A Workload-aware Resource Management and Scheduling System for Big Data Analysis

Luna Xu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Kirk W. Cameron
Dongyoon Lee
Min Li
Calvin J. Ribbens

Dec. 11, 2018
Blacksburg, Virginia

Keywords: Resource Management, Scheduling, Big Data, Machine Learning, Spark, Heterogeneity, Workload Characteristic, Efficiency

A Workload-aware Resource Management and Scheduling System for Big Data Analysis

Luna Xu

(ABSTRACT)

The big data era has driven the needs for data analysis in every aspect of our daily lives. With the rapid growth of data size and complexity of data analysis models, modern big data analytic applications face the challenge to provide timely results often with limited resources. Such demand drives the growth of new hardware resources including GPUs, FPGAs, SSDs and NVMs. It is challenging to manage the resources available in a cost restricted environment to best serve the applications with different characteristics. Extant approaches are agnostic to such heterogeneity in both underlying resources and workloads and require user knowledge assisted manual configuration for best performance. In this dissertation, we design, and implement a series of novel techniques, algorithms, and frameworks, to realize workload-aware resource management and scheduling. We demon strate our techniques for efficient resource mangement across memory resource for in-memory data analytic platforms, processing resources for compute-intensive machine learning applications, and finally we design and develop a workload and heterogeneity-aware scheduler for general big data platforms.

The dissertation demonstrates that designing an effective resource manager requires efforts from both application and system side. The presented approach makes and joins the efforts on both sides to provide a holistic heterogenity-aware resource manage and scheduling system. We are able to avoid task failure due to resource unavailability by workload-aware resource man agement, and improve the performance of data processing frameworks by carefully scheduling tasks according to the task characteristics and utilization and availability of the resources.

In the first part of this dissertation (Chapter 3), we focus on the problem of configration-based memory management for in-memory data analytic platforms and how we dynamically manage the memory resource based on workload runtime information. We further extend optimize memory usage by optimizing the data partition size and scheme (Chapter 4). In the second part of this dissertation (Chapter 5), we design and develop a dynamic hardware acceleration framework that utilizes both CPU and GPU processors to acceleterate both dense and sparse matrix operations for machine learning applications. In the third part of this dissertation (Chapter 6), we propose a framework that are able to host different types of applications in the same set of resources with workload-aware resource allocation. In the fourth part of this dissertation (Chapter 7), we combine all the previous resource findings and techniques, and present a heterogeneity-aware Spark scheduler that is aware of both task characteristics and also heterogeneity of underlying resources. We also prove that simple heuritics and algorithm are effective and improve performance of big data applications.

A Workload-aware Resource Management and Scheduling System for Big Data Analysis

Luna Xu

(GENERAL AUDIENCE ABSTRACT)

Clusters of multiple computers connected through internet are often deployed in industry for larger scale data processing or computation that cannot be handled by standalone computers. In such a cluster, resources such as CPU, memory, disks are integrated to work together. It is important to manage a pool of such resources in a cluster to efficiently work together to provide better performance for workloads running on top. This role is taken by a software component in the middle layer called resource manager. Resource manager coordinates the resources in the computers and schedule tasks to them for computation. This dissertation reveals that current resource managers often partition resources statically hence cannot capture the dynamic resource needs of workloads as well as the heterogeneous configurations of the underlying resources. For example, some computers in a clsuter might be older than the others with slower CPU, less memory, etc. Workloads can show different resource needs. Watching YouTube require a lot of network resource while playing games demands powerful GPUs. To this end, the disseration proposes novel approaches to manage resources that are able to capture the heterogeneity of resources and dynamic workload needs, based on which, it can achieve efficient resource management, and schedule the right task to the right resource.

Dedicated to my parents for their generous support and selfless love.

Acknowledgments

This dissertation would not be possible without the support and help of many professionals, fellow students, and other individuals. I would like to express my deep gratitude to everyone who has advised, funded, helped, and enlightened me in every possible way.

