1D- and 2D-partitioned kernels for the fp32 data type. Performance is normalized to that of COO . nnz - 1 f.

We draw two findings. First, we find that best performance is achieved using a much smaller number of DPUs than the available DPUs on the system. In the 1D-partitioned kernels (i.e., COO . nnz - 1 f and BCOO . block), replicating the input vector into a large number of DPUs significantly increases the load data transfer costs. Thus, best performance is achieved using 253 DPUs on average across all matrices. In the 2D-partitioned kernels (i.e., DCOO and DBCOO), creating *equally-sized* 2D tiles leads to a large disparity in non-zero element count across tiles, causing many tiles to be empty, i.e., without *any* non-zero element. Thus, best performance is achieved using 1329 DPUs on average across all matrices, since DPUs associated with empty tiles are idle.

#### OBSERVATION 17:

Expensive data transfers to PIM-enabled memory performed via the narrow memory bus impose significant performance overhead to end-to-end SpMV execution. Thus, it is hard to fully exploit all available PIM cores of the system.

Second, we observe that in regular matrices, the 2D-partitioned kernels outperform the 1D-partitioned kernels by  $1.45\times$  on average. This is because the 2D-partitioned kernels use a larger



number of DPUs, and thus their kernel times are lower. In contrast, in scale-free matrices, the 1D-partitioned kernels outperform the 2D-partitioned kernels by  $1.41\times$  on average. This is because the *equally-sized* 2D technique significantly increases the non-zero element disparity across DPUs, i.e., kernel time is bottlenecked by only one DPU or a few DPUs that process a much larger number of non-zero elements compared to the rest.

#### OBSERVATION 18:

In *regular* matrices, 2D-partitioned kernels outperform 1D-partitioned kernels, since the former provide a better trade-off between computation and data transfer overheads. In contrast, in *scale-free* matrices, 2D-partitioned kernels perform worse than 1D-partitioned kernels, since the former's performance is limited by one DPU or a few DPUs that process the largest number of non-zero elements.

## 5.7 Comparison with CPUs and GPUs

We compare SpMV execution on the UPMEM PIM architecture to a state-of-the-art CPU and a state-of-the-art GPU in terms of performance and energy consumption. Our goal is to quantify the potential of the UPMEM PIM architecture on the widely used memory-bound SpMV kernel.

We compare the UPMEM PIM system with 2528 DPUs to an Intel Xeon CPU [618] and an NVIDIA Tesla V100 GPU [619], the characteristics of which are shown in Table 5.5. We use peakperf [623] and stream [624] for CPU and GPU systems to calculate the peak performance, memory bandwidth, and Thermal Design Power (TDP). For the UPMEM PIM system, we estimate the peak performance as  $Total\_DPUs * AT$ , where the arithmetic throughput (AT) is calculated for the multiplication operation in Appendix 8.2 (Figure 8.12), the total bandwidth as  $Total\_DPUs * Bandwidth\_DPU$ , where the  $Bandwidth\_DPU$  is 700 MB/s [157, 161, 162], and TDP as  $(Total\_DPUs / DPU\_per\_chip) * 1.2W/chip$  from prior work [157, 161, 162].

System	Process Node	Total Cores	Frequency	Peak Performance	Memory Capacity	Total Bandwidth	TDP
Intel Xeon 4110 CPU [618]	14 nm	2x8 x86 cores (2x16 threads)	2.1 GHz	660 GFLOPS	128 GB	23.1 GB/s	2x85 W
NVIDIA Tesla V100 [619]	12 nm	5120 CUDA cores	1.25 GHz	14.13 TFLOPS	32 GB	897 GB/s	300 W
PIM System	2x nm	2528 DPUs	350 MHz	4.66 GFLOPS	159 GB	1.77 TB/s	379 W

Table 5.5: Evaluated CPU, GPU, and UPMEM PIM Systems.

### 5.7.1 Performance Comparison

For the CPU system, we use the optimized CSR kernel from the TACO library [104]. For the GPU system, we use the CSR5 CUDA [625, 626] for the int32 data type and cuSparse [627] for the other data types. For the UPMEM PIM system, we use the lock-free COO 1D-partitioned kernel (**COO**. nnz - 1f) and the *equally-sized* COO 2D-partitioned kernel (**DCOO**). In the former, we run experiments from 64 to 2528 DPUs, and in the latter, we use 2528 DPUs, and vary the number of vertical partitions from 2 to 32, in steps of multiple of 2. In both schemes, we select the best-performing end-to-end



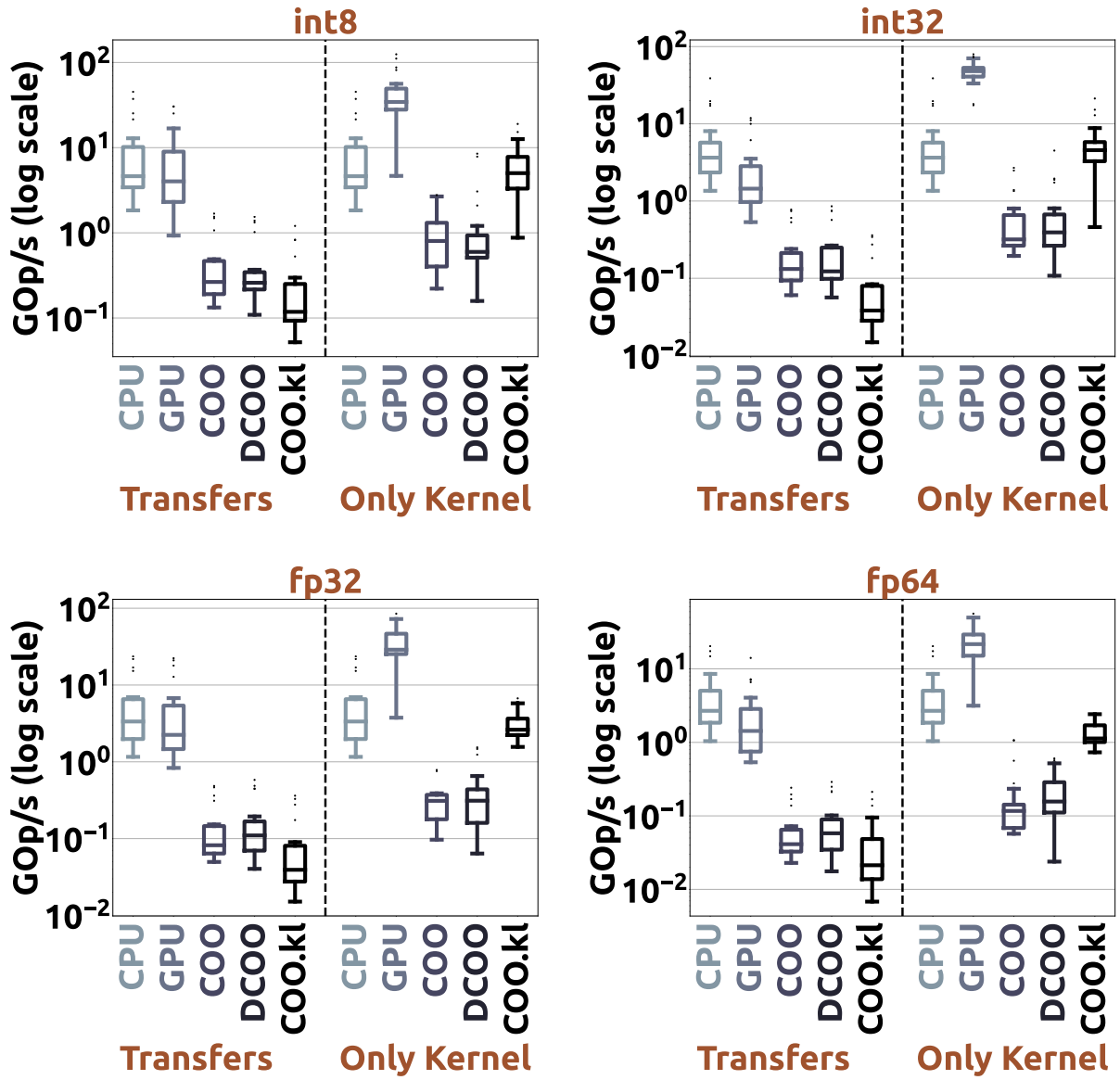


Figure 5.29: Performance comparison between the UPMEM PIM system, Intel Xeon CPU and Tesla V100 GPU on SpMV execution.

execution throughput. We also include the lock-free COO 1D-partitioned kernel using 2528 DPUs, named **COO.kl**, to evaluate SpMV execution using *all* available DPUs of the system.

Figure 5.29 shows the throughput of SpMV (in GOperations per second) in all systems, comparing both the end-to-end execution throughput (i.e., including the `load` and `retrieve` data transfer costs for the input and output vectors in case of the UPMEM PIM and GPU systems), and only the actual kernel throughput (i.e., including the `kernel` time in DPUs and the `merge` time in host CPU for the UPMEM PIM system).

We draw three conclusions. First, when data transfer costs to/from host CPU are included, CPU outperforms both the GPU and UPMEM PIM systems, since data transfers impose high overhead. When only the actual kernel time is considered, GPU performs best, since it is the system that provides the highest computation throughput, e.g., 14.13 TFlops for the fp32 data type. Second, we evaluate the portion of the machine’s peak performance achieved on SpMV in all systems, and observe that SpMV execution on the UPMEM PIM system achieves a much higher fraction of the peak performance

compared to CPU and GPU systems. For the fp32 data type, SpMV achieves on average 0.51% and 0.21% of the peak performance in CPU and GPU, respectively, while it achieves 51.7% of the peak performance in the UPMEM PIM system using the COO.k1 scheme. Achieving a high portion of machine's peak performance is highly desirable, since the software highly exploits the computation capabilities of the underlying hardware. This way, it improves the processor/resource utilization, and the cost of ownership of the underlying hardware. Third, we observe that when all DPUs are used, as in COO.k1, SpMV execution on the UPMEM PIM outperforms SpMV execution on the CPU by  $1.09\times$  and  $1.25\times$  for the int8 and int32 data types, respectively, the multiplication of which is supported by hardware. In contrast, SpMV execution on the UPMEM PIM performs  $1.27\times$  and  $2.39\times$  worse than SpMV execution on the CPU for the fp32 and fp64 data types, the multiplication of which is software emulated in the DPUs of the UPMEM PIM system.

#### OBSERVATION 19:

SpMV execution can achieve a *significantly higher* fraction of the peak performance on real memory-centric PIM architectures compared to that on processor-centric CPU and GPU systems, since PIM architectures greatly mitigate data movement costs.

### 5.7.2 Energy Comparison

For energy measurements, we consider only the actual kernel time in all systems (in the UPMEM PIM we consider the kernel and merge steps of SpMV execution). We use Intel RAPL [628] on the CPU, and NVIDIA SMI [629] on the GPU. For the UPMEM PIM system, we measure the number of cycles, instructions, WRAM accesses and MRAM accesses of each DPU, and estimate energy with energy weights provided by the UPMEM company [338]. Figure 5.30 shows the energy consumption (in Joules) and performance per energy (in (GOp/s)/W) for all systems.

We draw three findings. First, GPU provides the lowest energy on SpMV over the other two systems, since the energy results typically follow the performance results. Second, we find that the 2D-partitioned kernel, i.e., DCOO, consumes more energy than the 1D-partitioned kernels, i.e., COO and COO.k1, due to the energy consumed in the host CPU cores. CPU cores merge a large number of partial results in the 2D-partitioned kernels to assemble the final output vector, thereby increasing the energy consumption. Finally, we find that the 1D-partitioned kernels provide better energy efficiency on SpMV over the CPU system, when the multiplication operation is supported by hardware. Specifically, 1D-partitioned kernels provide  $3.16\times$  and  $4.52\times$  less energy consumption, and  $1.74\times$  and  $1.14\times$  better performance per energy over the CPU system for the int8 and int32 data types, respectively.

#### OBSERVATION 20:

Real PIM architectures can provide high energy efficiency on SpMV execution.

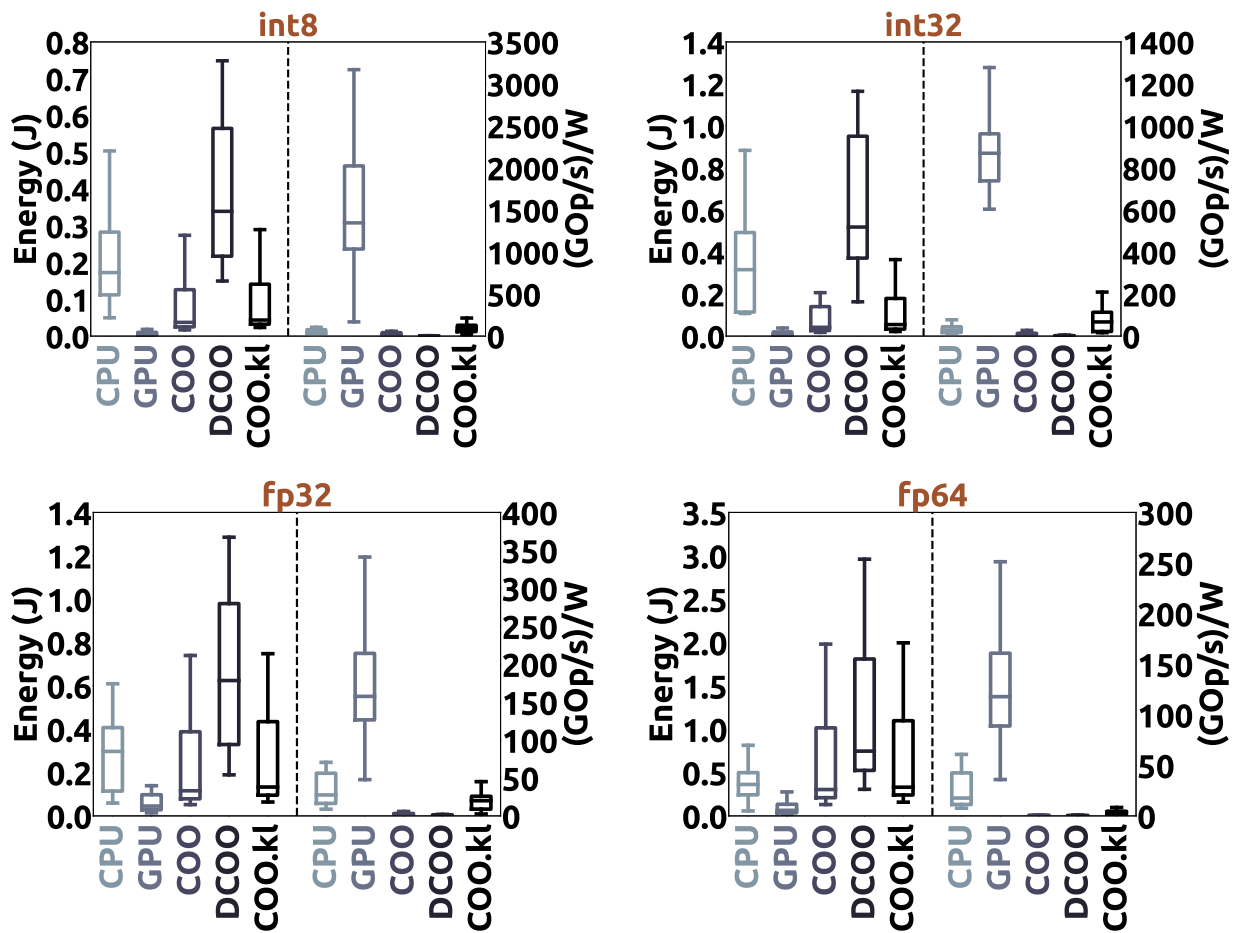


Figure 5.30: Energy comparison between the UPMEM PIM system, Intel Xeon CPU and Tesla V100 GPU on SpMV execution.

### 5.7.3 Discussion

These evaluations are useful for programmers to anticipate how much performance and energy savings memory-centric PIM systems can provide on SpMV over commodity processor-centric CPU and GPU systems. However, our evaluated SpMV kernels do not constitute the best-performing approaches for *all* matrices. Designing methods to select the best-performing SpMV parallelization scheme depending on the particular characteristics of the input matrix would further improve performance and energy savings of SpMV execution on memory-centric PIM systems. Moreover, the UPMEM PIM hardware is still maturing and is expected to run at a higher frequency in the near future (500 MHz instead of 350 MHz) [162, 338]. Hence, SpMV execution on the UPMEM PIM architecture might achieve even higher performance and energy benefits over the results we report in this comparison. Finally, note that our proposed *SparseP* kernels can be adapted and evaluated on other current and future real PIM systems with potentially higher computation capabilities and energy efficiency than the UPMEM PIM system.

## 5.8 Key Takeaways and Recommendations

This section summarizes our key takeaways in the form of recommendations to improve multiple aspects of PIM hardware and software.

**Recommendation #1.** *Design algorithms that provide high load balance across threads of a PIM core in terms of computations, loop control iterations, synchronization points and memory accesses.* Section 5.5 shows that in matrices and formats where the parallelization scheme used causes *high disparity* in the non-zero elements/blocks/rows processed across threads of a PIM core, or the number of lock acquisitions/lock releases/DRAM memory accesses performed across threads, SpMV performance severely degrades in compute-bound DPUs [161, 162]. Therefore, from a programmer's perspective, providing high operation balance across parallel threads is of vital importance in low-area and low-power PIM cores with relatively low computation capabilities [161, 162].

**Recommendation #2.** *Design compressed data structures that can be effectively partitioned across DRAM banks, with the goal of providing high computation balance across PIM cores.* Sections 5.6.1 and 5.6.2 demonstrate that (i) the compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory, and (ii) SpMV execution using the CSR and BCSR formats performs significantly worse than SpMV execution using the COO and BCOO formats. This is because the matrix is stored in row- or block-row-order for the CSR and BCSR formats, respectively, and thus data partitioning across DRAM banks is limited to be performed at row or block-row granularity, respectively, leading to high non-zero element imbalance across PIM cores. Therefore, we recommend that programmers design compressed data structures that can provide effective data partitioning schemes with high computation balance across thousands of PIM cores.

**Recommendation #3.** *Design adaptive algorithms that (i) trade off computation balance across PIM cores for lower data transfer costs to PIM-enabled memory, and (ii) adapt their configuration to the particular patterns of each input given, as well as the characteristics of the PIM hardware.* Our analysis in Sections 5.6.1, 5.6.2 and 5.6.2 demonstrates that the best-performing SpMV execution on the UPMEM PIM system can be achieved using algorithms that (i) trade off computation for lower data transfer costs, and (ii) select the load balancing strategy and data partitioning policy based on the particular sparsity pattern of the input matrix. In addition, the performance of each balancing scheme and data partitioning technique for SpMV execution highly depends on the characteristics of the underlying PIM hardware, as we explain in Section 5.6.2 and Appendix 8.1.3. To this end, we recommend that software designers implement heuristics and selection methods for their algorithms to adapt their configuration to the underlying hardware characteristics of the PIM system and the input data given.

**Recommendation #4.** *Provide low-cost synchronization support and hardware support to enable concurrent memory accesses by multiple threads to the local DRAM bank to increase parallelism in a multi-threaded PIM core.* Section 5.5 shows that (i) lock acquisitions/releases can cause high overheads in the DPU pipeline, and (ii) fine-grained locking approaches to increase parallelism in critical sections do not improve performance over coarse-grained approaches in the UPMEM PIM hardware. This is because the DMA engine of the DPU serializes DRAM memory accesses included in the critical sections. Based on these key takeaways, we recommend that hardware designers provide lightweight synchronization mechanisms for multithreaded PIM cores [5], and enable concurrent access to local DRAM memory arrays to increase execution parallelism. For example, sub-array level parallelism [620, 622] or multiple DRAM banks per PIM core could be supported in the PIM hardware to improve parallelism.

**Recommendation #5.** *Optimize the broadcast collective operation in data transfers from main memory to PIM-enabled memory to minimize overheads of copying the input data into all DRAM banks in the PIM system.* Figures 5.15 and 5.16 show that SpMV execution using the 1D partitioning technique cannot scale up to a large number of PIM cores. This is because it is severely limited by data transfer costs to broadcast the input vector into *each* DRAM bank of PIM-enabled DIMMs via the narrow off-chip memory bus. To this end, we suggest that hardware and system designers provide a fast broadcast collective primitive to DRAM banks of PIM-enabled memory modules [574].

**Recommendation #6.** *Optimize the gather collective operation at DRAM bank granularity for data transfers from PIM-enabled memory to the host CPU to minimize overheads of retrieving the output results.* Figures 5.19, 5.20 and 5.21 demonstrate that SpMV execution using the *equally-wide* and *variable-sized* 2D partitioning schemes is severely limited by data transfers to retrieve results for the output vector from DRAM banks of PIM-enabled DIMMs. This is due to two reasons: (i) 2D-partitioned kernels create a large number of partial results that need to be transferred from PIM-enabled memory to the host CPU via the narrow memory bus in order to assemble the final output vector, and (ii) the UPMEM PIM system has the limitation that the transfer sizes from/to all DRAM banks involved in the same parallel transfer need to be the same, and therefore a large amount of padding with empty bytes is performed in the *equally-wide* and *variable-sized* schemes. To this end, we suggest that hardware and system designers provide an optimized *gather* primitive to efficiently collect results from multiple DRAM banks to host CPU [574], and support parallel fine-grained data transfers from PIM-enabled memory to host CPU *at DRAM bank granularity* to avoid padding with empty bytes.

**Recommendation #7.** *Design high-speed communication channels and optimized libraries for data transfers to/from thousands of DRAM banks of PIM-enabled memory.* Section 5.7 demonstrates that SpMV execution on the memory-centric UPMEM PIM system achieves a much higher fraction of the machine's peak performance (on average 51.7% for the 32-bit float data type), compared to that on processor-centric CPU and GPU systems. However, the end-to-end performance of both 1D- and 2D-partitioned kernels is significantly limited by data transfer overheads on the narrow memory bus. To this end, we recommend that the hardware architecture and the software stack of real PIM systems be enhanced with low-cost and fast data transfers to/from PIM-enabled memory modules, and/or with support for efficient direct communication among PIM cores [180, 190, 191, 489, 557, 621].

## 5.9 Related Work

To our knowledge, this is the first work that (i) extensively characterizes the Sparse Matrix Vector Multiplication (SpMV) kernel in a real PIM system, and (ii) presents an open-source SpMV library for real-world PIM systems. We briefly discuss closely related prior work.

**Processing-In-Memory (PIM).** A large body of prior work examines Processing-Near-Memory (PNM) [5, 21–23, 25, 27–29, 32–37, 76, 158, 208, 211–213, 215, 216, 218, 221, 228, 265–267, 326, 338–340, 368, 369, 385, 389–392, 397, 543–546, 549, 553, 554, 574, 575, 581]. PNM integrates processing units near or inside the memory via a 3D PNM configuration (i.e., processing units are located at the logic

layer of 3D-stacked memories) [21, 28, 29, 32–35, 37, 76, 212, 266, 267, 369, 581], a 2.5D PNM configuration (i.e., processing units are located in the same package as the CPU connected via silicon interposers) [5, 208, 221], a 2D PNM configuration (i.e., processing units are placed inside DDRX DIMMs) [339, 385, 387, 389–392, 397, 400–403, 543, 544], or at the memory controller of CPU systems [215, 546, 553]. These works propose hardware designs for irregular applications like graph processing [29, 32, 33, 35–37, 218], bioinformatics [5, 25, 27, 400, 401], neural networks [21–23, 134, 208, 221, 324, 385], pointer-chasing workloads [5, 76, 216, 369], and databases [28]. However, *none* of these works examines the SpMV kernel in such systems.

Several prior works enable Processing-Using-Memory (PUM) [174, 175, 180, 181, 186, 190–193, 199, 203, 204, 207, 210, 219, 398, 489, 555–572, 576, 577]. PUM exploits the operational principles of memory cells to perform computation within the memory chip. Prior works propose PUM designs using SRAM [174, 175, 555, 556], DRAM [180, 181, 186, 190–192, 219, 398, 489, 557–560, 572, 576], PCM [193] or RRAM/memristive memory technologies [199, 203, 204, 207, 210, 561–571, 577]. A few PUM works [174, 199, 210, 398, 555, 559, 560] enable the multiplication operation inside memory cells with the goal of performing efficient matrix vector multiplication at low cost within the memory chip. These works design hardware-based solutions to accelerate the *dense* matrix vector multiplication (GEMV) kernel via PUM. However, there is *no* prior work that leverages PUM to accelerate the *Sparse* Matrix Vector Multiplication (SpMV) kernel using state-of-the-art compressed matrix storage formats.

**Sparse Matrix Kernels in PIM Systems.** Xie et al. [383] design heterogeneous PIM units to accelerate SpMV via a 3D PNM configuration, i.e., in HMC-based PIM systems. Sun et al. [574] leverage the buffer device space of DIMM modules to add one processing unit per each DIMM module, and design low-cost inter-DIMM broadcast collectives to minimize data transfer overheads on irregular workloads, like SpMV and graph processing, executed in 2D PNM configurations. Zhu et al. [228] propose a PIM accelerator for Sparse Matrix Matrix Multiplication via a 3D PNM configuration. Fujiki et al. [630] enhance the memory controllers of GPUs with PIM cores to transform the matrix from the CSR to the DCSR format [606] on the fly to minimize memory traffic on SpMV execution. These works propose hardware designs for sparse matrix kernels. In contrast, our work studies software optimizations and strategies to efficiently map compressed matrix storage formats on real near-bank PIM systems, and accelerate SpMV execution on such systems.

**SpMV in Commodity Systems.** Numerous prior works propose optimized SpMV algorithms for CPUs [18, 44, 102–124], GPUs [125–140], heterogeneous CPU-GPU systems [631–638], and distributed CPU systems [141–152]. Optimized SpMV kernels for processor-centric CPU and GPU systems exploit the shared memory model of these systems and data locality in deep cache hierarchies. However, these kernels cannot be directly mapped to most near-bank PIM systems, which have a distributed memory model and a shallow cache hierarchy. Most well-tuned SpMV kernels for distributed CPU and CPU-GPU systems improve performance by overlapping computation with communication among processing units, and exploiting data locality in large cache memories. In contrast, real near-bank PIM architectures are fundamentally different from CPU-GPU systems, since they are *highly distributed*, i.e., there is no direct communication among PIM cores, and include a shallow memory hierarchy. Therefore, SpMV kernels designed for common processor-centric systems cannot be

directly used in near-bank PIM systems.

**Hardware Accelerators for SpMV.** Recent works design accelerators for SpMV [281–288] or other sparse kernels [20, 269–280]. In contrast, our work proposes software optimizations and provides the first characterization study of SpMV on a real PIM system.

**Compressed Matrix Storage Formats.** Prior works propose a range of compressed matrix storage formats [17, 116, 117, 120, 132, 384, 582–585, 600–612] and selection methods to find the most efficient compressed format [98, 130, 613, 617, 639–646]. In this work, we extensively explore the four most widely used *general* compressed matrix formats, and observe that the compressed format (i) needs to provide good balance between computation and memory accesses inside the core pipeline, and (ii) affects load balancing across PIM cores, with corresponding performance implications. Therefore, some compressed formats designed for commodity processor-centric systems might not be suitable or efficient for real PIM systems. We leave the exploration of other PIM-suitable compressed matrix storage formats for future work.

## 5.10 Summary

We present *SparseP*, the first open-source SpMV library for real Processing-In-Memory (PIM) systems, and conduct the first comprehensive characterization analysis of the widely used SpMV kernel on a real-world PIM architecture.

First, we design efficient SpMV kernels for real PIM systems. Our proposed *SparseP* software package supports (1) a wide range of data types, (2) two types of well-crafted data partitioning techniques of the sparse matrix to DRAM banks of PIM-enabled memory, (3) the most popular compressed matrix formats, (4) a wide variety of load balancing schemes across PIM cores, (5) several load balancing schemes across threads of a multithreaded PIM core, and (6) three synchronization approaches among threads within PIM core.

Second, we conduct an extensive characterization study of *SparseP* kernels on the state-of-the-art UPMEM PIM system. We analyze SpMV execution on one single multithreaded PIM core and thousands of PIM cores using 26 sparse matrices with diverse sparsity patterns. We also compare the performance and energy consumption of SpMV on the UPMEM PIM system with those of state-of-the-art CPU and GPU systems to quantify the potential of a real memory-centric PIM architecture on the widely used SpMV kernel over conventional processor-centric architectures. Our analysis of *SparseP* kernels provides programming recommendations for software designers, as well as suggestions and hints for hardware and system designers of future PIM systems.

We believe and hope that our work will provide valuable insights to programmers in the development of efficient sparse linear algebra kernels and other irregular kernels from different application domains tailored for real PIM systems, as well as to architects and system designers in the development of future memory-centric computing systems.





# CHAPTER 6

---

## Conclusions and Future Directions

---

The goal of this dissertation is to significantly improve performance and efficiency of important irregular applications in modern processor-centric CPU and memory-centric NDP/PIM systems. To this end, we develop low-overhead synchronization and well-crafted data access approaches for emerging irregular applications including graph processing kernels, pointer-chasing, data analytics, and sparse linear algebra.

First, we comprehensively analyze prior state-of-the-art algorithms for the widely used graph coloring kernel, and we find that they are still inefficient, since they access application data from the last levels of the memory hierarchy (e.g., main memory) of commodity CPU architectures. Therefore, we introduce the *ColorTM* parallel algorithm, which provides highly efficient execution of the graph coloring kernel. *ColorTM* (i) accesses application data by leveraging the low-cost on-chip cache mem-

ories of CPU systems to minimize data access costs, and (ii) executes short and small critical sections by performing many computations and data accesses *outside* the critical section to minimize synchronization overheads and increase the levels of parallelism among parallel threads. We also extend our proposed design to introduce a highly efficient *balanced* graph coloring algorithm (*BalColorTM*) that can provide high load balance and high resource utilization in the real-world end-applications of graph coloring. Our evaluations show that *ColorTM* and *BalColorTM* can provide significant performance improvements over prior state-of-the-art parallel graph coloring algorithms. We hope that *ColorTM* and *BalColorTM* will encourage further studies on the graph coloring kernel in modern multicore computing systems.

Second, we extensively characterize prior state-of-the-art NUMA-oblivious and NUMA-aware concurrent priority queues in a NUMA CPU architecture using a wide variety of contention scenarios, and find that none of them performs best across all various contention scenarios. Based on this observation, we introduce *SmartPQ*, an adaptive concurrent priority queue for NUMA architectures that achieves the highest performance in all different contention scenarios. We design *SmartPQ* that integrates (i) *Nuddle*, a generic framework that wraps *any* arbitrary NUMA-oblivious concurrent data structure and transforms it to its NUMA-aware counterpart, and (ii) a simple decision tree classifier which predicts the best-performing algorithmic mode between a NUMA-oblivious and a NUMA-aware algorithmic mode. Therefore, *SmartPQ* can dynamically switch during runtime between the NUMA-aware *Nuddle* and its underlying NUMA-oblivious implementation with negligible transition overheads. We demonstrate that *SmartPQ* outperforms prior state-of-the-art NUMA-oblivious and NUMA-aware concurrent priority queues under various contention scenarios, and when the contention of the workload varies over time. We hope that our study will inspire future work on designing *adaptive* algorithmic designs and/or adaptive runtime frameworks for concurrent data structures for modern computing systems.

Third, we rigorously examine the applicability of synchronization mechanisms tailored for processor-centric systems, including CPU, GPU and Massively Parallel Processing systems, to memory-centric NDP architectures, and find that such synchronization approaches are *not* efficient or suitable for NDP systems. To this end, we introduce *SynCron*, the first end-to-end hardware synchronization mechanism for NDP architectures. *SynCron* achieves the goals of high performance, low cost, high programming ease and generality to cover a wide range of synchronization primitives by (1) adding low-cost hardware support near memory for synchronization acceleration, (2) including a specialized cache memory structure to store synchronization information and minimize latency overheads, (3) implementing a hierarchical message-passing communication protocol to minimize expensive network traffic, and (4) integrating a programmer-transparent hardware-only overflow management scheme to minimize performance degradation when hardware resources for synchronization tracking are exceeded. Our evaluations show that *SynCron* can significantly improve system performance and system energy in NDP systems across a wide variety of emerging irregular applications and under various contention scenarios. We hope that *SynCron* will encourage further studies of the synchronization problem in NDP systems and other unconventional computing systems.

Finally, we examine and efficiently map the fundamental memory-bound SpMV kernel on near-

bank PIM systems. Specifically, we design *SparseP*, the first open-source SpMV library for real PIM systems that includes 25 efficient SpMV kernels to cover a wide variety of sparse matrices and real-world applications of SpMV. *SparseP* supports various (1) data types, (2) compressed matrix storage formats, (3) data partitioning techniques of the sparse matrix to PIM-enabled memory modules, (4) load balancing schemes across PIM cores of the system, (5) load balancing schemes across parallel threads of a multithreaded PIM core, and (6) synchronization approaches among parallel threads within PIM core. We comprehensively evaluate the *SparseP* kernels on a real PIM system with 2528 PIM cores using 26 sparse matrices with diverse sparsity patterns. Our extensive evaluations provide new recommendations for software, system and hardware designers of real PIM systems. We also demonstrate that the SpMV execution on a memory-centric PIM system achieves a much higher fraction of the machine's peak performance compared to that on processor-centric CPU and GPU systems, while also having high energy efficiency. We hope that our *SparseP* analysis on a real PIM system will provide valuable insights to software engineers in the development of efficient irregular kernels for real PIM systems, as well as to system designers and hardware architects in the development of future memory-centric computing platforms.

## 6.1 Future Research Directions

The concepts and methods proposed in this dissertation can potentially enable and open up several new research directions. This section describes some promising directions for future work.

### 6.1.1 Accelerating Irregular Applications in Unconventional Systems

Traditional data centers comprise monolithic servers that use DRAM as the main memory of the system, and tightly integrate it with the compute units, e.g., processors or accelerators. However, the increasing demand and growing size of data in modern applications in combination with the device scaling problems of DRAM memory technology [647] have enabled the commercialization of new unconventional systems that consist of heterogeneous memory technologies (e.g., combine DRAM with alternative memory technologies such as 3D-DRAM [467, 468], Phase Change Memory [648], STT-RAM [649], NAND flash-based SSD [650]) or physically separate compute and memory devices as independent network-attached hardware components (e.g., disaggregated memory systems [651–654]). These unconventional computing systems can satisfy the increasing memory capacity demands of emerging applications by providing a large pool of main memory either as a second-tier main memory tightly integrated within the server [465, 655–659] or as remote disaggregated memory components accessed over a high-bandwidth network [651, 654]. Therefore, future work can take inspiration from the techniques proposed in this dissertation to accelerate irregular applications in other unconventional computing systems.

#### In Heterogeneous Memory Systems

Hybrid or heterogeneous memory systems typically include two (or even three) tiers of memory, e.g., integrating a die-stacked DRAM [467, 468] organized as a cache of a larger main memory.

Therefore, the key challenge to fully leverage the heterogeneity of such systems is to accurately identify the performance-criticality of application data and place the corresponding memory pages in the “best-fit” tier of main memory.

At the same time, memory pages corresponding to application data of irregular applications exhibit high variability in their memory access patterns. For example, in SpMV, the memory pages that store the compressed sparse matrix exhibit high spatial locality [286], since the values and the positions of non-zero elements of the compressed matrix are accessed and traversed with a streaming manner in the SpMV execution. Instead, the memory pages that store the input vector typically exhibit low spatial locality [10, 286], since SpMV causes irregular/random memory accesses to the elements of the input vector. However, the accesses on the input vector are *input driven*, i.e., they follow the sparsity pattern of the particular input matrix given: e.g., in sparse matrices with power-law distribution, a small subset of the rows of the matrix has a very large number of non-zero elements (accounting for the majority of the matrices’ non-zero elements) [10, 295], and thus processing these few rows can lead to high spatial locality in the memory pages that store the input vector. Therefore, irregular applications have dynamic access patterns, e.g., memory pages might exhibit either low or high spatial locality during runtime, a fact that also depends on the particular characteristics of the input data given.

Future work could investigate intelligent hot memory page placement approaches and selection methods tailored for irregular applications executed in heterogeneous memory systems. Even though past works [465, 655, 656, 660] propose many different memory page placement techniques, these works do not handle variability in memory access patterns of irregular applications, and do not consider the *dynamic* access patterns exhibited at memory pages for each particular input data given. Therefore, the first steps would involve to investigate the memory access patterns and page hotness/coldness across a wide variety of irregular applications (e.g., graph analytics, pointer-chasing, sparse matrix kernels) executed in modern heterogeneous memory systems, and understand the variability on the memory access patterns exhibited across memory pages. The long-term research goal is to design (i) intelligent data placement approaches for irregular applications that take into consideration the characteristics of the particular input data given, (ii) easy-to-use programming interfaces that communicate information for the characteristics of the application data to the underlying system and hardware in order to leverage data properties, and (iii) cost-effective frameworks and runtime systems that are general to support various types of memory/storage devices and more than two tiers of main memory.

### **In Disaggregated Memory Systems**

Disaggregated memory systems propose to physically separate compute (e.g., processors, accelerators), memory (e.g., DRAM) and storage (e.g., disk) devices as independent and failure-isolated components connected over a high-bandwidth network [651–654]. This way they can provide a cost-effective solution to improve resource utilization, resource scaling and failure handling in data centers, thus decreasing data center costs. In disaggregated systems, almost all the memory in the data center is separated as network-attached disaggregated memory components, and the majority of the application working sets are accessed from the remote disaggregated memory components over the

network. Moreover, disaggregated memory systems are not monolithic: each component in the system implements its own resource allocation and management policy in a completely transparent way from the remaining components in the system.

Achieving high system performance for irregular applications in disaggregated memory systems is challenging for three reasons. First, accesses across the network can be significantly slower than these within the server, and data is typically migrated at a page granularity (e.g., 4KB) [651,652,661–667], thus incurring high data movement overheads. Second, there is high variability in data access latencies as they depend on the location of the remote disaggregated memory components and the contention with other compute components that share the same remote memory components and network. Third, prior runtime systems and hot page selection/placement schemes for heterogeneous systems [465,655,656,660] are not suitable for fully disaggregated memory systems: prior approaches for heterogeneous systems assume that the management of memory pages is handled by the compute component itself and the OS running on it. Instead, this is not the case with fully disaggregated systems, in which remote disaggregated memory components have their own kernel modules and hardware controllers to manage their resources and memory pages (transparently to compute components) [651–653]. Therefore, to efficiently execute irregular applications in such systems new software and hardware solutions are necessary.

Future work would investigate the following new challenges in the execution of irregular applications in fully disaggregated memory systems: (i) the high data movement overheads imposed by remotely accessing data over the network, (ii) the high variability in data access costs during runtime due to network and memory sharing, (iii) the unconventional distributed approach of managing the data on multiple components in the system with a completely transparent way to each other, and (iv) the high memory sharing and the memory protection issues for pages located in remote disaggregated memory components, which can be accessed by multiple processes that concurrently run at different compute components of the system.