First, I would like to thank my research and life advisor, Professor Ali R. Butt, without whom I could not have started and finished this Ph.D. journey. I was not confident that I could write any paper. I had many difficulties, in both research and life, when I first arrived at the lab. It was even difficult for me to read and understand papers. I could not have possibly continued if it were not for Ali's tolerance and encouragement. Ali's support was not limited to research resources, as he also truly believed in me. I have had ups and downs in my Ph.D. life. I am deeply grateful that Ali did not give up on me during my down phases. Another huge benefit I received from my Ph.D. study is the countless chances to attend conferences and interact with the research community. This could only be achieved because Ali always encourages and supports us to attend all opportunities. Through these conferences, I was able to communicate with other students and researchers to gain precious opportunities for collaborations and internships. Anyone can learn a lot from Ali. Apart from paper writing skills and research methods, more importantly, I learned a new way of looking at both research and the world, as well as how to interact with people. Professor Ali is not only my research advisor, but also my mentor for life, as he indeed changed my life.

Second, I want to express my great thanks to Dr. Min Li, who is also an alumnus of DSSL. If Ali is my high-level research advisor, Min is my actual research mentor and advisor. Min began working closely with me at the beginning of my Ph.D. and taught me everything I needed to know about research and writing papers. The topic of this dissertation was inspired by Min and a large portion of it was completed together with her. Not only did she help to form the problems and discuss ideas, which laid the foundation of this dissertation, but her research methodology and paper writing heavily influenced my research. She is the other person without whom I could not finish this dissertation. I would like to thank her sincerely.

I also want to thank Dr. Seung-Hwan Lim, who acted as my mentor during my two internships in Oak Ridge National Lab. Seung-Hwan gave me opportunities to work in the

HPC environment. He also supported my Ph.D. study through internships and project funding. Chapter 5 would not be possible without his help and support.

Next, I would like to thank all my Ph.D. committee members: Professors Kirk Cameron, Cal Ribbens, and Dongyoon Lee, for their kind comments and insightful feedback on my dissertation. Special thanks to Dr. Cameron and Lee, who provided many useful comments and questions for me to improve my dissertation and guide my future research.

I have been fortunate to work with several distinguished researchers and students through internships (IBM T.J Watson in Summer 2015, two in Oak Ridge National Lab in the Summers of 2016 and 2017, and Alibaba Group in Summer 2018). I would like to thank my mentors Min Li, Seung-Hwan Lim, Li Zhang, Yandong Wang, Zane Zhenhua Hu, and Ramakrishnan Kannan for sharing these amazing projects with me.

I also thank all my lab mates in DSSL: Krish K.R., Yue Cheng, Hyogi Sim, Ali Anwar, Xiaozhong Pan, Bharti Wadhwa, Arnab K. Paul, Nannan Zhao, and Jingoo Han. Special thanks to Yue, who helped me a lot in preparing my proposal and interview talks. Ali helped me a lot in improving this dissertation and defense. Thanks to Hyogi for all the encouragement when it was difficult to continue, as well as the valuable technical help and research discussions. I have not been socially active with my lab mates; however, I still thank everyone who helped me when I asked for help. I am grateful that you tolerated me when it was not easy. I appreciate your respect and kindness while being part of this special journey.

I thank all staff in the Computer Science Department at Virginia Tech, especially Sharon Kinder-Potter, a reliable coordinator and friend who I always go to if I have any problems. She always gives me the best solutions and suggestions. Teresa M. Hall made all conference travel possible. It is a pleasure to work with her when she eases all the hustle and procedures. I also thank all other staff working in CRC, Megan, Matt, Melanie, and the Tech Staff, who provide us with an easy and cozy working environment.

Funding Acknowledgment My research was supported by National Science Foundation (CNS-1615411, CNS-1565314, CCF-0746832, CNS-1016408, CNS-1016793, CNS-1405697 and CNS-1422788), CS at Virginia Tech, DOE (DE-AC05-00OR22725) from Oak Ridge National Lab. Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my adviser Ali R. Butt, this dissertation is contributed by many respected collaborators, especially:

- Min Li has contributed to most of the work of this dissertation with multiple publications [203, 204, 207]. She made huge contribution in both design and evaluation of work included in Chapter 6 and Capter 3.