The first step is to develop a cost model, a software-based simulator, or a hardware-based emulator for fully disaggregated memory systems, which can support various configurations for the network characteristics (e.g., network topology, network bandwidth/latency), and evaluate, analyze and understand critical performance overheads in the execution of a wide variety of irregular applications with diverse access patterns. Rigorously and comprehensively understanding performance implications of irregular applications in fully disaggregated memory systems can provide valuable insights to software engineers, system designers and hardware architects of this architecture. The next steps are to propose new address translation approaches and kernel modules to minimize system-level overheads (e.g., page faults), flexible (asynchronous and synchronous) programming interfaces and abstractions to easily access remote data over the network, fast network technologies to mitigate network-related bottlenecks, low-overhead synchronization and memory sharing/coherence mechanisms for multiple memory components in the system to ensure correctness at low cost, as well as to leverage the NDP paradigm [158] in disaggregated memory components to reduce access costs to remote data. The long-term goal is to perform research on designing fundamentally new approaches for all key components of the computing stack, which need to be distributed, *disaggregated* and scale

elastically, in keeping with the promise of resource disaggregation.

### 6.1.2 Adaptive Algorithmic, System-Level and Hardware-Based Approaches for Irregular Applications

Emerging irregular applications exhibit dynamic workload demands and contention, i.e., their memory access patterns, bandwidth, latency and parallelization demands vary over time. For instance, irregular key-value stores such as binary search trees [334, 444, 447–449], linked lists [13, 70, 445, 446], priority queues [4, 13, 15, 77], hash tables [459, 668, 669] are used in database management systems, and multiple users perform lookup and update operations (e.g., insert or delete) on them with various frequencies over runtime: concurrent key-value store data structures exhibit high variability during time in the levels of contention and their memory access patterns as they depend on the amount and types of operations (lookup, insert, delete) that users perform on them during runtime. Similarly, modern computing systems and large-scale architectures exhibit high variability into network characteristics (e.g., memory bandwidth, latency, network topology), runtime contention (e.g., co-running applications), and available hardware resources (e.g., memory devices, accelerators). For example, in disaggregated memory systems, the architectures, component placements and network characteristics can highly vary over time, since multiple hardware components can be dynamically added, removed or upgraded, and network technologies or topologies can also flexibly change over time [651–653]. Similarly, virtualized environments support dynamic sets of resources, in which virtual machines can be dynamically added, removed or change their hardware characteristics/configuration over time [670]. The dynamic variability on the (i) runtime workload demands of irregular applications, and (ii) architecture and network characteristics of modern computing systems results in significant variations on data access latencies and data movement overheads in the execution of emerging irregular applications, which might thus significantly degrade system performance and resource utilization. To this end, future work would involve (i) designing *adaptive* algorithmic approaches for irregular applications depending on the runtime contention, and application, hardware and network characteristics, and (ii) enabling the system and hardware to *dynamically* change their configurations depending on the availability of resources and runtime application behavior.

#### Adaptive Algorithmic Designs

The research goal is to design adaptive algorithms that *on-the-fly* change their parallelization strategy, synchronization approach and data management policy over time to significantly improve system performance, energy efficiency and data access costs in irregular applications. The key idea is to enable parallel threads or background/monitor threads to track properties of application data and/or runtime statistics, and employ low-overhead decision-making mechanisms to select between multiple configurations (e.g., different parallelization strategies, synchronization approaches, data management policies) during the execution. For example, in Chapter 3, we propose an adaptive priority queue for NUMA CPU architectures that dynamically changes its data access policy and synchronization scheme by tracking the levels of contention during runtime and integrating a lightweight decision tree classifier that predicts the optimal parallelization strategy based on runtime statistics.

Other examples include to change on-the-fly the graph traversal strategy on graph processing kernels depending on the number of the edges of the current vertex that is being processed (e.g., in real-world graphs with power-law distribution [295] a few vertices have a significantly larger number of edges compared the remaining vertices) or alternate the data access policy in sparse matrix kernels when processing rows with a small/large number of non-zero elements. The challenge in designing adaptive algorithms for irregular applications is to minimize the performance overheads between transitions on different configurations.

### **Adaptive Runtime Systems**

The research goal is to develop adaptive runtime systems that dynamically adjust the task assignments, task scheduling and data distribution policies during runtime to significantly improve system performance, resource utilization and financial costs. The key idea is to integrate in the runtime systems dedicated managers that monitor tasks and jobs running across the computing nodes of the system, and decide on the optimal configuration, e.g., optimal task scheduling across the computing nodes of the system, when architecture, hardware and/or network characteristics change. For instance, a recent work [670] proposes a novel runtime system for distributed machine learning training, that *dynamically* tunes the number of pipeline stages, depending on the network load/contention at any given time and the number of available computing nodes (i.e., GPUs) in the system. Therefore, future work could investigate designing adaptive runtime systems for distributed training of sparse neural networks, that include dedicated monitoring managers which track the execution of running tasks and on-the-fly tune the parallelization approach and data distribution policy across multiple computing nodes (e.g., GPUs, TPUs, NPUs) of large-scale clusters and in cloud environments (e.g., when using virtual machines), when new computing nodes are added or removed in the system and when network load/contention changes. Similarly to adaptive algorithmic designs, the key challenge in such intelligent runtime systems is to achieve low synchronization overheads between transitions from one configuration to another.

### **Adaptive Hardware Mechanisms**

The key research goal is to propose adaptive hardware mechanisms that on-the-fly adjust their performance optimization strategies depending on availability of resources and runtime application characteristics. The key idea is to integrate hardware controllers in the computing system that decide between different optimization policies by leveraging system-level metadata (e.g., page tables/TLBs), simple prediction heuristics, or statistics collected during runtime at low cost. For instance, a few recent works [671–675] propose hardware compression mechanisms for cache and main memory of CPU systems that dynamically enable/disable compression [671–673] or on-the-fly select the best-performing compression algorithm [674, 675] based on properties of application data or the runtime application behavior. Other examples include to design (i) intelligent hardware prefetchers that on-the-fly enable/disable fetching application data from main memory to cache memory depending on the current bandwidth utilization and locality of application data, or (ii) effective selection granularity mechanisms that on-the-fly decide the granularity at which data migrations should be served (e.g., in heterogeneous systems choosing if a data migration from the second tier main memory to the cache-based main memory should be served by a page or a smaller granularity, e.g., cache line granu-

larity) depending on the runtime network and application characteristics, and the available memory resources. The key challenge in designing adaptive hardware mechanisms is to implement intelligent prediction heuristics and/or to enable keeping metadata for the runtime system and application behavior at low hardware- and system-level cost.

## 6.2 Concluding Remarks

In this dissertation we extensively characterize the execution of irregular applications in modern processor-centric (e.g., CPUs) and memory-centric (e.g., NDP/PIM) systems, and provide directions to bridge the gap between processor-centric systems and memory-centric systems in the context of important yet difficult irregular applications. We observe that excessive synchronization and high memory intensity of irregular applications can significantly degrade system performance. Therefore, we propose low-overhead synchronization and well-crafted data access techniques for irregular applications, and demonstrate that they can significantly increase parallelism, improve energy efficiency, minimize data access costs, and accelerate performance of emerging irregular applications in CPU and NDP/PIM systems. Specifically, we introduce four new designs that enable efficient execution of irregular applications in modern computing systems: (1) *ColorTM*, a speculative synchronization scheme co-designed with an effective data access policy that accelerates the graph coloring kernel in modern CPU systems, (2) *SmartPQ*, an adaptive algorithm design that improves performance of priority queue data structure in NUMA CPU architectures, (3) *SynCron*, a practical and low-overhead hardware synchronization mechanism that effectively leverages the benefits of NDP for a wide range of irregular applications, and (4) *SparseP*, a wide collection of parallel algorithms to easily attain high performance of the SpMV kernel on real PIM systems. We hope that the ideas, analysis, methods and techniques presented in this dissertation will enable new studies and research directions to accelerate the execution of important data-intensive irregular applications in current and future computing platforms.



# CHAPTER 7

---

## Other Works of the Author

---

In addition to the works presented in this dissertation, the author of this dissertation has also contributed to several other research works done in collaboration with SAFARI Research Group members at ETH Zürich. This chapter briefly overviews these works.

**PrIM [156, 161, 162, 404]:** In modern computing systems like CPU and GPU systems, a large fraction of the execution time and energy consumption of modern data-intensive irregular workloads is spent on moving data between memory and processor cores. Recent research explores different PIM configurations [5, 22, 25, 27–29, 33–37, 129, 156–268], since the PIM paradigm provides a promising way to alleviate the data movement bottleneck between memory and processors. The UPMEM company [157, 338, 399] has designed and fabricated the first commercially-available near-bank PIM architecture. In this work, we conduct an experimental characterization of the UPMEM-based PIM system using microbenchmarks to assess various architecture limits such as compute throughput and memory bandwidth, and we present PrIM (Processing-In-Memory benchmarks), a benchmark suite of 16 irregular workloads from different application domains (e.g., dense/sparse linear algebra, databases, data analytics, graph processing, neural networks, bioinformatics, image processing),

which we identify as memory-bound. We evaluate the performance and scaling characteristics of PrIM benchmarks on the UPMEM PIM architecture, and compare their performance and energy consumption to their state-of-the-art CPU and GPU counterparts. Our extensive evaluation conducted on two real UPMEM-based PIM systems provides new insights about suitability of different irregular workloads to the PIM system, programming recommendations for software designers, and suggestions and hints for hardware and architecture designers of future PIM systems.

**NATSA [208, 676]:** Time series analysis is an irregular computation kernel that processes a chronologically ordered set of samples of a real-valued variable that can contain millions of observations, and is used to analyze information in a wide variety of domains including epidemiology, genomics, neuroscience, medicine and environmental sciences. Matrix profile is the state-of-the-art algorithm to perform time series analysis, by computing the most similar subsequence for a given query subsequence within a sliced time series. In this work, we evaluate the state-of-the-art CPU implementation of the matrix profile algorithm on a real multi-core machine, i.e., Intel Xeon Phi KNL, and observe that its performance is heavily bottlenecked by data movement between the off-chip memory units and the on-chip computation units that execute matrix profile. To reduce the data movement overheads, we design a near-data processing accelerator for time series analysis, called NATSA. NATSA exploits the low-latency, high-bandwidth, and energy-efficient memory access provided by modern 3D-stacked High Bandwidth Memory (HBM), and integrates specialized custom processing units in the logic layer of HBM. This way NATSA enables energy-efficient and fast matrix profile computation near memory, i.e., where time series data resides, and reduces the data movement costs between the computation units and the memory units. NATSA provides generality and flexibility supporting a wide range of time series applications, and significantly improves system performance and energy efficiency over state-of-the-art CPU, GPU and NDP systems.

**SMASH [286, 677]:** The matrices involved in irregular sparse linear algebra computation kernels are very large in size and highly sparse, i.e., the vast majority of the elements are zeros. Prior research works [17, 116, 117, 120, 132, 384, 582–585, 600–612] design compressed storage formats for sparse matrices: the non-zero elements and their positions within the matrix are stored using additional data structures and different encodings. However, determining the positions of the non-zero elements in the compressed encoding (i.e., *indexing*) requires a series of pointer-chasing operations in memory, that are highly inefficient in modern processors and memory hierarchies, and incur high data access costs. The key idea of SMASH is to explicitly enable the hardware to recognize and exploit the compression encoding used in software for any sparse matrix. On the software side, SMASH efficiently compresses any sparse matrix via a novel software encoding that is based on a hierarchy of bitmaps. On the hardware side, SMASH includes a lightweight hardware unit, named Bitmap Management Unit, that is used to perform highly-efficient scans of the hierarchy of bitmaps, and thus enabling highly efficient indexing in sparse matrices and minimizing data access costs in sparse linear algebra computation kernels. SMASH provides significant speedups in sparse matrix computations by eliminating the expensive pointer-chasing operations required in state-of-the-art compressed matrix storage formats.

# CHAPTER 8

---

## Appendix A

---

### 8.1 Extended Results for *SparseP*

#### 8.1.1 Synchronization Approaches in Block-Based Compressed Matrix Formats

We compare the coarse-grained locking (*lb-cg*) and the fine-grained locking (*lb-fg*) approaches in the BCOO format. Figure 8.1 shows the performance achieved by the BCOO format for all the data types when balancing the blocks or the non-zero elements across 16 tasklets of one DPU. We evaluate all small matrices of Table 5.3, i.e., *delaunay\_n13* (**D**), *wing\_nodal* (**W**), *raefsky4* (**R**) and *pkustk08* (**P**) matrices.

Our key finding is that the fine-grained locking approach performs similarly with the coarse-grained locking approach. The fine-grained locking approach does not increase parallelism in the UPMEM PIM architecture, since memory accesses executed by multiple tasklets to the local DRAM bank are serialized in the DMA engine of the DPU. The same key finding holds independently of the

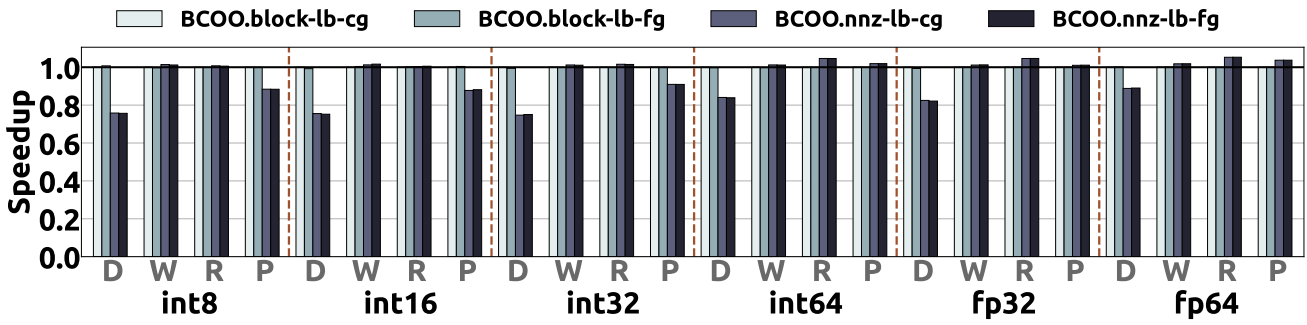


Figure 8.1: Performance of the BCOO format with various load balancing schemes and synchronization approaches for all the data types and small matrices using 16 tasklets of one DPU.

compressed matrix format used.

### 8.1.2 Fine-Grained Data Transfers in 2D Partitioning Techniques

Figures 8.2 and 8.3 compare coarse-grained data transfers (i.e., performing parallel data transfers to all 2048 DPUs at once, padding with empty bytes at the granularity of 2048 DPUs) with fine-grained data transfers (i.e., iterating over the ranks and for each rank performing parallel data transfers to the 64 DPUs of the same rank, padding with empty bytes at the granularity of 64 DPUs) for all matrices of our large matrix dataset in the *equally-wide* and *variable-sized* schemes, respectively. The reported key findings of Figure 5.17 (Section 5.6.2) apply to all matrices with diverse sparsity patterns.

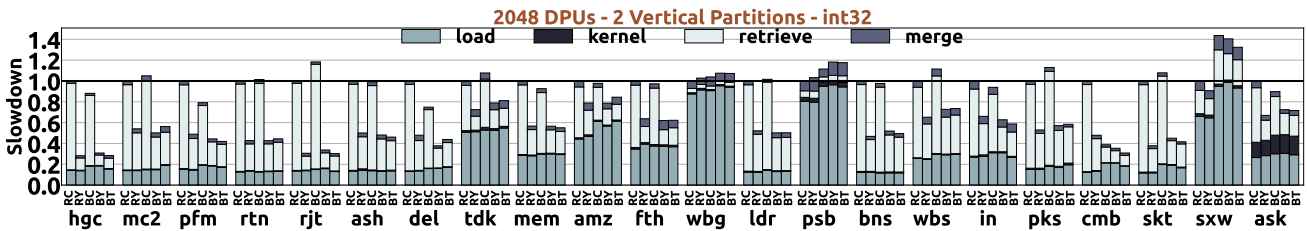


Figure 8.2: Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 2 vertical partitions. Performance is normalized to that of the RC scheme.

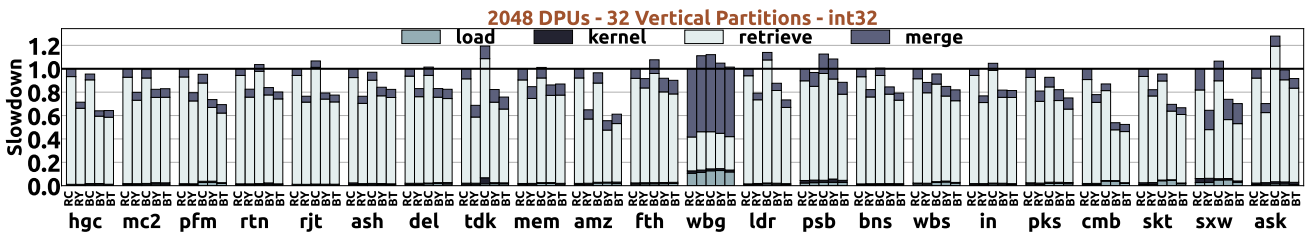


Figure 8.3: Performance comparison of RC: RBDCOO with coarse-grained transfers, RY: RBDCOO with fine-grained transfers in the output vector, BC: BDCOO with coarse-grained transfers, BY: BDCOO with fine-grained transfers only in the output vector, and BT: BDCOO with fine-grained transfers in both the input and the output vector using the int32 data type, 2048 DPUs and having 32 vertical partitions. Performance is normalized to that of the RC scheme.

### 8.1.3 Effect of the Number of Vertical Partitions Using Two Different UP-MEM PIM Systems

We compare SpMV execution in the two different UPMEM PIM systems using 2048 DPUs and 16 tasklets for each DPU. Table 8.1 shows the characteristics of two different UPMEM PIM systems. We calculate the available PIM peak performance and PIM bandwidth assuming 2048 DPUs for both PIM systems<sup>1</sup>. We estimate the PIM peak performance as  $Total\_DPUs * AT$ , where the arithmetic throughput (AT) is calculated for the multiplication operation by running the arithmetic throughput microbenchmark of the PrIM benchmark suite [161, 162] in each of the two UPMEM PIM systems (See Appendix 8.2). We estimate the PIM bandwidth as  $Total\_DPUs * Bandwidth\_DPU$ , where the  $Bandwidth\_DPU$  is calculated according to prior work [161, 162]. Specifically, the theoretical maximum MRAM bandwidth (i.e.,  $Bandwidth\_DPU$ ) is 700 MB/s and 850 MB/s at a DPU frequency of 350 MHz (PIM system A) and 425 MHz (PIM system B), respectively.

System	Avail. DPUs	Frequency	PIM Peak Performance	PIM Bandwidth	Host CPU	CPU Peak Performance	Bus Bandwidth
PIM System A	2048 DPUs	350 MHz	3.78 GFLOPS	1.43 TB/s	Intel Xeon Silver 4110 @2.1 GHz	660 GFLOPS	23.1 GB/s
PIM System B	2048 DPUs	425 MHz	4.63 GFLOPS	1.74 TB/s	Intel Xeon Silver 4215 @2.5 GHz	1016 GFLOPS	21.8 GB/s

Table 8.1: Evaluated UPMEM PIM Systems.

Figures 8.4, 8.5 and 8.6 compare SpMV execution in the two different UPMEM PIM systems (2048 DPUs) using 2D-partitioned kernels with the COO format, when varying the number of vertical partitions from 1 to 32 (in steps of multiple of 2) for the int32 (left) and fp64 (right) data types.

We observe that the number of vertical partitions that provides the best performance on SpMV execution varies depending on the input matrix and the PIM system. For example, in PIM system B with the int32 data type, DCOO performs best for the `hgc` matrix with 16 vertical partitions, while in PIM system A, DCOO performs best for the same matrix with 8 vertical partitions. Similarly, in PIM system A with the fp64 data type, BDCOO performs best for the `rjt` matrix with 4 vertical partitions. Instead, in PIM system B with the fp64 data type, BDCOO's performance does not improve for the `rjt` matrix when having more than 1 vertical partition (i.e., compared to when using the 1D partitioning technique). We conclude that the best-performing parallelization scheme that achieves the best performance in SpMV depends on the characteristics of both the input sparse matrix and the underlying PIM system.