- Li Zhang, Yandong Wang, Zane Hu also contributed to the algorithms for dynamic memory management in Chapter 3.
- Arnab K. Paul, Wenjie Zhuang, and M. Mustafa Rafique contributed to the work of optimizing partitions for memory utilization [159] described in Chapter 4. Especially, Arnab contributed a significant amount to the implementation and evaluation for this project.
- Seung-Hwan Lim and Ramakrishnan Kannan contributed to the work of utilizing hardware accelerators for big data analysis [205, 207, 208] included in Chapther 5 and Chapter 7. Oak Ridge National Lab provided the HPC platfrom to conduct all the experiments. Seung-Hwan also contributed a lot in discussion of the ideas.

Contents

List of Figures xiii

List of Tables xvi

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 Memory Management for In-memory Data Analytics . . . 2

1.1.2 Exploiting Specialized Accelerators for Big Data Analytics . . . 3

1.1.3 Workload-aware Heterogeneous Resource Scheduling . . . 3

1.2 Research Contributions . . . 4

1.3 Dissertation Organization . . . 6

2 Background 7

2.1 In-Memory Data Analytic Frameworks . . . 7

2.2 Hardware Acceleration in Big Data Analytics . . . 9

2.3 Heterogeneity-aware Resource Management . . . 10

3 Dynamic Memory Management for In-memory Data Analytic Platforms 13

3.1 Introduction . . . 13

3.2 Background and Motivation . . . 15

3.2.1 Spark Memory Management . . . 15

3.2.2 Empirical Study . . . 16

3.3 System Design . . . 19

3.3.1 Architecture Overview . . . 19
3.3.2 Dynamically Tuning RDD Cache and JVM Heap Size . . . 21
3.3.3 RDD Eviction Policy . . . 24
3.3.4 Prefetch and Prefetch Window . . . 25
3.3.5 Discussion . . . 26
3.4 Evaluation . . . 26
3.4.1 Overall Performance of MemTune . . . 27
3.4.2 Impact of Garbage Collection . . . 27
3.4.3 Impact on Cache Hit Ratio . . . 28
3.4.4 Dynamic RDD Cache Size Tuning . . . 29
3.4.5 DAG-Aware RDD Prefetching . . . 30
3.5 Chapter Summary . . . 30

4 Optimizing Data Partitioning for In-Memory Data Analytics Frameworks 31
4.1 Introduction . . . 31
4.2 Background and Motivation . . . 33
4.2.1 Spark Data Partitioning . . . 34
4.2.2 Workload Study . . . 35
4.3 System Design . . . 37
4.3.1 Enable Auto-Partitioning . . . 38
4.3.2 Determine Stage-Level Partition Scheme . . . 39
4.3.3 Globally-Optimized Partition Scheme . . . 42
4.4 Evaluation . . . 44
4.4.1 Overall Performance of Chopper . . . 44
4.4.2 Timing Breakdown of Execution Stages . . . 45
4.4.3 Impact on Shuffle Stages . . . 46
4.4.4 Impact on System Utilization . . . 47
4.5 Chapter Summary . . . 48