### 8.1.4 Performance of Compressed Matrix Formats Using 2D Partitioning Techniques

Figures 8.7, 8.8, 8.9 compare the performance achieved by various compressed matrix formats for each of the three types of the 2D partitioning technique for all matrices of our large matrix dataset. The reported key findings explained in Section 5.6.2 apply to all matrices with diverse sparsity patterns.

<sup>1</sup>Both UPMEM PIM systems support 20 UPMEM PIM DIMMs with 2560 DPUs in total. However, both UPMEM-based PIM systems include multiple faulty DPUs. Thus, for a fair comparison between two systems we conduct our experiments using 2048 DPUs in both systems.

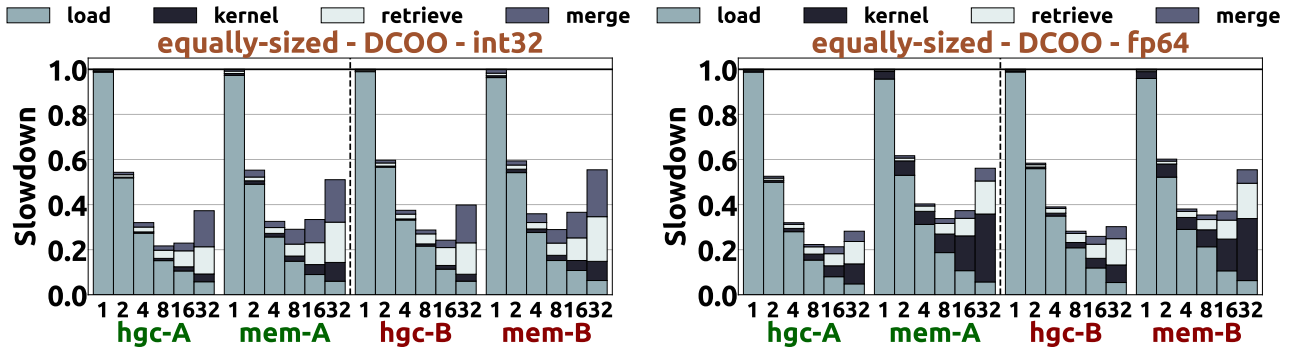


Figure 8.4: Execution time breakdown of DCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems.

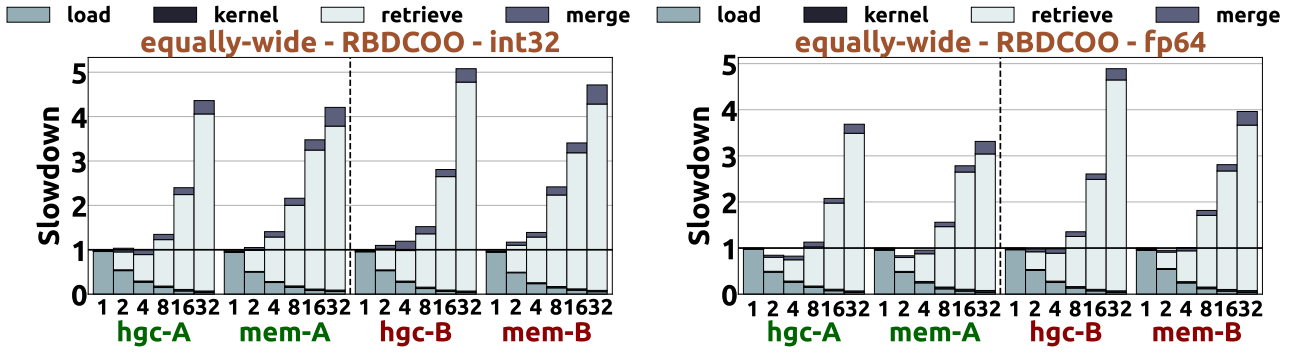


Figure 8.5: Execution time breakdown of RBDCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems.

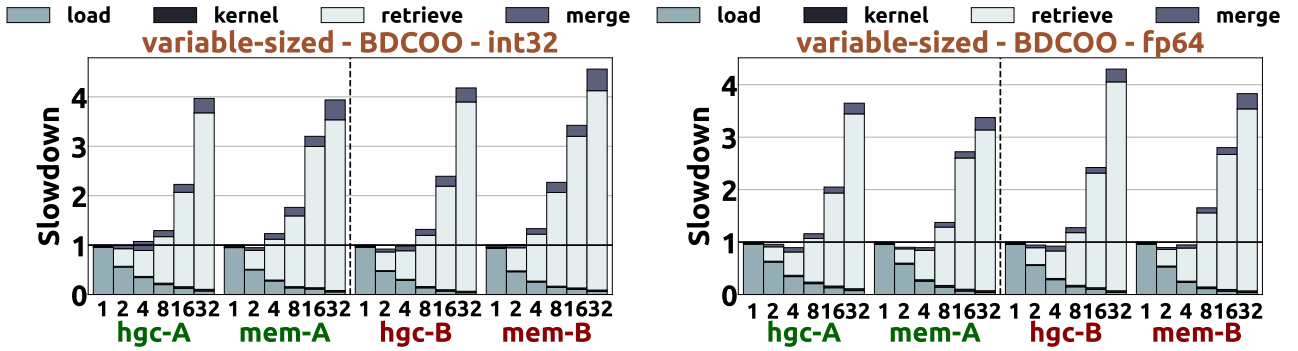


Figure 8.6: Execution time breakdown of BDCOO using 2048 DPUs when varying the number of vertical partitions from 1 to 32 for the int32 (left) and fp64 (right) data types on two different UPMEM PIM systems.

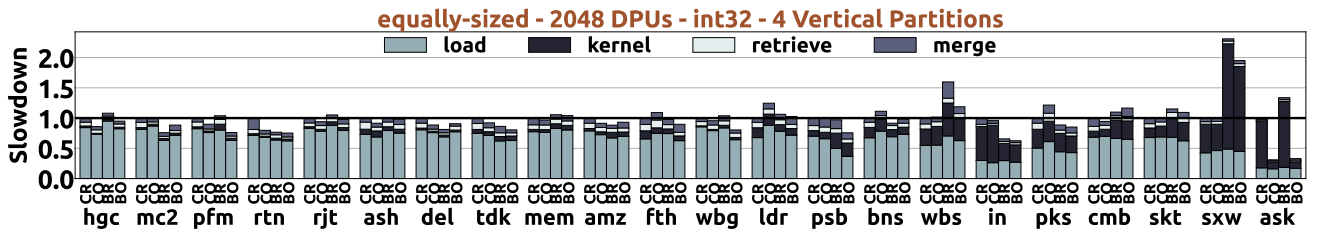


Figure 8.7: End-to-end execution time breakdown of the *equally-sized* 2D partitioning technique for CR: DCSR, CO: DCOO, BR: DBCSR and BO: DBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of DCSR.

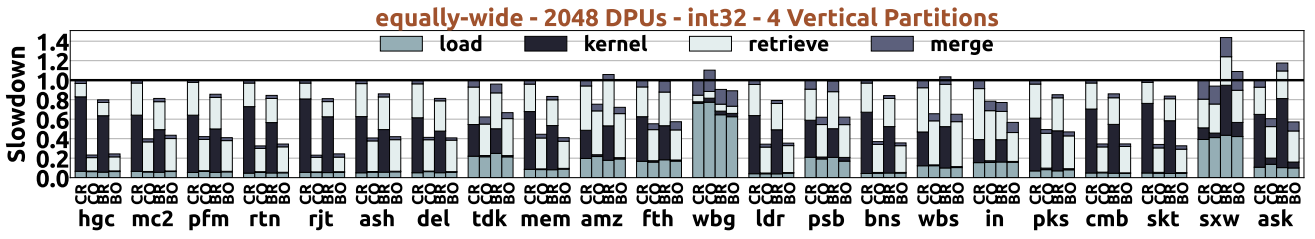


Figure 8.8: End-to-end execution time breakdown of the *equally-wide* 2D partitioning technique for CR: RBDCSR, CO: RBDCOO, BR: RBDBCSR and BO: RBDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of RBDCSR.

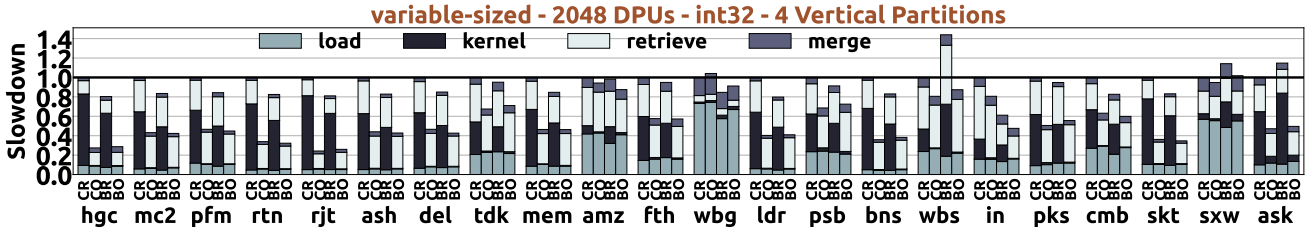


Figure 8.9: End-to-end execution time breakdown of the *variable-sized* 2D partitioning technique for CR: BDCOO, CO: BDCOO, BR: BDBCSR and BO: BDBCOO schemes using 4 vertical partitions and the int32 data type. Performance is normalized to that of BDCSR.

### 8.1.5 Analysis of 1D- and 2D-Partitioned Kernels in Two UPMEM PIM Systems

Figures 8.10 and 8.11 compare the throughput and the performance, respectively, achieved by the best-performing 1D- and 2D-partitioned kernels in two different UPMEM PIM systems (Table 8.1 presents the characteristics of the two UPMEM PIM systems). For 1D partitioning, we use the lock-free COO (COO.nnz-1f) and coarse-grained locking BCOO (BCOO.block) kernels. For each matrix, we vary the number of DPUs from 64 to 2048 DPUs, and select the best-performing end-to-end execution throughput. For 2D partitioning, we use the *equally-sized* COO (DCOO) and BCOO (BCOO) kernels with 2048 DPUs for both systems. For each matrix, we vary the number of vertical partitions from 2 to 32 (in steps of multiple of 2), and select the best-performing end-to-end execution throughput.

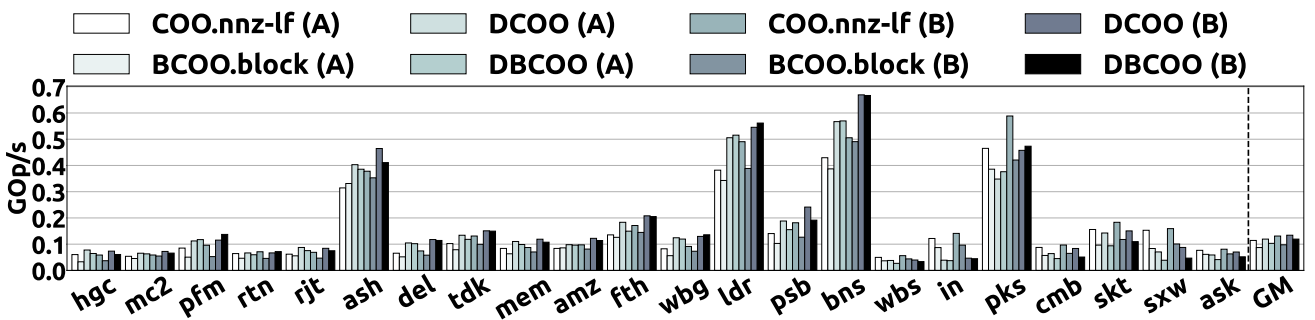


Figure 8.10: Throughput of 1D- and 2D-partitioned kernels for the fp32 data type using two different UPMEM PIM systems.

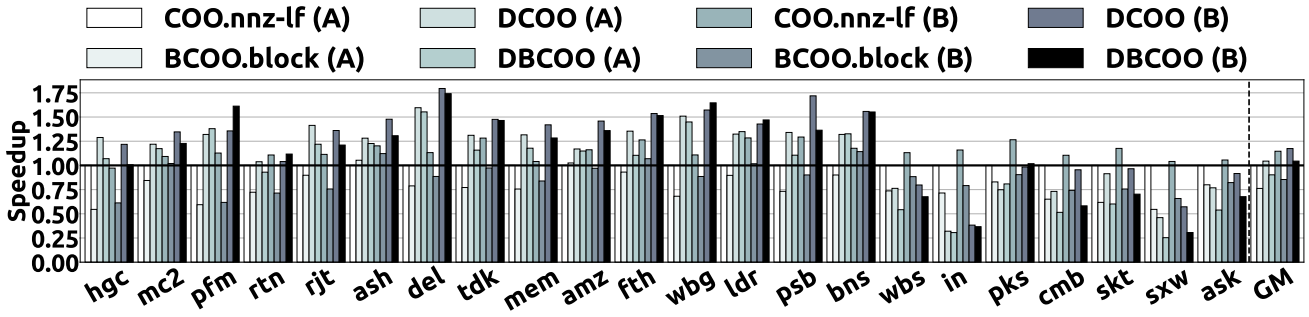


Figure 8.11: Performance comparison of 1D- and 2D-partitioned kernels for the fp32 data type using two different UPMEM PIM systems. Performance is normalized to that of `COO.nnz-lf (A)`.

We draw three findings. First, we observe that in both systems the best performance is achieved using a smaller number of DPUs than 2048 DPUs. This is because SpMV execution in both UPMEM PIM systems is significantly bottlenecked by expensive data transfers performed via the narrow memory bus. As a result, the best-performing 1D- and 2D-partitioned kernels trade off computation with lower data transfer costs, thus causing many DPUs to be *idle*. Second, we find that in both systems the 2D-partitioned kernels outperform the 1D-partitioned kernels in regular matrices (i.e., from `hgc` to `bns` matrices on x axis), while the 1D-partitioned kernels outperform the 2D-partitioned kernels in scale-free matrices, i.e., in matrices that have high non-zero element disparity among rows and columns (i.e., from `wbs` to `ask` matrices on x axis). Third, we observe that PIM system B improves performance over PIM system A by  $1.14\times$  (averaged across all matrices). This is because the DPUs of the PIM system B run at a higher frequency than that of PIM system A (425 MHz vs 350 MHz), providing higher peak performance on the system. Specifically, with 2048 DPUs, peak performance of the PIM system A and PIM system B is 3.78 GFlops and 4.63 GFlops, respectively, i.e., PIM system B provides  $1.22 \times$  higher computation throughput than PIM system A.



## 8.2 Arithmetic Throughput of One DPU for the Multiplication Operation

We evaluate the arithmetic throughput of the DPU for the multiplication (MUL) operation. We use the arithmetic throughput microbenchmark of the PrIM benchmark suite [161, 162] and configure it for the all data types.

Figure 8.12 shows the measured arithmetic throughput (in MOperations per second) for the MUL operation varying the number of tasklets of one DPU at 350 MHz (PIM system A in Table 8.1) for all the data types. The arithmetic throughput for the MUL operation is 12.941 MOps, 10.524 MOps, 8.861 MOps, 2.381 MOps, 1.847 MOps, and 0.517 MOps for the int8, int16, int32, int64, fp32 and fp64 data types, respectively.

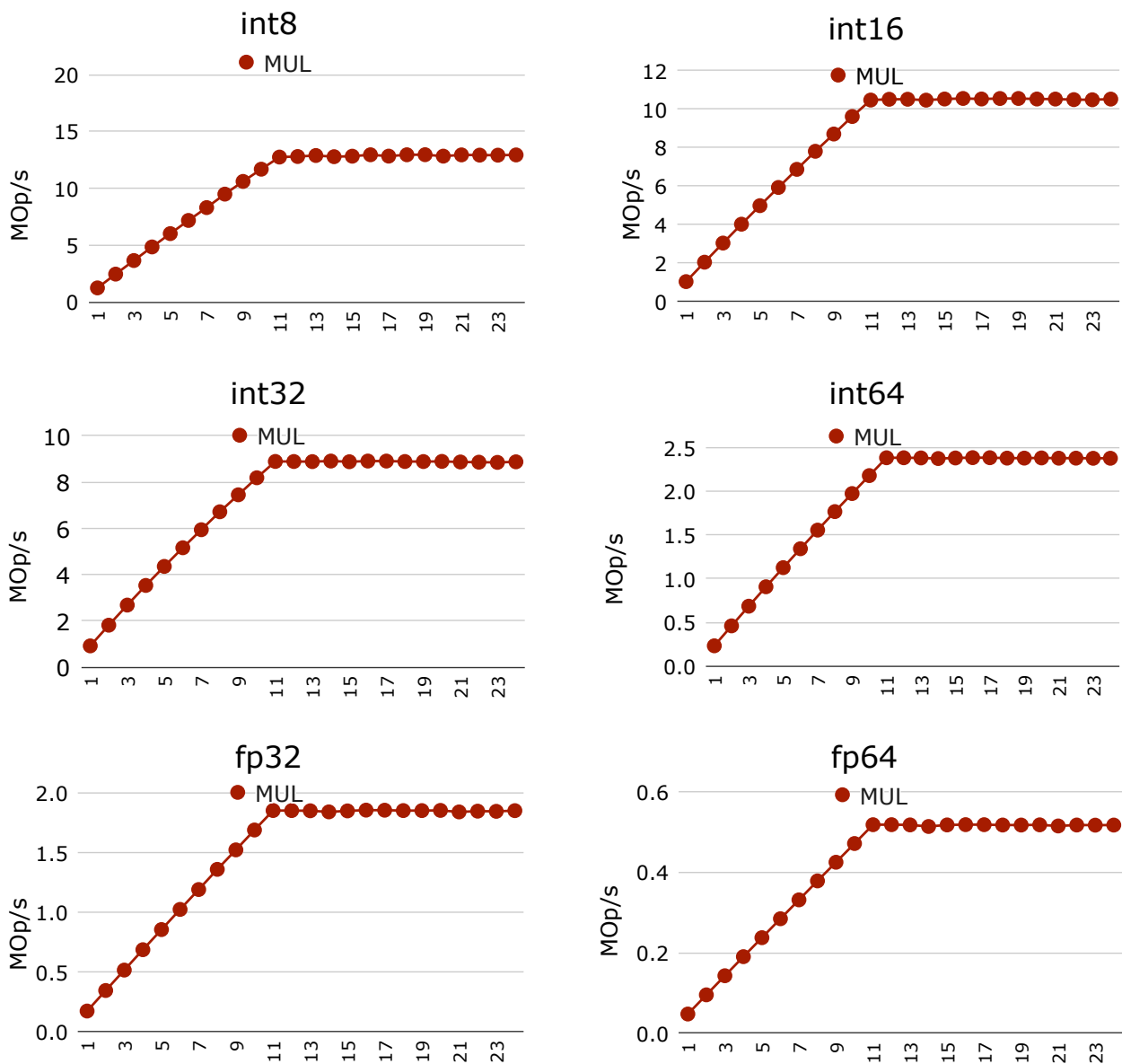


Figure 8.12: Throughput of the MUL operation on one DPU at 350 MHz for all the data types.

Figure 8.13 shows the measured arithmetic throughput (in MOperations per second) for the MUL operation varying the number of tasklets of one DPU at 425 MHz (PIM system B in Table 8.1) for all the data types. The arithmetic throughput for the MUL operation is 15.656 MOps, 12.721 MOps, 10.732 MOps, 2.888 MOps, 2.259 MOps, and 0.631 MOps for the int8, int16, int32, int64, fp32 and fp64 data types, respectively.

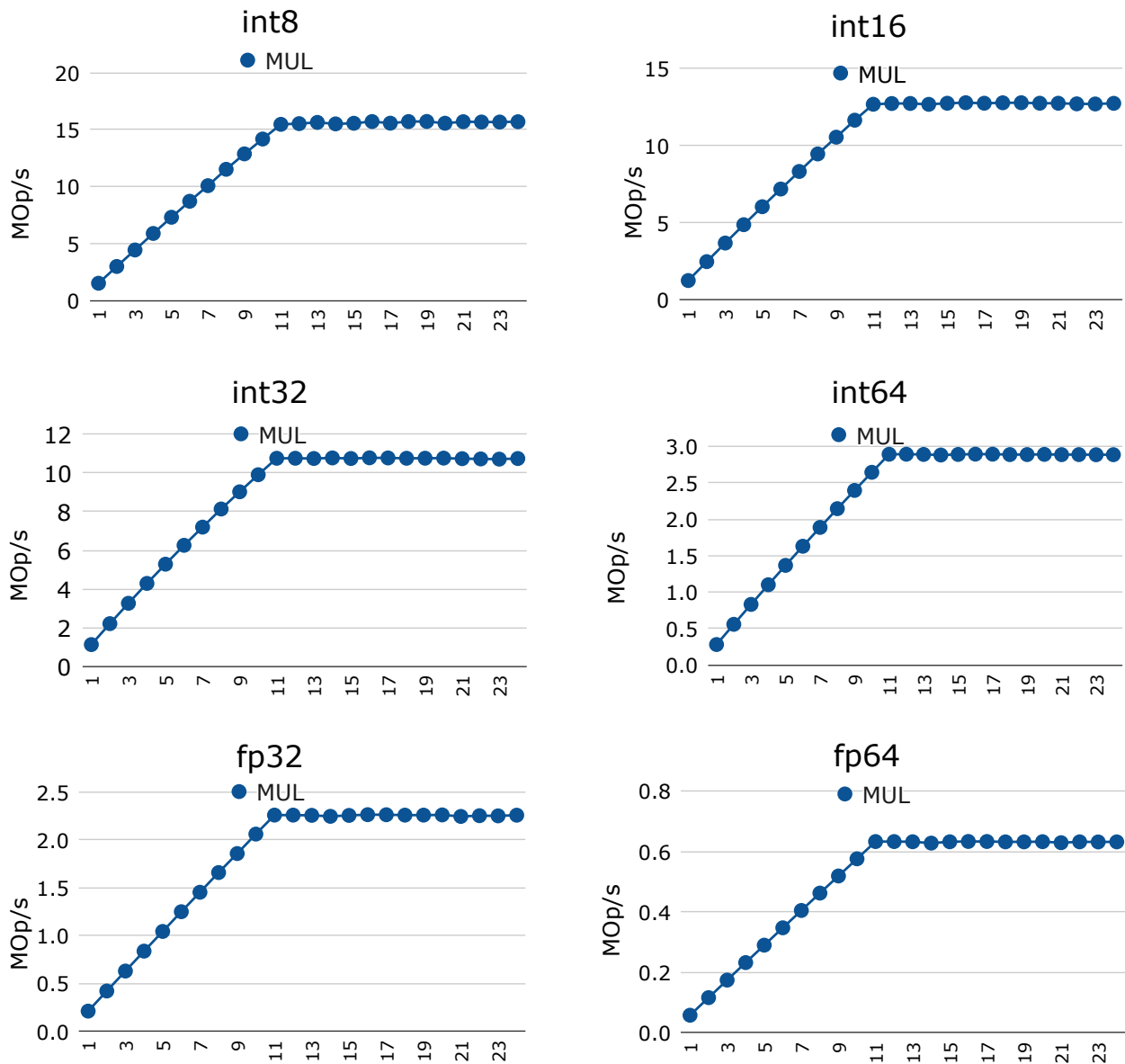


Figure 8.13: Throughput of the MUL operation on one DPU at 425 MHz for all the data types.

### 8.3 The *SparseP* Software Package

Table 8.2 summarizes the SpMV PIM kernels provided by the *SparseP* library. All kernels support a wide range of data types, i.e., 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float, and 64-bit float data types.

Partitioning Technique	Compressed Format	Balancing Across PIM Cores	Balancing Across Threads	Synchronization Approach
1D	CSR	rows nnz <sup>*</sup>	rows, nnz <sup>*</sup> rows, nnz <sup>*</sup>	- -
	COO	rows nnz <sup>*</sup> nnz	rows, nnz <sup>*</sup> rows, nnz <sup>*</sup> nnz	- - lb-cg / lb-fg / lf
	BCSR	blocks <sup>†</sup> nnz <sup>†</sup>	blocks <sup>†</sup> , nnz <sup>†</sup> blocks <sup>†</sup> , nnz <sup>†</sup>	lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup> lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup>
	BCOO	blocks nnz	blocks, nnz blocks, nnz	lb-cg / lb-fg / lf lb-cg / lb-fg / lf
2D <i>equally-sized</i>	CSR	-	rows, nnz <sup>*</sup>	-
	COO	-	nnz	lb-cg / lb-fg / lf
	BCSR	-	blocks <sup>†</sup> , nnz <sup>†</sup>	lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup>
	BCOO	-	blocks, nnz	lb-cg / lb-fg
2D <i>equally-wide</i>	CSR	nnz <sup>*</sup>	rows, nnz <sup>*</sup>	-
	COO	nnz	nnz	lb-cg / lb-fg / lf
	BCSR	blocks <sup>†</sup> nnz <sup>†</sup>	blocks <sup>†</sup> , nnz <sup>†</sup> blocks <sup>†</sup> , nnz <sup>†</sup>	lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup> lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup>
	BCOO	blocks nnz	blocks, nnz blocks, nnz	lb-cg / lb-fg lb-cg / lb-fg
2D <i>variable-sized</i>	CSR	nnz <sup>*</sup>	rows, nnz <sup>*</sup>	-
	COO	nnz	nnz	lb-cg / lb-fg / lf
	BCSR	blocks <sup>†</sup> nnz <sup>†</sup>	blocks <sup>†</sup> , nnz <sup>†</sup> blocks <sup>†</sup> , nnz <sup>†</sup>	lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup> lb-cg <sup>‡</sup> / lb-fg <sup>‡</sup>
	BCOO	blocks nnz	blocks, nnz blocks, nnz	lb-cg / lb-fg lb-cg / lb-fg

Table 8.2: The *SparseP* library. \*: row-granularity, <sup>†</sup>: block-row-granularity, <sup>‡</sup>: (only for 8-bit integer and small block sizes)

## 8.4 Large Matrix Dataset

We present the characteristics of the sparse matrices of our large matrix data set. Table 8.3 presents the sparsity of the matrix (i.e.,  $\text{NNZ} / (\text{rows} \times \text{columns})$ ), the standard deviation of non-zero elements among rows (NNZ-r-std) and columns (NNZ-c-std).

Matrix Name	Rows x Columns	NNZs	Sparsity	NNZ-r-std	NNZ-c-std
hugetric-00020	7122792 x 7122792	21361554	4.21e-07	0.031	0.031
mc2depi	525825 x 525825	2100225	7.59e-06	0.076	0.076
parabolic_fem	525825 x 525825	3674625	1.33e-05	0.153	0.153
roadNet-TX	1393383 x 1393383	3843320	1.98e-06	1.037	1.037
rajat31	4690002 x 4690002	20316253	9.24e-07	1.106	1.106
af_shell1	504855 x 504855	17588875	6.90e-05	1.275	1.275
delaunay_n19	524288 x 524288	3145646	1.14e-05	1.338	1.338
thermomech_dK	204316 x 204316	2846228	6.81e-05	1.431	1.431
memchip	2707524 x 2707524	14810202	2.02e-06	2.062	1.173
amazon0601	403394 x 403394	3387388	2.08e-05	2.79	15.29
FEM_3D_thermal2	147900 x 147900	3489300	1.59e-04	4.481	4.481
web-Google	916428 x 916428	5105039	6.08e-06	6.557	38.366
ldoor	952203 x 952203	46522475	5.13e-05	11.951	11.951
poisson3Db	85623 x 85623	2374949	3.24e-04	14.712	14.712
boneS10	914898 x 914898	55468422	6.63e-05	20.374	20.374
webbase-1M	1000005 x 1000005	3105536	3.106e-06	25.345	36.890
in-2004	1382908 x 1382908	16917053	8.846e-06	37.230	144.062
pkustk14	151926 x 151926	14836504	6.428e-04	46.508	46.508
com-Youtube	1134890 x 1134890	5975248	4.639e-06	50.754	50.754
as-Skitter	1696415 x 1696415	22190596	7.71e-06	136.861	136.861
sx-stackoverflow	2601977 x 2601977	36233450	5.352e-06	137.849	65.367
ASIC_680	682862 x 682862	3871773	8.303e-06	659.807	659.807

Table 8.3: Large Matrix Dataset. Matrices are sorted by NNZ-r-std, i.e., based on their irregular pattern.

---

# Bibliography

---

- [1] Christina Giannoula, Georgios Goumas, and Nectarios Koziris. Combining HTM with RCU to Speed up Graph Coloring on Multicore Platforms. In *ISC HPC*, 2018.
- [2] Christina Giannoula. ColorTM: A High-Performance Graph Coloring Algorithm. <https://github.com/cgiannoula/ColorTM.git>.
- [3] Christina Giannoula, Athanasios Peppas, Georgios Goumas, and Nectarios Koziris. High-Performance and Balanced Parallel Graph Coloring on Multicore Platforms. *The Journal of Supercomputing*, 2022.
- [4] Foteini Strati, Christina Giannoula, Dimitrios Siakavaras, Georgios Goumas, and Nectarios Koziris. An Adaptive Concurrent Priority Queue for NUMA Architectures. In *CF*, 2019.
- [5] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios I. Goumas, and Onur Mutlu. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In *HPCA*, 2021.
- [6] SAFARI Research Group. *SparseP* Software Package. In <https://github.com/CMU-SAFARI/SparseP>, 2022.
- [7] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures. In *SIGMETRICS*, 2022.
- [8] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems. In *CoRR*, 2022.

- [9] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems. In *CoRR*, 2022.
- [10] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *Proc. ACM Meas. Anal. Comput. Syst.*, 2022.
- [11] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *ISVLSI*, 2022.
- [12] ExaGraph. Grappolo: Parallel clustering using the Louvain method as the serial template. <https://github.com/Exa-Graph/grappolo>.
- [13] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: a Scalable Relaxed Priority Queue. In *PPoPP*, 2015.
- [14] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *SIROCCO*, 2007.
- [15] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (Much) Faster Than You Think. In *SOSP*, 2017.
- [16] Tudor David, Rachid Guerraoui, and 30 Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP*, 2013.
- [17] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. In *PPoPP*, 2011.
- [18] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. In *ACM TOMS*, 2018.
- [19] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.
- [20] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.
- [21] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.

- [22] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*, 2017.
- [23] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*, 2016.
- [24] Jiawen Liu, Hengyu Zhao, Matheus Almeida Ogleari, Dong Li, and Jishen Zhao. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach. In *MICRO*, 2018.
- [25] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Norion, A. Scibisz, S. Subramoneyon, C. Alkan, S. Ghose, and O. Mutlu. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *MICRO*, 2020.
- [26] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alser, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. SeGraM: A Universal Hardware Accelerator for Genomic Sequence-to-Graph and Sequence-to-Sequence Mapping. In *ISCA*, 2022.
- [27] Jeremie Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. In *BMC Genomics*, 2018.
- [28] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian Data Engine. In *ISCA*, 2017.
- [29] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. CoNDA: Efficient Cache Coherence Support for Near-data Accelerators. In *ISCA*, 2019.
- [30] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *PARA*, 2006.
- [31] Jeremy Kepner, David A. Bader, Aydın Buluç, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. In *ICCS*, 2015.
- [32] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

- [33] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*, 2017.
- [34] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *HPCA*, 2018.
- [35] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable PIM-Based Graph Processing. In *MICRO*, 2019.
- [36] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. PIM-Enabled Instructions: A Low-overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.
- [37] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. In *CAL*, 2017.
- [38] Andrew Canning, Giulia Galli, Francesco Mauri, Alessandro De Vita, and Roberto Car.  $O(N)$  tight-binding molecular dynamics on massively parallel computers: An orbital decomposition approach. In *Computer Physics Communications*, 1996.
- [39] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging Gustavson’s Algorithm to Accelerate Sparse Matrix Multiplication. In *ASPLOS*, 2021.
- [40] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.
- [41] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-Performance Conjugate-Gradient Benchmark: A New Metric for Ranking High-Performance Computing Systems. *IJHPCA*, 2016.
- [42] Jack Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozoz, and Karin Remington. Sparse Matrix Libraries in C++ for High Performance Architectures. In *Mathematics*, 1994.
- [43] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms. *arXiv*, 2016.
- [44] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. Conflict-Free Symmetric Sparse Matrix-Vector Multiplication on Multicore Architectures. In *SC*, 2019.
- [45] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *ASPLOS*, 2017.
- [46] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julianne: A Framework for Parallel Graph Algorithms Using Work-Efficient Bucketing. In *SPAA*, 2017.
- [47] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, 2013.



- [48] D. J. A. Welsh and M. B. Powell. An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems. *The Computer Journal*, 1967.
- [49] Hao Lu, Mahantesh Halappanavar, Daniel Chavarría-Miranda, Assefaw Gebremedhin, and Ananth Kalyanaraman. Balanced Coloring for Parallel Computing Applications. In *IEEE IPDPS*, 2015.
- [50] Mark T. Jones and Paul E. Plassmann. A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing*, 1993.
- [51] Mehmet Deveci, Erik G Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Parallel Graph Coloring for Manycore Architectures. In *IPDPS*, 2016.
- [52] Mustafa Kemal Tas, Kamer Kaya, and Erik Saule. Greed Is Good: Parallel Algorithms for Bipartite-Graph Partial Coloring on Multicore Architectures. In *ICPP*, 2017.
- [53] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable Parallel Graph Coloring Algorithms. *Concurrency: Practice and Experience*, 2000.
- [54] Erik G. Boman, Doruk Bozdağ, Umit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers. *EuroPar*, 2005.
- [55] Ümit V. Çatalyürek, John Feo, Assefaw Hadish Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph Coloring Algorithms for Muti-core and Massively Multithreaded Architectures. *Parallel Computing*, 2012.
- [56] Georgios Rokos, Gerard Gorman, and Paul H. J. Kelly. A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures. *Euro-Par*, 2015.
- [57] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering Heuristics for Parallel Graph Coloring. In *SPAA*, 2014.
- [58] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality. In *SC*, 2020.
- [59] Konstantinos Sagonas and Kjell Winblad. The Contention Avoiding Concurrent Priority Queue. In *LCPC*, 2017.
- [60] Håkan Sundell and Philippas Tsigas. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *Journal of Parallel and Distributed Computing*, 2005.
- [61] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The Adaptive Priority Queue with Elimination and Combining. In *DISC*, 2014.

- [62] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *PPoPP*, 2015.
- [63] Hamza Rihani, Peter Sanders, and Roman Dementiev. MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues, 2015.
- [64] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A Parallel Priority Queue with Constant Time Operations. *J. Parallel Distrib. Comput.*, 1998.
- [65] Deli Zhang and Damian Dechev. A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. *TPDS*, 2016.
- [66] Peter Sanders. Randomized Priority Queues for Fast Parallel Access. *J. Parallel Distrib. Comput.*, 1998.
- [67] Konstantinos Sagonas and Kjell Winblad. The Contention Avoiding Concurrent Priority Queue. In *LCPC*, 2016.
- [68] M. Rab, R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia. NUMA-Aware Non-Blocking Calendar Queue. In *DS-RT*, 2020.
- [69] Tyler Crain, Vincent Gramoli, and Michel Raynal. No Hot Spot Non-blocking Skip List. In *ICDCS*, 2013.
- [70] K. Fraser. Practical Lock-Freedom. *PhD thesis, University of Cambridge*, 2004.
- [71] Mikhail Fomitchev and Eric Ruppert. Lock-free Linked Lists and Skip Lists. In *PODC*, 2004.
- [72] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
- [73] Ian Dick, Alan Fekete, and Vincent Gramoli. A Skip List for Multicore. *Concurrency and Computation: Practice and Experience*, 2017.
- [74] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent Data Structures for Near-Memory Computing. In *SPAA*, 2017.
- [75] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 1990.
- [76] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation. In *SPAA*, 2019.
- [77] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *IPDPS*, 2000.
- [78] Jonatan Lindén and Bengt Jonsson. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *OPODIS*, 2013.

- [79] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *SPAA*, 2010.
- [80] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable Flat-Combining Based Synchronous Queues. In *DISC*, 2010.
- [81] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *OPODIS*, 2013.
- [82] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation Locking Libraries for Improved Performance of Multithreaded Programs. In *EuroPar*, 2014.
- [83] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *USENIX ATC*, 2012.
- [84] Darko Petrović, Thomas Ropars, and André Schiper. On the Performance of Delegation over Cache-Coherent Shared Memory. In *ICDCN*, 2015.
- [85] M. Aater Suleman, Onur Mutlu, Moinuddin Qureshi, and Yale Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *ASPLOS*, 2009.
- [86] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. *ASPLOS*, 2017.
- [87] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, 2018.
- [88] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *ICPP*, 2017.
- [89] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP*, 2018.
- [90] Juan Carlos Pichel and Beatriz Pateiro-López. A New Approach for Sparse Matrix Classification Based on Deep Learning Techniques. In *CLUSTER*, 2018.
- [91] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *ICPP*, 2016.
- [92] Peter Grönquist, Chengyuan Yao, Tal Ben-Nun, Nikoli Dryden, Peter Dueben, Shigang Li, and Torsten Hoefler. Deep Learning for Post-Processing Ensemble Weather Forecasts. *Philosophical Transactions of the Royal Society A*, 2021.

- [93] Suejb Memeti, Sabri Pllana, Alécio Binotto, Joanna Kołodziej, and Ivona Brandic. Using Meta-Heuristics and Machine Learning for Software Optimization of Parallel Computing Systems: A Systematic Literature Review. *Computing*, 2019.
- [94] Amlan Kusum, Iulian Neamtiu, and Rajiv Gupta. Safe and Flexible Adaptation via Alternate Data Structure Representations. In *CC*, 2016.
- [95] Donald Michie. 'Memo' Functions and Machine Learning. In *Nature*, 1968.
- [96] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *PPoPP*, 2019.
- [97] Laxman Dhulipala, Changwan Hong, and Julian Shun. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. In *VLDB Endowment*, 2020.
- [98] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *ICS*, 2015.
- [99] Akrem Benatia, W. Ji, Y. Wang, and Feng Shi. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. *ICPP*, 2016.
- [100] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *SIGMOD*, 2020.
- [101] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *ICAC*, 2011.
- [102] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *IPDPS*, 2011.
- [103] Athena Elafrou, G. Goumas, and N. Koziris. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *ICPP*, 2017.
- [104] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. TACO: A Tool to Generate Tensor Algebra Kernels. In *ASE*, 2017.
- [105] Duane Merrill and Michael Garland. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC*, 2016.
- [106] Jeremiah Willcock and Andrew Lumsdaine. Accelerating Sparse Matrix Computations via Data Compression. In *ICS*, 2006.
- [107] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC*, 2007.