5 Scaling Up Data-Parallel Analytics Platforms **49**

5.1 Introduction . . . 49

5.2 Background . . . 51

5.3 Design . . . 52

5.3.1 System Architecture . . . 52

5.3.2 Hardware Acceleration for Dense and Spare Matrices . . . 53

5.3.3 Choice of Processing Variants . . . 54

5.3.4 Improving System Utilization . . . 55

5.4 Evaluation . . . 58

5.4.1 Overall Performance . . . 59

5.4.2 Performance Breakdown . . . 60

5.4.3 System Utilization . . . 61

5.4.4 Impact of Block Size . . . 63

5.4.5 Scalability of ARION . . . 65

5.5 Chapter Summary . . . 65

6 Resource Mangement for Heterogeneous Applications **67**

6.1 Introduction . . . 67

6.2 Enabling Technologies . . . 69

6.2.1 The MPI Framework . . . 69

6.2.2 YARN . . . 70

6.3 System Design . . . 70

6.3.1 Design Objectives . . . 70

6.3.2 Architecture Overview . . . 71

6.3.3 User-Specified Container Allocation Modes . . . 72

6.3.4 Container and MPI Process Management . . . 75

6.3.5 Discussion . . . 78

6.4 Implementation . . . 79

6.5 Evaluation . . . 80

6.5.1 Experimental Setup . . . 80
6.5.2 Performance Comparison of Gerbil and Native MPI . . . 81
6.5.3 Performance Comparison of Gerbil and MapReduce . . . 82
6.5.4 Performance Impact of Gerbil Container Allocations . . . 82
6.5.5 Impact of Oversubscription . . . 83
6.5.6 Impact on Application Launch Time . . . 85
6.6 Chapter Summary . . . 85

7 A Heterogeneity-Aware Task Scheduler for Spark **87**
7.1 Introduction . . . 87
7.2 Background and motivation . . . 89
7.2.1 Spark scheduling . . . 90
7.2.2 Motivational study . . . 90
7.3 Design . . . 92
7.3.1 System architecture . . . 92
7.3.2 Real-time resource and task monitoring . . . 94
7.3.3 Task scheduling . . . 96
7.4 Evaluation . . . 100
7.4.1 Hardware capabilities . . . 101
7.4.2 Overall performance . . . 102
7.4.3 Impact on locality . . . 103
7.4.4 Performance breakdown . . . 104
7.4.5 Impact on system utilization . . . 106
7.5 Chapter Summary . . . 107

8 Conclusion and Future Work **108**
8.1 Summary . . . 109
8.2 Future Work . . . 109
8.2.1 Scalable Resource Monitor . . . 110

8.2.2 Accurate Workload Profiler 110
8.2.3 Performance Prediction 111

Bibliography **112**

List of Figures

3.1 Typical memory partitioning used in Spark. 15

3.2 Total execution time and garbage collection time of *Logistic Regression* under `MEMORY_ONLY`. 16

3.3 Computation time and garbage collection time of *Logistic Regression* under `MEMORY_AND_DISK`. 16

3.4 Memory usage of *TeraSort*. 16

3.5 RDD sizes in memory in different stages of *Shortest Path* under default configurations. 19

3.6 Ideal RDD sizes in memory based on stage RDD dependencies of *Shortest Path*. . 19

3.7 System architecture of MemTune. 21

3.8 Generation of tasks based on RDD dependency after an RDD action is submitted to DAGSchduler. 24

3.9 Execution time of studied workloads under the default Spark, MemTune, MemTune with prefetch only, and MemTune with tuning only. LogR: *Logistic Regression*, LinR: *Linear Regression*, PR: *Page Rank*, CC: *Connected Components*, SP: *Shortest Path*. 26

3.10 Gabbage collection ratio of studied workloads under different scenarios. 28

3.11 RDD memory cache hit ratio of studied workloads under different scenarios. . . . 28

3.12 Dynamic change in RDD cache size during execution. 29

3.13 RDD cache sizes in memory with the default configuration under different stages with MemTune. 29

4.1 Overview of Spark DAGScheduler. 33

4.2 Execution time per stage under different number of partitions. 35

4.3 Execution time of stage 0 under different partition numbers. 36

4.4 Shuffle data per stage under different partition numbers. 36
4.5 System architecture of CHOPPER. 37
4.6 An example generated Spark workload configuration in CHOPPER. 38
4.7 Execution time of Spark and CHOPPER. 45
4.8 Execution time per stage breakdown of KMeans. 45
4.9 Shuffle data per stage for SQL. 46
4.10 Execution time per stage breakdown of SQL. 46
4.11 CPU utilization. 47
4.12 Memory utilization. 47
4.13 Total transmitted and received packets per second. 47
4.14 Transactions per second. 47