- [108] Naveen Namashivayam, Sanyam Mehta, and Pen-Chung Yew. Variable-Sized Blocks for Locality-Aware SpMV. In *CGO*, 2021.
- [109] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huyng, Xibai Li, and Rick Siow Mong Goh. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *CGO*, 2015.
- [110] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *booktitle of Physics: Conference Series*, 2005.
- [111] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *IPDPSW*, 2017.
- [112] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. Sparso: Context-Driven Optimizations of Sparse Linear Algebra. In *PACT*, 2016.
- [113] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. CVR: Efficient Vectorization of SpMV on X86 Processors. In *CGO*, 2018.
- [114] Guoqing Xiao, Kenli Li, Yuedan Chen, Wangquan He, Albert Y. Zomaya, and Tao Li. CASpMV: A Customized and Accelerative SpMV Framework for the Sunway TaihuLight. In *IEEE TPDS*, 2021.
- [115] Kaixi Hou, Wu-chun Feng, and Shuai Che. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *IPDPSW*, 2017.
- [116] Ali Pinar and Michael T. Heath. Improving Performance of Sparse Matrix-Vector Multiplication. In *SC*, 1999.
- [117] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient Sparse Matrix-Vector Multiplication on X86-Based Many-Core Processors. In *ICS*, 2013.
- [118] John Mellor-Crummey and John Garvin. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. In *IJHPCA*, 2004.
- [119] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations. In *SIAM Review*, 2002.
- [120] Richard W. Vuduc and Hyun-Jin Moon. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *HPCC*, 2005.
- [121] S. Toledo. Improving the Memory-System Performance of Sparse-Matrix Vector Multiplication. In *IBM booktitle of Research and Development*, 1997.
- [122] O. Temam and W. Jalby. Characterizing the Behavior of Sparse Algorithms on Caches. In *SC*, 1992.

- [123] Baris Aktemur. A Sparse Matrix-Vector Multiplication Method with Low Preprocessing Cost. In *Concurrency and Computation: Practice and Experience*, 2018.
- [124] Haoran Zhao, Tian Xia, Chenyang Li, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon. In *ICCD*, 2020.
- [125] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM Transactions on Graphics*, 2003.
- [126] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC*, 2018.
- [127] Weifeng Liu and Brian Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *IPDPS*, 2014.
- [128] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. Efficient PageRank and SpMV Computation on AMD GPUs. In *ICPP*, 2010.
- [129] Ping Guo, Liqiang Wang, and Po Chen. A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs. In *IEEE TPDS*, 2014.
- [130] Bor-Yiing Su and Kurt Keutzer. ClSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *ICS*, 2012.
- [131] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU. In *ICS*, 2017.
- [132] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. YaSpMV: Yet Another SpMV Framework on GPUs. In *PPoPP*, 2014.
- [133] Nathan Bell and Michael Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC*, 2009.
- [134] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *PPoPP*, 2010.
- [135] Juan C. Pichel, Francisco F. Rivera, Marcos Fernández, and Aurelio Rodríguez. Optimization of Sparse Matrix-Vector Multiplication Using Reordering Techniques on GPUs. In *Microprocess. Microsyst.*, 2012.
- [136] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan, and Li Rao. Optimizing SpMV for Diagonal Sparse Matrices on GPU. In *ICPP*, 2011.
- [137] F. Vázquez, J. J. Fernández, and E. M. Garzón. A New Approach for Sparse Matrix Vector Product on NVIDIA GPUs. In *Concurrency and Computation: Practice and Experience*, 2011.

- [138] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. In *Proc. VLDB Endow.*, 2011.
- [139] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. BASMAT: Bottleneck-Aware Sparse Matrix-Vector Multiplication Auto-Tuning on GPGPUs. In *PPoPP*, 2019.
- [140] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse Matrix-Vector Multiplication on GPGPUs. In *ACM TOMS*, 2017.
- [141] Seyong Lee and Rudolf Eigenmann. Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems. In *ICS*, 2008.
- [142] Rob H. Bisseling and Wouter Meesen. Communication Balancing in Parallel Sparse Matrix-Vector Multiplication. In *ETNA. Electronic Transactions on Numerical Analysis*, 2005.
- [143] Beata Bylina, Jarosław Bylina, Przemysław Stpoczyński, and Dominik Szalkowski. Performance Analysis of Multicore and Multinodal Implementation of SpMV Operation. In *FedCSIS*, 2014.
- [144] Brian A. Page and Peter M. Kogge. Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication. In *HPCS*, 2018.
- [145] Enver Kayaaslan, Bora Uçar, and Cevdet Aykanat. Semi-Two-Dimensional Partitioning for Parallel Sparse Matrix-Vector Multiplication. In *IPDPS Workshop*, 2015.
- [146] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. Towards Efficient SpMV on Sunway Manycore Architectures. In *ICS*, 2018.
- [147] U.V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. In *IEEE TPDS*, 1999.
- [148] Brendan Vastenhouw and Rob H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. In *SIAM Rev.*, 2005.
- [149] S.G. Nastea, O. Frieder, and T. El-Ghazawi. Load-Balancing in Sparse Matrix-Vector Multiplication. In *SPDP*, 1996.
- [150] Daniël M. Pelt and Rob H. Bisseling. A Medium-Grain Method for Fast 2D Bipartitioning of Sparse Matrices. In *IPDPS*, 2014.
- [151] Anael Grandjean, Johannes Langguth, and Bora Uçar. On Optimal and Balanced Sparse Matrix Partitioning Problems. In *CLUSTER*, 2012.
- [152] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *SC*, 2013.
- [153] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *MICRO*, 2019.

- [154] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient Speculative Parallelism for Accelerators. In *ASPLOS*, 2020.
- [155] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *SIGMETRICS*, 2022.
- [156] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. In *IEEE Access*, 2022.
- [157] F. Devaux. The True Processing In Memory Accelerator. In *Hot Chips*, 2019.
- [158] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing Data Where It Makes Sense: Enabling In-Memory Computation. In *MICPRO*, 2019.
- [159] Saugata Ghose, Amirali Boroumand, Jeremie Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-Memory: A Workload-Driven Perspective. In *IBM JRD*, 2019.
- [160] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. In *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
- [161] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. In *IGSC*, 2021.
- [162] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. In *CoRR*, 2021.
- [163] Harold S. Stone. A Logic-in-Memory Computer. In *IEEE TC*, 1970.
- [164] W. H. Kautz. Cellular Logic-in-Memory Arrays. In *IEEE TC*, 1969.
- [165] David Elliot Shaw, Salvatore J. Stolfo, Hussein Ibrahim, Bruce Hillyer, Gio Wiederhold, and JA Andrews. The NON-VON Database Machine: A Brief Overview. *IEEE Database Eng. Bull.*, 1981.
- [166] P. M. Kogge. EXECUBE - A New Architecture for Scaleable MPPs. In *ICPP*, 1994.
- [167] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer*, 1995.
- [168] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 1997.



- [169] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *ISCA*, 1998.
- [170] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *ICCD*, 1999.
- [171] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA*, 2000.
- [172] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The Architecture of the DIVA Processing-in-Memory Chip. In *SC*, 2002.
- [173] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *HPCA*, 2017.
- [174] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural Cache: Bit-Serial in-Cache Acceleration of Deep Neural Networks. In *ISCA*, 2018.
- [175] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality Cache for Data Parallel Acceleration. In *ISCA*, 2019.
- [176] Mingu Kang, Min-Sun Keel, Naresh R Shanbhag, Sean Eilert, and Ken Curewitz. An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM. In *ICASSP*, 2014.
- [177] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.
- [178] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM. arXiv:1611.09988 [cs:AR], 2016.
- [179] Vivek Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015.
- [180] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Genady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *MICRO*, 2013.
- [181] Shaahin Angizi and Deliang Fan. GraphiDe: A Graph Processing Accelerator Leveraging In-DRAM-Computing. In *GLSVLSI*, 2019.

- [182] J. Kim, M. Patel, H. Hassan, and O. Mutlu. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices. In *HPCA*, 2018.
- [183] J. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *HPCA*, 2019.
- [184] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.
- [185] Kevin K. Chang, Prashant J. Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K. Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016.
- [186] Xin Xin, Youtao Zhang, and Jun Yang. ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM. In *HPCA*, 2020.
- [187] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator. In *MICRO*, 2017.
- [188] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang. DrAcc: A DRAM Based Accelerator for Accurate CNN Inference. In *DAC*, 2018.
- [189] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDram: A Framework for Bit-Serial SIMD Processing Using DRAM. In *ASPLOS*, 2021.
- [190] Seyyed Hossein SeyyedAghaei Rezaei, Mehdi Modarressi, Rachata Ausavarungnirun, Mohammad Sadrosadati, Onur Mutlu, and Masoud Daneshtalab. NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories. In *IEEE CAL*, 2020.
- [191] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S. Kim, Juan Gómez-Luna, Mohammad Sadrosadati, Nika Mansouri-Ghiasi, and Onur Mutlu. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *MICRO*, 2020.
- [192] Mustafa F. Ali, Akhilesh Jaiswal, and Kaushik Roy. In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology. In *IEEE TCSI*, 2020.
- [193] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A Processing-In-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In *DAC*, 2016.
- [194] S. Angizi, Z. He, and D. Fan. PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-efficient Logic Computation. In *DAC*, 2018.

- [195] S. Angizi, A. S. Rakin, and D. Fan. CMP-PIM: An Energy-efficient Comparator-based Processing-in-Memory Neural Network Accelerator. In *DAC*, 2018.
- [196] S. Angizi, J. Sun, W. Zhang, and D. Fan. AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM. In *DAC*, 2019.
- [197] Yifat Levy, Jehoshua Bruck, Yuval Cassuto, Eby G. Friedman, Avinoam Kolodny, Eitan Yaakobi, and Shahar Kvatinsky. Logic Operations in Memory Using a Memristive Akers Array. *Microelectronics Journal*, 2014.
- [198] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC—Memristor-Aided Logic. *IEEE TCAS II: Express Briefs*, 2014.
- [199] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ISCA*, 2016.
- [200] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman. Memristor-Based IMPLY Logic Design Procedure. In *ICCD*, 2011.
- [201] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies. *TVLSI*, 2014.
- [202] P.-E. Gaillardon, L. Amaru, A. Siemon, and et al. The Programmable Logic-in-Memory (PLiM) Computer. In *DATE*, 2016.
- [203] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. ReVAMP: ReRAM based VLIW Architecture for In-Memory Computing. In *DATE*, 2017.
- [204] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, and Jan van Lunteren. Memristor Based Computation-In-Memory Architecture for Data-Intensive Applications. In *DATE*, 2015.
- [205] L. Xie, H. A. D. Nguyen, M. Taouil, and et al. Fast Boolean Logic Papped on Memristor Crossbar. In *ICCD*, 2015.
- [206] S. Hamdioui, S. Kvatinsky, and et al. G. Cauwenberghs. Memristor for Computing: Myth or Reality? In *DATE*, 2017.
- [207] Jintao Yu, Hoang Anh Du Nguyen, Lei Xie, Mottaqiallah Taouil, and Said Hamdioui. Memristive Devices for Computation-In-Memory. In *DATE*, 2018.
- [208] Ivan Fernandez, Ricardo Quisilant, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, Eladio Gutiérrez, Oscar Plata, and Onur Mutlu. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *ICCD*, 2020.

- [209] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. *BMC Genomics*, 2018.
- [210] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA*, 2016.
- [211] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*, 2015.
- [212] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*, 2015.
- [213] Mingyu Gao and Christos Kozyrakis. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *HPCA*, 2016.
- [214] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Yuan Xie, and Eren Kursun. Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges. In *GLSVLSI*, 2016.
- [215] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale N Patt, et al. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*, 2016.
- [216] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.
- [217] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters. In *SC*, 2015.
- [218] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. In *IEEE TCAD*, 2018.
- [219] João Dinis Ferreira, Gabriel Falcao, Juan Gómez-Luna, Mohammed Alser, Lois Orosa, Mohammad Sadrosadati, Jeremie S. Kim, Geraldo F. Oliveira, Taha Shahroodi, Anant Nori, and Onur Mutlu. pLUTo: Enabling Massively Parallel Computation In DRAM via Lookup Tables. In *CoRR*, 2021.
- [220] Ataberk Olgun, Minesh Patel, A. Giray Yağlikçi, Haocong Luo, Jeremie S. Kim, F. Nisa Bostanci, Nandita Vijaykumar, Oğuz Ergin, and Onur Mutlu. QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips. In *ISCA*, 2021.

- [221] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *FPL*, 2020.
- [222] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems. In *MICRO*, 2016.
- [223] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. JAFAR: Near-Data Processing for Databases. In *SIGMOD*, 2015.
- [224] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and Reducing Memory Interference in COTS-based Multi-core Systems. *Real-Time Systems*, 2016.
- [225] Amir Morad, Leonid Yavits, and Ran Ginosar. GP-SIMD Processing-in-Memory. *ACM TACO*, 2015.
- [226] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In *PACT*, 2016.
- [227] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*, 2014.
- [228] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry Pileggi, and Franz Franchetti. Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-In-Memory Hardware. In *HPEC*, 2013.
- [229] Alain Denzler, Rahul Bera, Nastaran Hajinazar, Gagandeep Singh, Geraldo F Oliveira, Juan Gómez-Luna, and Onur Mutlu. Casper: Accelerating stencil computation using near-cache processing. *arXiv preprint arXiv:2112.14216*, 2021.
- [230] Amirali Boroumand, Saugata Ghose, Geraldo F Oliveira, and Onur Mutlu. Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design. *arXiv:2103.00798 [cs.AR]*, 2021.
- [231] Amirali Boroumand, Saugata Ghose, Geraldo F Oliveira, and Onur Mutlu. Polynesia: Enabling Effective Hybrid Transactional Analytical Databases with Specialized Hardware Software Co-Design. In *ICDE*, 2022.
- [232] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications. *IEEE Micro*, 2021.

- [233] Gagandeep Singh, Dionysios Diamantopoulos, Juan Gómez-Luna, Christoph Hagleitner, Sander Stuijk, Henk Corporaal, and Onur Mutlu. Accelerating Weather Prediction using Near-Memory Reconfigurable Fabric. *ACM TRETS*, 2021.
- [234] Jose M Herruzo, Ivan Fernandez, Sonia González-Navarro, and Oscar Plata. Enabling Fast and Energy-Efficient FM-Index Exact Matching Using Processing-Near-Memory. *The Journal of Supercomputing*, 2021.
- [235] Leonid Yavits, Roman Kaplan, and Ran Ginosar. GIRAF: General Purpose In-Storage Resistive Associative Framework. *IEEE TPDS*, 2021.
- [236] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *HPCA*, 2021.
- [237] Vivek Seshadri and Onur Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers, Volume 106*. 2017.
- [238] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory. *arXiv preprint arXiv:2204.02085*, 2022.
- [239] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory. In *HICOMB*, 2022.
- [240] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-Memory Data Parallel Processor. In *ASPLOS*, 2018.
- [241] Yue Zha and Jing Li. Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization. In *ISCA*, 2020.
- [242] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. *CoRR*, 2018.
- [243] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *2013 5th IEEE International Memory Workshop*, 2013.
- [244] Onur Mutlu and Lavanya Subramanian. Research Problems and Opportunities in Memory Systems. *Supercomput. Front. Innov.: Int. J.*, 2014.
- [245] Vivek Seshadri and Onur Mutlu. In-DRAM Bulk Bitwise Execution Engine. *CoRR*, 2019.
- [246] Vivek Seshadri and Onur Mutlu. Chapter Four - Simple Operations in Memory to Reduce Data Movement. volume 106 of *Advances in Computers*, pages 107–166. Elsevier, 2017.

- [247] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.
- [248] Berkin Akin, Franz Franchetti, and James C. Hoe. Data Reorganization in Memory Using 3D-Stacked DRAM. In *ISCA*, 2015.
- [249] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing. In *IPDPS*, 2020.
- [250] Paulo C Santos, Geraldo F Oliveira, Diego G Tomé, Marco AZ Alves, Eduardo C Almeida, and Luigi Carro. Operand Size Reconfiguration for Big Data Processing in Memory. In *DATE*, 2017.
- [251] Wen-Mei Hwu, Izzat El Hajj, Simon Garcia De Gonzalo, Carl Pearson, Nam Sung Kim, Deming Chen, Jinjun Xiong, and Zehra Sura. Rebooting the Data Access Hierarchy of Computing Systems. In *ICRC*, 2017.
- [252] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO*, 2021.
- [253] Scott Lloyd and Maya Gokhale. In-memory Data Rearrangement for Irregular, Data-intensive Computing. *Computer*, 2015.
- [254] Duncan G Elliott, Michael Stumm, W Martin Snelgrove, Christian Cojocar, and Robert McKenzie. Computational RAM: Implementing Processors in Memory. *IEEE Design & Test of Computers*, 1999.
- [255] Le Zheng, Sangho Shin, Scott Lloyd, Maya Gokhale, Kyungmin Kim, and Sung-Mo Kang. RRAM-based TCAMs for pattern search. In *ISCAS*, 2016.
- [256] Joshua Landgraf, Scott Lloyd, and Maya Gokhale. Combining Emulation and Simulation to Evaluate a Near Memory Key/Value Lookup Accelerator, 2021.
- [257] Arun Rodrigues, Maya Gokhale, and Gwendolyn Voskuilen. Towards a Scatter-Gather Architecture: Hardware and Software Issues. In *MEMSYS*, 2019.
- [258] Scott Lloyd and Maya Gokhale. Design Space Exploration of Near Memory Accelerators. In *MEMSYS*, 2018.
- [259] Scott Lloyd and Maya Gokhale. Near Memory Key/Value Lookup Acceleration. In *MEMSYS*, 2017.

- [260] Maya Gokhale, Scott Lloyd, and Chris Hajas. Near Memory Data Structure Rearrangement. In *MEMSYS*, 2015.
- [261] Arpith C Jacob, Zehra Sura, Tong Chen, Carlo Bertolli, Samuel Antao, Olivier Sallenave, Kevin O'Brien, Hans Jacobson, Ravi Nair, Jose R Brunheroto, et al. Compiling for the Active Memory Cube. Technical report, Tech. rep. RC25644 (WAT1612-008). IBM Research Division, 2016.
- [262] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, et al. Data Access Optimization in a Processing-in-Memory System. In *CF*, 2015.
- [263] Ravi Nair. Evolution of Memory Architecture. *Proceedings of the IEEE*, 2015.
- [264] Yue Xi, Bin Gao, Jianshi Tang, An Chen, Meng-Fan Chang, Xiaobo Sharon Hu, Jan Van Der Spiegel, He Qian, and Huaqiang Wu. In-Memory Learning With Analog Resistive Switching Memory: A Review and Perspective. *Proceedings of the IEEE*, 2020.
- [265] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F. Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *DAC*, 2019.
- [266] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *ISPASS*, 2014.
- [267] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, and et al. Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems. In *IBM JRD*, 2015.
- [268] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 2014.
- [269] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. AL-RESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *HPCA*, 2020.
- [270] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. ExTensor: An Accelerator for Sparse Tensor Algebra. In *MICRO*, 2019.
- [271] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In *MICRO*, 2016.
- [272] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *HPCA*, 2018.



- [273] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. Hardware Accelerator for Analytics of Sparse Data. In *DAC*, 2016.
- [274] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *HPCA*, 2020.
- [275] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *MICRO*, 2018.
- [276] Asit K. Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. Fine-grained Accelerators for Sparse Machine Learning Workloads. In *ASP-DAC*, 2017.
- [277] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging Gustavson’s Algorithm to Accelerate Sparse Matrix Multiplication. In *ASPLOS*, 2021.
- [278] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *HPCA*, 2020.
- [279] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *ISCA*, 2020.
- [280] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *HPCA*, 2020.
- [281] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization. In *MICRO*, 2019.
- [282] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *FCCM*, 2014.
- [283] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *FCCM*, 2015.
- [284] Colin Yu Lin, Zheng Zhang, Ngai Wong, and Hayden Kwok-Hay So. Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs. In *FPT*, 2010.
- [285] Yaman Umuroglu and Magnus Jahre. An energy efficient column-major backend for fpga spmv accelerators. In *ICCD*, 2014.
- [286] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. SMASH: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *MICRO*, 2019.

- [287] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *MICRO*, 2018.
- [288] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine. In *CASES*, 2015.
- [289] J. Leskovec and R. Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. In *TIST*, 2016.
- [290] A. Smith. 6 New Facts About Facebook. In *http://mediashift.org*, 2019.
- [291] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *PDP*, 2008.
- [292] J.B. White and P. Sadayappan. On Improving the Performance of Sparse Matrix-Vector Multiplication. In *HIPC*, 1997.
- [293] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *ICS*, 2021.
- [294] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures. In *J. Supercomput.*, 2009.
- [295] Albert-László Barabási. Scale-Free Networks: A Decade and Beyond. *Science*, 2009.
- [296] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *HPDC*, 2017.
- [297] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (Much) Faster Than You Think. In *SOSP*, 2017.
- [298] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. Locking Made Easy. In *Middleware*, 2016.
- [299] Ching-Kai Liang and Milos Prvulovic. MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management. In *ISCA*, 2015.
- [300] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-Core Architectures. In *ISCA*, 2007.
- [301] Jose L Abellán, Juan Fernández, Manuel E Acacio, et al. Glocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *IPDPS*, 2011.

- [302] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman Jouppi, Mike Schlansker, and Brad Calder. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *MICRO*, 2006.
- [303] José L Abellán, Juan Fernández, and Manuel E Acacio. A g-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs. In *ICPP*, 2010.
- [304] Jungju Oh, Milos Prvulovic, and Alenka Zajic. TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines. In *ISCA*, 2011.
- [305] Sergi Abadal, Albert Cabellos-Aparicio, Eduard Alarcon, and Josep Torrellas. WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication. In *ASPLOS*, 2016.
- [306] Bilge Saglam Akgul, Jaehwan Lee, and Vincent John Mooney. A System-on-a-Chip Lock Cache with Task Preemption Support. In *CASES*, 2001.
- [307] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural Support for Fair Reader-Writer Locking. In *MICRO*, 2010.
- [308] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, and et al. The Network Architecture of the Connection Machine CM-5 (Extended Abstract). In *SPAA*, 1992.
- [309] Stefanos Kaxiras and Georgios Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 2010.
- [310] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.
- [311] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-Cache Coherence. In *ISCA*, 2013.
- [312] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. *ASPLOS*, 2013.
- [313] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *ISCA*, 1995.
- [314] Alberto Ros and Stefanos Kaxiras. Complexity-Effective Multicore Coherence. In *PACT*, PACT 2012.
- [315] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, 1997.

- [316] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. *ICS*, 1990.
- [317] Allan Gottlieb, Ralph Grishman, Clyde Kruskal, Kevin McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine. In *ISCA*, 1982.
- [318] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *SIGMOD*, 2014.
- [319] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-Memory: A Workload-Driven Perspective. *IBM JRD*, 2019.
- [320] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Performance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In *ICPP*, 2009.
- [321] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. In *IJHPCA*, 2004.
- [322] Richard Wilson Vuduc and James W. Demmel. Automatic performance tuning of sparse matrix kernels. In *PhD Thesis*, 2003.
- [323] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC*, 2002.
- [324] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks. In *PACT*, 2021.
- [325] Google LLC. Edge TPU.
- [326] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. In *IEEE Access*, 2021.
- [327] Kaige Yan, Xingyao Zhang, and Xin Fu. Characterizing, Modeling, and Improving the QoE of Mobile Devices with Low Battery Level. In *MICRO*, 2015.
- [328] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The Locality Descriptor: A Holistic Cross-layer Abstraction to Express Data Locality in GPUs. In *ISCA*, 2018.
- [329] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *ISCA*, 2018.

- [330] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-reliability Memory. In *DSN*, 2014.
- [331] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference using Approximate DRAM. In *MICRO*, 2019.
- [332] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo Francisco de Oliveira Jr, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *ISCA*, 2020.
- [333] Nandita Vijaykumar, Ataberk Olgun, Konstantinos Kanellopoulos, F. Nisa Bostanci, Hasan Hassan, Mehrshad Lotfi, Phillip B. Gibbons, and Onur Mutlu. MetaSys: A Practical Open-Source Metadata Management System to Implement and Evaluate Cross-Layer Optimizations. *ACM TACO*, 2022.
- [334] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees. In *PACT*, PACT 2017.
- [335] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *SPAA*, 2016.
- [336] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.
- [337] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *SC*, 2013.
- [338] UPMEM. UPMEM Website. In <https://www.upmem.com>, 2020.
- [339] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems. In *MICRO*, 2016.
- [340] Sukhan Lee, Shin-Haeng Kang, Jaehoon Lee, H. Kim, Eojin Lee, Seung young Seo, H. Yoon, Seungwon Lee, K. Lim, Hyunsung Shin, Jinhyun Kim, O. Seongil, Anand Iyer, David Wang, K. Sohn, and N. Kim. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *ISCA*, 2021.

- [341] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *ISSCC*, 2021.
- [342] Dániel Marx. Graph Coloring Problems and Their Applications in Scheduling. In *Proc. John Von Neumann PhD Students Conference*, 2004.
- [343] E. M. Arkin and E. B. Silverberg. Scheduling Jobs with Fixed Start and End Times. *Discrete Applied Mathematics*, 1987.
- [344] Dániel Marx. Graph Colouring Problems and their Applications in Scheduling. *Periodica Polytechnica Electrical Engineering*, 2004.
- [345] R. Ramaswami and K.K. Parhi. Distributed Scheduling of Broadcasts in a Radio Network. In *IEEE INFOCOM*, 1989.
- [346] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 1981.
- [347] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *SIGPLAN Symposium on Compiler Construction*, 1982.
- [348] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *TOPLAS*, 1994.
- [349] Wei-Yu Chen, Guei-Yuan Lueh, Pratik Ashar, Kaiyu Chen, and Buqi Cheng. Register Allocation for Intel Processor Graphics. In *CGO*, 2018.
- [350] Albert Cohen and Erven Rohou. Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems. In *DAC*, 2010.
- [351] Thomas F. Coleman and Jorge J. Moré. Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. *SIAM Journal on Numerical Analysis*, 1983.
- [352] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.
- [353] Mark T. Jones and Paul E. Plassmann. The Efficient Parallel Iterative Solution of Large Sparse Linear Systems. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, 1993.
- [354] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothén. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 2005.

- [355] Santo Fortunato. Community Detection in Graphs. *Physics Reports*, 2010.
- [356] Vladimir Kolmogorov. Blossom V: A new Implementation of a Minimum Cost Perfect Matching Algorithm. 2009.
- [357] Dominique Lasalle and George Karypis. Multi-threaded Graph Partitioning. In *IPDPS*, 2013.
- [358] Mikkel Thorup. Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. In *STOC*, 2003.
- [359] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.
- [360] R. C. Prim. Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal*, 1957.
- [361] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. Ladder Queue: An  $O(1)$  Priority Queue Structure for Large-scale Discrete Event Simulation. *TOMACS*, 2005.
- [362] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A Non-Blocking Priority Queue for the Pending Event Set. In *SIMUTOOLS*, 2016.
- [363] Yuming Xu, Keqin Li, and Jingtong Hu. A Genetic Algorithm for Task Scheduling on Heterogeneous Computing Systems using Multiple Priority Queues. *Information Sciences*, 2014.
- [364] Azin Heidarshenas, Tanmay Gangwani, Serif Yesil, Adam Morrison, and Josep Torrellas. Snug: Architectural support for relaxed concurrent priority queueing in chip multiprocessors. In *ICS*, 2020.
- [365] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.
- [366] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. Memory-Centric System Interconnect Design with Hybrid Memory Cubes. In *PACT*, 2013.
- [367] Po-An Tsai, Changping Chen, and Daniel Sanchez. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *MICRO*, 2018.
- [368] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. Transparent Offloading and Mapping: Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*, 2016.
- [369] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent Data Structures for Near-Memory Computing. In *SPAA*, 2017.
- [370] Ayse Yilmazer and David Kaeli. HQL: A Scalable Synchronization Mechanism for GPUs. In *IPDPS*, 2013.

- [371] Ahmed ElTantawy and Tor M Aamodt. Warp Scheduling for Fine-Grained Synchronization. In *HPCA*, 2018.
- [372] Richard E. Kessler and James L. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. *Digest of Papers. Compcon Spring*, 1993.
- [373] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS*, 1996.
- [374] Lixin Zhang, Zhen Fang, and John B Carter. Highly Efficient Synchronization based on Active Memory Operations. In *IPDPS*, 2004.
- [375] Harry F. Jordan. Performance Measurements on HEP - a Pipelined MIMD Computer. *ISCA*, 1983.
- [376] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. *ICPP*, 1978.
- [377] William Dally, J. Stuart Fiske, John Keen, Richard Lethin, Michael Noakes, Peter Nuth, Roy Davison, and Gregory Fyler. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 1992.
- [378] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *ISCA*, 1998.
- [379] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS*, 1989.
- [380] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer*, 1992.
- [381] Kai Wang, Don Fussell, and Calvin Lin. Fast Fine-Grained Global Synchronization on GPUs. In *ASPLOS*, 2019.
- [382] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *ICS*, 2015.
- [383] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *HPCA*, 2021.
- [384] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *CF*, 2008.



- [385] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture. In *ISCA*, 2020.
- [386] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. Near Data Acceleration with Concurrent Host Access. In *ISCA*, 2020.
- [387] Benjamin Y. Cho, Jeageun Jung, and Mattan Erez. Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators. In *SC*, 2021.
- [388] Pranith Kumar and Hyesoon Kim. Parallel Hash Table Design for NDP Systems. In *MEMSYS*, 2020.
- [389] Anirban Nag and Rajeev Balasubramonian. OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations. In *MICRO*, 2021.
- [390] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *MICRO*, 2021.
- [391] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration. In *MICRO*, 2021.
- [392] Peng Gu, Xinfeng Xie, Shuangchen Li, Dimin Niu, Hongzhong Zheng, Krishna T. Malladi, and Yuan Xie. DLUX: A LUT-Based Near-Bank Accelerator for Data Center Deep Learning Training Workloads. In *IEEE TCAD*, 2021.
- [393] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *MEMSYS*, 2019.
- [394] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM. In *IEEE TCAD*, 2018.
- [395] Seunghwan Cho, Haerang Choi, Eunhyeok Park, Hyunsung Shin, and Sungjoo Yoo. McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge. In *IEEE Access*, 2020.
- [396] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. In-DRAM near-Data Approximate Acceleration for GPUs. In *PACT*, 2018.
- [397] Marco A. Z. Alves, Paulo C. Santos, Matthias Diener, and Luigi Carro. Opportunities and Challenges of Performing Vector Operations inside the DRAM. In *MEMSYS*, 2015.
- [398] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *MICRO*, 2017.

- [399] UPMEM. Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper). 2018.
- [400] Dominique Lavenier, Remy Cimadomo, and Romaric Jodin. Variant Calling Parallelization on Processor-in-Memory Architecture. In *BIBM*, 2020.
- [401] Dominique Lavenier, Jean-Francois Roy, and David Furodet. DNA Mapping Using Processor-in-Memory Architecture. In *BIBM*, 2016.
- [402] Vasileios Zois, Divya Gupta, Vassilis J. Tsotras, Walid A. Najjar, and Jean-Francois Roy. Massively Parallel Skyline Computation for Processing-in-Memory Architectures. In *PACT*, 2018.
- [403] Joel Nider, Craig Mustard, Andrada Zoltan, and Alexandra Fedorova. Processing in Storage Class Memory. In *HotStorage*, 2020.
- [404] SAFARI Research Group. PrIM Benchmark Suite. In <https://github.com/CMUSAFARI/prim-benchmarks>, 2021.
- [405] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System, 2022.
- [406] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling. *ACM TOPC*, 2016.
- [407] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. Executing Dynamic Data-graph Computations Deterministically Using Chromatic Scheduling. In *SPAA*, 2014.
- [408] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *STOC*, 1974.
- [409] Daniel Brélaz. New Methods to Color the Vertices of a Graph. *Communications of ACM*, 1979.
- [410] David W. Matula and Leland L. Beck. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM*, 1983.
- [411] Richard M. Karp and Avi Wigderson. A Fast Parallel Algorithm for the Maximal Independent Set Problem. *Journal of the ACM*, 1985.
- [412] M Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *STOC*, 1985.
- [413] Mark Goldberg and Thomas Spencer. A New Parallel Algorithm for the Maximal Independent Set Problem. In *SFCS*, 1987.
- [414] John Mitchem. On Various Algorithms for Estimating the Chromatic Number of a Graph. *The Computer Journal*, 1976.

- [415] László Miklós Lovász, Michael E. Saks, and William T. Trotter. An On-Line Graph Coloring Algorithm with Sublinear Performance Ratio. *Discrete Mathematics*, 1989.
- [416] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *ISCA*, 2013.
- [417] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, 2012.
- [418] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *WWW 2004*, 2004.
- [419] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel Heuristics for Scalable Community Detection. *Parallel Computing*, 2015.
- [420] Daniel Chavarria-Miranda, Mahantesh Halappanavar, and Ananth Kalyanaraman. Scaling Graph Community Detection on the Tiler Many-Core Architecture. In *HiPC*, 2014.
- [421] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Communities in Large Networks. *JSTAT*, 2008.
- [422] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed Louvain Algorithm for Graph Community Detection. In *IPDPS*, 2018.
- [423] Md. Naim, Fredrik Manne, Mahantesh Halappanavar, and Antonino Tumeo. Community Detection on the GPU. In *IPDPS*, 2017.
- [424] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable Static and Dynamic Community Detection using Grappolo. In *HPEC*, 2017.
- [425] Andre Vincent Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. Evaluating Graph Coloring on GPUs. In *PPoPP*, 2011.
- [426] Muhammad Osama, Minh Truong, Carl Yang, Aydın Buluç, and John Owens. Graph Coloring on the GPU. In *IPDPSW*, 2019.
- [427] Xuhao Chen, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang, and Canqun Yang. Efficient and high-quality sparse graph coloring on gpus. *Concurrency and Computation: Practice and Experience*, 2017.
- [428] Shuai Che, Gregory Rodgers, Brad Beckmann, and Steve Reinhardt. Graph Coloring on the GPU and Some Techniques to Improve Load Imbalance. In *IPDPS*, 2015.

- [429] Ghadeer Alabandi, Evan Powers, and Martin Burtcher. Increasing the Parallelism of Graph Coloring via Shortcutting. In *PpopP*, 2020.
- [430] Ian Holyer. The NP-Completeness of Edge-Coloring. *SIAM Journal on Computing*, 1981.
- [431] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In *SC*, 2016.
- [432] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Effective and Efficient Dynamic Graph Coloring. *VLDB*, 2017.
- [433] Jakob Bossek, Frank Neumann, Pan Peng, and Dirk Sudholt. Runtime Analysis of Randomized Search Heuristics for Dynamic Graph Coloring. In *GECCO*, 2019.
- [434] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André Renssen, Marcel Roelofsen, and Sander Verdonschot. Dynamic Graph Coloring. *Algorithmica*, 2019.
- [435] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic Algorithms for Graph Coloring. In *ACM SIAM*, 2018.
- [436] Shay Solomon and Nicole Wein. Improved Dynamic Graph Coloring. *TALG*, 2020.
- [437] Doruk Bozdağ, Ümit V. Çatalyürek, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, and Füsün Özgüner. Distributed-Memory Parallel Algorithms for Distance-2 Coloring and Related Problems in Derivative Computation. *SIAM Journal on Scientific Computing*, 2010.
- [438] Doruk Bozdağ, Ümit V. Çatalyürek, Assefaw Hadish Gebremedhin, Fredrik Manne, Erik G. Boman, and Füsün Özgüner. A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers. In *HPCC*, 2005.
- [439] Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. A Reduction based Method for Coloring Very Large Graphs. In *IJCAI*, 2017.
- [440] Anurag Verma, Austin Buchanan, and Sergiy Butenko. Solving the Maximum Clique and Vertex Coloring Problems on Very Large Sparse Networks. *INFORMS Journal on Computing*, 2015.
- [441] Emmanuel Hebrard and George Katsirelos. A Hybrid Approach for Exact Coloring of Massive Graphs. In *CPAIOR*, 2019.
- [442] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In *DISC*, 2018.
- [443] Tudor Alexandru David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *ASPLOS*, 2015.

- [444] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. RCU-HTM: A Generic Synchronization Technique for Highly Efficient Concurrent Search Trees. *CCPE*, 2021.
- [445] Tim Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *DISC*, 2001.
- [446] Maged M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *SPAA*, 2002.
- [447] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *PODC*, 2010.
- [448] Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, 2012.
- [449] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP*, 2014.
- [450] Lavanya Doddipalli and K. Rani. Ensemble Decision Tree Classifier For Breast Cancer Data. *International Journal of Information Technology Convergence and Services*, 2012.
- [451] Kemal Polat and Salih Güneş. A Novel Hybrid Intelligent Method Based on C4.5 Decision Tree Classifier and One-against-All Approach for Multi-Class Classification Problems. *Expert Syst. Appl.*, 2009.
- [452] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra. In *DSN*, 2012.
- [453] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 2011.
- [454] L. T. Schermerhorn. Automatic Page Migration for Linux [A Matter of Hygiene]. 2007.
- [455] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI*, 2010.
- [456] Mingzhe Zhang, Haibo Chen, Luwei Cheng, Francis C. M. Lau, and Cho-Li Wang. Scalable Adaptive NUMA-Aware Lock. *TPDS*, 2017.
- [457] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *PACT*, 2009.

- [458] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-Free Contention Adapting Search Trees. In *SPAA*, 2018.
- [459] Rachid Guerraoui and Vasileios Trigonakis. Optimistic Concurrency with OPTIK. In *PPoPP*, PPOPP 2016.
- [460] Shane V. Howley and Jeremy Jones. A Non-Blocking Internal Binary Search Tree. In *SPAA*, 2012.
- [461] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *PPoPP*, 2010.
- [462] Mayank Rawat and Ajay D. Kshemkalyani. SWIFT: Scheduling in Web Servers for Fast Response Time. In *NCA*, 2003.
- [463] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based Scheduling to Improve Web Performance. *TOCS*, 2003.
- [464] Lisa A. Torrey, Joyce Coleman, and Barton Miller. A Comparison of Interactivity in the Linux 2.6 Scheduler and an MLFQ Scheduler. *Software Practice and Experience.*, 2007.
- [465] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. Sibyl: Adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.
- [466] Maurice Herlihy. Wait-free Synchronization. *TOPLAS*, 1991.
- [467] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. HBM DRAM Technology and Architecture. In *IMW*, 2017.
- [468] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the Characteristics of 3D-stacked Memories: A case Study for Hybrid Memory Cube. In *IISWC*, 2017.
- [469] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *CAL*, 2015.
- [470] hybridmemorycube.org. Hybrid Memory Cube Specification rev. 2.1. *Hybrid Memory Cube Consortium*, 2015.
- [471] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 25.2 A 1.2V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV. In *ISSCC*, 2014.

- [472] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. In *TACO*, 2016.
- [473] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and R. Ausavarungnirun. A Modern Primer on Processing in Memory. *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
- [474] Maya Gokhale, Scott Lloyd, and Chris Hajas. Near Memory Data Structure Rearrangement. In *MEMSYS*, 2015.
- [475] Amirhossein Mirhosseini and Josep Torrellas. Survive: Pointer-Based In-DRAM Incremental Check-Pointing for Low-Cost Data Persistence and Rollback-Recovery. *CAL*, 2016.
- [476] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*, 2016.
- [477] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *FPL*, 2020.
- [478] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. Near-Memory Computing: Past, Present, and Future. In *MICPRO*, 2019.
- [479] Ahsan Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server. In *BDCC*, 2015.
- [480] Ahsan Javed Awan, Vladimir Vlassov, Mats Brorsson, and Eduard Ayguade. Node Architecture Implications for In-Memory Data Analytics on Scale-in Clusters. In *BDCAT*, 2016.
- [481] Geraldo Francisco de Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study. In *SIGMETRICS*, 2021.
- [482] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *PPoPP*, 2010.
- [483] José Joao, M. Aater Suleman, Onur Mutlu, and Yale Patt. Bottleneck Identification and Scheduling in Multithreaded Applications. In *ASPLOS*, 2012.
- [484] José Joao, M. Aater Suleman, Onur Mutlu, and Yale Patt. Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. In *ISCA*, 2013.

- [485] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. Parallel Application Memory Scheduling. In *MICRO*, 2011.
- [486] M. Aater Suleman, Onur Mutlu, José A. Joao, Khubaib, and Yale N. Patt. Data Marshaling for Multi-Core Architectures. In *ISCA*, 2010.
- [487] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. 2008.
- [488] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte*, 2004.
- [489] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.
- [490] Leslie Lamport. A New Solution of Dijkstra’s Concurrent Programming Problem. *Commun. ACM*, 1974.
- [491] Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *TC*, 1979.
- [492] Craig Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 2011.
- [493] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides Benítez, and N. Guil Mata. Performance Modeling of Atomic Additions on GPU Scratchpad Memory. *TPDS*, 2013.
- [494] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.
- [495] Intel. *64 and IA-32 Architectures Software Developer’s Manual*. 2009.
- [496] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore Locks: The Case Is Not Closed Yet. In *USENIX ATC*, 2016.
- [497] Larry Rudolph and Zary Segall. *Dynamic Decentralized Cache Schemes for MIMD Parallel Processors*. 1984.
- [498] THOMASE Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *ICPP*, 1989.
- [499] John M Mellor-Crummey and Michael L Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS*, 1991.
- [500] Michael L Scott. Non-Blocking Timeout in Scalable Queue-based Spin Locks. In *PODC*, 2002.



- [501] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *TOPC*, 2015.
- [502] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *IPDPS*, 1994.
- [503] Travis Craig. Building FIFO and Priority Queuing Spin Locks from Atomic Swap. Technical report, 1993.
- [504] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In *Euro-Par*, 2006.
- [505] Dave Dice, Virendra J Marathe, and Nir Shavit. Flat-Combining NUMA Locks. In *SPAA*, 2011.
- [506] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. *PPoPP*, 2015.
- [507] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *HPDC*, 2015.
- [508] John Mellor-Crummey and Michael Scott. Synchronization without Contention. *ASPLOS*, 1991.
- [509] Mark Heinrich, Vijayaraghavan Soundararajan, John Hennessy, and Anoop Gupta. A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. *TC*, 1999.
- [510] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *ISCA*, 2010.
- [511] John H. Kelm, Matthew R. Johnson, Steven S. Lumetta, and Sanjay J. Patel. WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In *PACT*, 2010.
- [512] Xiongchao Tang, Jidong Zhai, Xuehai Qian, and Wenguang Chen. pLock: A Fast Lock for Architectures with Explicit Inter-core Message Passing. In *ASPLOS*, 2019.
- [513] Debra Hensgen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 1988.
- [514] Dirk Grunwald and Suvas Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In *IPDPS*, 1994.
- [515] David Culler, Jaswinder Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware-Software Approach*. 1999.
- [516] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *ISCA*, 2013.

- [517] Jochem Rutgers, Marco Bekooij, and Gerard Smit. Portable Memory Consistency for Software Managed Distributed Memory in Many-Core SoC. In *IPDPSW*, 2013.
- [518] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*, 2007.
- [519] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A Detailed on Chip Network Model inside a Full-System Simulator. In *ISPASS*, 2009.
- [520] P. T. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. E. Becker, and J. Becker. Energy Model of Networks-on-Chip and a Bus. In *SOCC*, 2005.
- [521] U. Narayan Bhat. *An Introduction to Queueing Theory: Modeling and Analysis in Applications*. Birkhäuser Basel, 2nd edition, 2015.
- [522] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019.
- [523] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, and et al. Alleviating Irregularity in Graph Analytics Acceleration: A Hardware/Software Co-Design Approach. In *MICRO*, 2019.
- [524] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin. In *MICRO*, 2016.
- [525] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavayan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. Profiling DNN Workloads on a Volta-based DGX-1 System. In *IISWC*, 2018.
- [526] NVIDIA. ONTAP AI—NVIDIA DGX-2 POD with NetApp AFF A800. *White Paper*, 2019.
- [527] Xilinx. Virtex UltraScale+ HBM FPGA, 2019.
- [528] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, 2015.
- [529] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. *PPoPP*, 2014.
- [530] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *IISWC*, 2015.
- [531] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *CGO*, 2014.
- [532] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. In *TOMS*, 2011.

- [533] Sahar Torkamani and Volker Lohweg. Survey on Time Series Motif Discovery. *Wiley Interdis. Rev.: Data Mining and Knowledge Discovery*, 2017.
- [534] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. In *ICDM*, 2016.
- [535] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*, 1996.
- [536] William Pugh. Concurrent Maintenance of Skip Lists. Technical report, 1990.
- [537] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *PPoPP*, 2015.
- [538] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 1998.
- [539] ARM. Cortex-A7 Technical Reference Manual. 2009.
- [540] A. Ros and S. Kaxiras. Callback: Efficient Synchronization without Invalidation with a Directory just for Spin-Waiting. In *ISCA*, 2015.
- [541] NVIDIA. NVIDIA Tesla V100 GPU Architecture. *White Paper*, 2017.
- [542] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, and David A. Kranz. The MIT Alewife Machine: Architecture and Performance. In *ISCA*, 1998.
- [543] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *MICRO*, 2019.
- [544] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, et al. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *ISCA*, 2020.
- [545] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Enabling Practical Processing In and Near Memory for Data-Intensive Computing. In *DAC*, 2019.
- [546] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *ASPLOS*, 2020.
- [547] F. Nisa Bostanci, Ataberk Olgun, Lois Orosa, A. Giray Yağlıkçı, Jeremie S. Kim, Hasan Hassan, Oğuz Ergin, and Onur Mutlu. DR-STRaNGE: End-to-End System Design for DRAM-based True Random Number Generators. In *HPCA*, 2022.

- [548] Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *HPCA*, 2019.
- [549] Mohammed Alser, Zülal Bingöl, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Accelerating Genome Analysis: A Primer on an Ongoing Journey. In *IEEE Micro*, 2020.
- [550] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. DRAMA: An Architecture for Accelerated Processing Near Memory. In *IEEE CAL*, 2015.
- [551] Mohammad Alian and Nam Sung Kim. NetDIMM: Low-Latency Near-Memory Network Interface Architecture. In *MICRO*, 2019.
- [552] Vivek Seshadri and Onur Mutlu. In-DRAM Bulk Bitwise Execution Engine. In *CoRR*, 2020.
- [553] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *MICRO*, 2016.
- [554] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. Medal: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm. In *MICRO*, 2019.
- [555] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *HPCA*, 2017.
- [556] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM. In *ICASSP*, 2014.
- [557] Kevin K. Chang, Prashant J. Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K. Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016.
- [558] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.
- [559] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. DrAcc: a DRAM based Accelerator for Accurate CNN Inference. In *DAC*, 2018.
- [560] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *ASPLOS*, 2021.
- [561] Yifat Levy, Jehoshua Bruck, Yuval Cassuto, Eby G. Friedman, Avinoam Kolodny, Eitan Yaakobi, and Shahar Kvatinsky. Logic Operations in Memory Using a Memristive Akers Array. In *Microelectronics booktitle*, 2014.

- [562] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. MAGIC—Memristor-Aided Logic. In *IEEE TTCSII*, 2014.
- [563] Shahar Kvatinsky, Avinoam Kolodny, Uri C. Weiser, and Eby G. Friedman. Memristor-Based IMPLY Logic Design Procedure. In *ICCD*, 2011.
- [564] Pierre-Emmanuel Gaillardon, Luca Amarú, Anne Siemon, Eike Linn, Rainer Waser, Anupam Chattopadhyay, and Giovanni De Micheli. The Programmable Logic-in-Memory (PLiM) computer. In *DATE*, 2016.
- [565] Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, and Said Hamdioui. Fast Boolean Logic Mapped on Memristor Crossbar. In *ICCD*, 2015.
- [566] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating Graph Processing Using ReRAM. In *HPCA*, 2018.
- [567] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Sapan Agarwal, Matthew Marinella, Martin Foltin, John Paul Strachan, Dejan Milojicic, Wen-Mei Hwu, and Kaushik Roy. PANTHER: A Programmable Architecture for Neural Network Training Harnessing Energy-Efficient ReRAM. In *IEEE TC*, 2020.
- [568] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference. In *ASPLOS*, 2019.
- [569] Yue Xi, Bin Gao, Jianshi Tang, An Chen, Meng-Fan Chang, Xiaobo Sharon Hu, Jan Van Der Spiegel, He Qian, and Huaqiang Wu. In-memory Learning with Analog Resistive Switching Memory: A Review and Perspective. In *Proceedings of the IEEE*, 2021.
- [570] Le Zheng, Sangho Shin, Scott Lloyd, Maya Gokhale, Kyungmin Kim, and Sung-Mo Kang. RRAM-based TCAMs for Pattern Search. In *ISCAS*, 2016.
- [571] Said Hamdioui, Shahar Kvatinsky, Gert Cauwenberghs, Lei Xie, Nimrod Wald, Siddharth Joshi, Hesham Mostafa Elsayed, Henk Corporaal, and Koen Bertels. Memristor for Computing: Myth or Reality? In *DATE*, 2017.
- [572] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Trade-off in Modern Commodity DRAM Devices. In *HPCA*, 2018.
- [573] Lois Orosa, Yaohua Wang, Mohammad Sadrosadati, Jeremie S. Kim, Minesh Patel, Ivan Puddu, Haocong Luo, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Nika Mansouri-Ghiasi, Saugata Ghose, and Onur Mutlu. CODIC: A Low-Cost Substrate for Enabling Custom in-DRAM Functionalities and Optimizations. In *ISCA*, 2021.

- [574] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast. In *ISCA*, 2021.
- [575] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oğuz Ergin, and Onur Mutlu. PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM. In *CoRR*, 2021.
- [576] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. In *ISCA*, 2021.
- [577] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, and Caiwen Ding. FORMS: Fine-grained Polarized ReRAM-based In-situ Computation for Mixed-signal DNN Accelerator. In *ISCA*, 2021.
- [578] Kamil Khan, Sudeep Pasricha, and Ryan Gary Kim. A Survey of Resource Management for Processing-In-Memory and Near-Memory Processing Architectures. In *booktitle of Low Power Electronics and Applications*, 2020.
- [579] Min-Jae Kim, Jeong-Geun Kim, Su-Kyung Yoon, and Shin-Dug Kim. Functionality-Based Processing-in-Memory Accelerator for Deep Convolutional Neural Networks. In *IEEE Access*, 2021.
- [580] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. Active-Routing: Compute on the Way for Near-Data Processing. In *HPCA*, 2019.
- [581] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*, 2014.
- [582] Åke Björck. Numerical Methods for Least Squares Problems. In *SIAM*, 1996.
- [583] Udo W. Pooch and Al Nieder. A Survey of Indexing Techniques for Sparse Matrices. In *ACM Comput. Surv.*, 1973.
- [584] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *GH*, 2007.
- [585] Eun-Jin Im and Katherine A. Yelick. Optimizing Sparse Matrix Vector Multiplication on SMP. In *PPSC*, 1999.
- [586] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW*, 1998.

- [587] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *IPDPS*, 2017.
- [588] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse Convolutional Neural Networks. In *CVPR*, 2015.
- [589] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.
- [590] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning Both Weights and Connections for Efficient Neural Networks. In *NIPS*, 2015.
- [591] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP*, 2018.
- [592] Greg Linden, Brent Smith, and Jeremy York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. In *IC*, 2003.
- [593] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Malleovich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep Learning Recommendation Model for Personalization and Recommendation Systems. In *CoRR*, 2019.
- [594] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The Architectural Implications of Facebook’s DNN-based Personalized Recommendation. In *CoRR*, 2019.
- [595] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *ISCA*, 2020.
- [596] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *SIGGRAPH*, 2003.
- [597] R. D. Falgout. An Introduction to Algebraic Multigrid. In *Computing in Science Engineering*, 2006.
- [598] Robert D Falgout and Ulrike Meier Yang. hypre: A Library of High Performance Preconditioners. In *ICCS*, 2002.
- [599] Pascal Hénon, Pierre Ramet, and Jean Roman. PASTIX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. In *PMAA*, 2002.

- [600] Daniel Langr and Pavel Tvrdík. Evaluation Criteria for Sparse Matrix Storage Formats. In *TPDS*, 2016.
- [601] Weifeng Liu and Brian Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS*, 2015.
- [602] Wangdong Yang, Kenli Li, Yan Liu, Lin Shi, and Lanjun Wan. Optimization of Quasi-Diagonal Matrix-Vector Multiplication on GPU. In *Int. J. High Perform. Comput. Appl.*, 2014.
- [603] Mehmet Belgin, Godmar Back, and Calvin J. Ribbens. Pattern-Based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. In *ICS*, 2009.
- [604] SciPy. List-of-list Sparse Matrix. In *SciPy*, 2021.
- [605] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2D User’s Guide. 1989.
- [606] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC*, 2018.
- [607] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *HiPEAC*, 2010.
- [608] Youcef Saad. Krylov Subspace Methods on Supercomputers. In *SIAM J. Sci. Stat. Comput.*, 1989.
- [609] Aydın Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *SPAA*, 2009.
- [610] Michele Martone. Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-Vector Multiplication with the Recursive Sparse Blocks Format. In *Parallel Computing*, 2014.
- [611] Michele Martone, Salvatore Filippone, Marcin Paprzycki, and Salvatore Tucci. On BLAS Operations with Recursively Stored Sparse Matrices. In *SYNASC*, 2010.
- [612] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop. Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. In *IPDPSW*, 2012.
- [613] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. BestSF: A Sparse Meta-Format for Optimizing SpMV on GPU. In *TACO*, 2018.
- [614] JEDEC. JESD79-4 DDR4 SDRAM standard. In *JEDEC*, 2012.
- [615] UPMEM. UPMEM User Manual. Version 2021.3. In *UPMEM*, 2021.



- [616] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. In *IEEE Comput. Sci. Eng.*, 1998.
- [617] Bahar Asgari, Ramyad Hadidi, Joshua Dierberger, Charlotte Steinichen, and Hyesoon Kim. Copernicus: Characterizing the Performance Implications of Compression Formats Used in Sparse Workloads. In *CoRR*, 2020.
- [618] Intel. Intel Xeon Silver 4110 Processor. In <https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache-2-10-ghz.html>, 2017.
- [619] NVIDIA. NVIDIA Tesla V100 GPU Architecture. In <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>, 2017.
- [620] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.
- [621] Vivek Seshadri and Onur Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers*, 2017.
- [622] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R. Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *HPCA*, 2014.
- [623] peakperf. peakperf. In <https://github.com/Dr-Noob/peakperf.git>, 2021.
- [624] stream. stream. In <https://github.com/jeffhammond/STREAM.git>, 2021.
- [625] Weifeng Liu and Brian Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS*, 2015.
- [626] CSR5. CSR5 Cuda. In <https://github.com/weifengliu-ssslab/BenchmarkSpMVusingCSR5>, 2015.
- [627] cuSparse. cuSparse. In <https://docs.nvidia.com/cuda/cusparses/index.html>, 2021.
- [628] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in Action: Experiences in Using RAPL for Power Measurements. In *TOMPECS*, 2018.
- [629] NVIDIA. NVIDIA System Management Interface Program. In <http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf>, 2016.

- [630] Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. Near-Memory Data Transformation for Efficient Sparse Matrix Multi-Vector Multiplication. In *SC*, 2019.
- [631] Wangdong Yang, Kenli Li, and Keqin Li. A Hybrid Computing Method of SpMV on CPU-GPU Heterogeneous Computing Systems. In *JPDC*, 2017.
- [632] Sivaramakrishna Bharadwaj Indarapu, Manoj Maramreddy, and Kishore Kothapalli. Architecture- and Workload- Aware Heterogeneous Algorithms for Sparse Matrix Vector Multiplication. In *COMPUTE*, 2014.
- [633] Wangdong Yang, Kenli Li, Zeyao Mo, and Keqin Li. Performance Optimization Using Partitioned SpMV on GPUs and Multicore CPUs. In *IEEE Transactions on Computers*, 2015.
- [634] Akrem Benatia, Weixing Ji, and Yizhuo Wang. Sparse Matrix Partitioning for Optimizing SpMV on CPU-GPU Heterogeneous Platforms. In *IJHPCA*, 2019.
- [635] Brice Boyer, Jean-Guillaume Dumas, and Pascal Giorgi. Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures. In *PASCO*, 2010.
- [636] Juan C. Pichel and Francisco F. Rivera. Sparse Matrix-Vector Multiplication on the Single-Chip Cloud Computer Many-Core Processor. In *J. Parallel Distrib. Comput.*, 2013.
- [637] Sivaramakrishna Bharadwaj Indarapu, Manoj Maramreddy, and Kishore Kothapalli. Architecture- and Workload-Aware Heterogeneous Algorithms for Sparse Matrix Vector Multiplication. In *ICPADS*, 2013.
- [638] Petros Anastasiadis, Nikela Papadopoulou, Georgios Goumas, and Nectarios Koziris. Co-CoPeLia: Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs. In *ISPASS*, 2021.
- [639] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. Tile-SpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *IPDPS*, 2021.
- [640] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. Overhead-Conscious Format Selection for SpMV-Based Applications. In *IPDPS*, 2018.
- [641] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *ICPP*, 2016.
- [642] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP*, 2018.
- [643] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In *PLDI*, 2013.
- [644] Marco Maggioni and Tanya Berger-Wolf. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In *ICPP*, 2013.

- [645] Guangming Tan, Junhong Liu, and Jiajia Li. Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture. In *ACM Trans. Math. Softw.*, 2018.
- [646] Kenli Li, Wangdong Yang, and Keqin Li. Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling. In *IEEE TPDS*, 2015.
- [647] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of Research and Development*, 2002.
- [648] Moinuddin Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. 2011.
- [649] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS*, 2013.
- [650] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *HPCA*, 2018.
- [651] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*, 2018.
- [652] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP*, 2021.
- [653] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *ASPLOS*, 2022.
- [654] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.
- [655] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *HPDC*, 2019.
- [656] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *HPCA*, 2019.
- [657] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *ASPLOS*, 2017.
- [658] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *HPCA*, 2015.

- [659] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *MICRO*, 2018.
- [660] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems. In *IPDPS*, 2021.
- [661] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*, 2019.
- [662] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *SoCC*, 2017.
- [663] Qizhen Zhang, Yifan Cai, Sebastian G. Angel, Vincent Liu, Ang Chen, and B. T. Loo. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR*, 2020.
- [664] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-Level Implications of Disaggregated Memory. In *HPCA*, 2012.
- [665] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the Application. In *USENIX HotCloud*, 2020.
- [666] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.
- [667] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: A Simple Abstraction for Remote Memory. In *ATC*, 2018.
- [668] Zhiwen Chen, Xin He, Jianhua Sun, Hao Chen, and Ligang He. Concurrent Hash Tables on Multicore Machines: Comparison, Evaluation and Implications. *Future Generation Computer Systems*, 2018.
- [669] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, 2002.
- [670] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *EuroSys*, 2022.
- [671] A.R. Alameldeen and D.A. Wood. Adaptive Cache Compression for High-Performance Processors. In *ISCA*, 2004.

- 
- [672] Alaa R. Alameldeen and David A. Wood. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *HPCA*, 2007.
- [673] Irina Chihaiia Tuduce and Thomas Gross. Adaptive Main Memory Compression. In *USENIX ATC*, 2005.
- [674] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. HyComp: A Hybrid Cache Compression Method for Selection of Data-Type-Specific Compression Methods. In *MICRO*, 2015.
- [675] Seikwon Kim, Seonyoung Lee, Taehoon Kim, and Jaehyuk Huh. Transparent Dual Memory Compression Architecture. In *PACT*, 2017.
- [676] SAFARI Research Group. NATSA. In <https://github.com/CMU-SAFARI/NATSA>, 2020.
- [677] SAFARI Research Group. SMASH. In <https://github.com/CMU-SAFARI/SMASH>, 2019.