5.1 System architecture of ARION. 52
5.2 Execution time of matrix multiplication using different implementations. X-axis represents b in a block matrix $b \times b$. Both axes are log-scaled. 54
5.3 Throughput of NVBLAS. X-axis represents b in a block matrix $b \times b$. 54
5.4 End-to-end execution time with default Spark MLlib, ARION, and the naive adoption of other processing variants. (Note log-scaled Y-axis in (a).) 59
5.5 Performance breakdown of studied matrices under ARION. 62
5.6 Speed-up of aggregated compute time with ARION, and other processing variants with respect to the Scala implementation in MLlib. 62
5.7 CPU utilization of $16K \times 16K$ dense matrix using Spark MLlib, ARION, and OpenBLAS. 64
5.8 Aggregated GPU utilization of $16K \times 16K$ dense matrix using ARION, **GPU Impl**, and NVBLAS. 64
5.9 Aggregated utilization of $64K \times 64K$ sparse matrix using ARION. 64
5.10 Breakdown of end-to-end execution time with different block sizes. 64
5.11 Average computation time of *BlockMatrix* multiplication using different block sizes with **GPU Impl**, OpenBLAS, and NVBLAS. 64
5.12 Performance of default Spark MLlib and ARION in a 128-node scale. Y-axis is log-scaled. . 64

6.1 Steps of launching an application in YARN. 70

6.2 GERBIL architecture for running MPI on YARN. 71
6.3 GERBIL components and their interactions. 80
6.4 Job execution time for small MPI applications. 81
6.5 Execution time breakdown for small MPI applications. 81
6.6 Job execution time for large MPI applications. 81
6.7 Execution time breakdown for large MPI applications. 81
6.8 Performance of GERBIL MPI vs. MapReduce application versions. 81
6.9 Speedup of GERBIL MPI over MapReduce Pi-MC versions with increasing number of iterations. 81
6.10 Total job execution time under studied allocation algorithms. 84
6.11 Slowdown due to oversubscription with increasing number of processes, normalized to `alloc_relax_all`. 84
6.12 Slowdown due to oversubscription with increasing number of iterations, normalized to `alloc_relax_all`. 84
6.13 GERBIL vs. MapReduce execution time under oversubsciption. 85
6.14 Container allocation slowdown of YARN RM normalized to `alloc_relax_all`. . . 85

7.1 Spark application architecture. 88
7.2 System resource utilization under $4K \times 4K$ matrix multiplication. X-axis represents relative timestamp (trimmed for Disk performance). 88
7.3 Task distribution and execution breakdown of PageRank. 89
7.4 RUPAM system architecture. 92
7.5 Overall performance of studied workloads under default Spark and RUPAM. . 103
7.6 Speedup of LR under RUPAM wrt. default Spark with increasing number of workload iterations. 104
7.7 Performance breakdown of selected workloads under default Spark and RUPAM. Note that y-axis is log-scaled. 106
7.8 Average system utilization of the nodes with selected workloads under default Spark and RUPAM. 106
7.9 Standard deviation of system utilization among the nodes with selected workloads under the default Spark and RUPAM. 107

List of Tables

3.1 Maximum input size under Spark with default configurations. 18
3.2 RDD dependencies for different stages of *Shortest Path*. '×' and '·' denote whether a stage is dependent on an RDD or not, respectively. 19
3.3 Key APIs provided by MemTune. 20
3.4 Cases of memory contention and corresponding actions taken by MemTune. Yes (Y) and No (N) under the Shuffle, Task and RDD columns denote whether there is memory contention detected for that usage. 22

4.1 Workloads and input data sizes. 44

4.2 Execution time for stage 0 in KMeans. 45

4.3 Repartition of stages using Chopper. 46

5.1 Summary of all processing variants supported in Arion. Methods on "GPU (direct)" platform can co-execute with methods running on "CPU" platform. 55

5.2 System specification. 58

5.3 Studied matrix sizes, densities, and raw file size. 58

6.1 Command-line arguments supported by Gerbil-Client. 72
6.2 Container request modes in Gerbil. For each mode, '×' and '·' denote whether a factor is relaxed or not, respectively. Here, location, distribution, and node count means the location (i.e., nodes on which processes are launched), the distribution (number) of processes per node, and the total number of nodes used to service a request, respectively. 75
6.3 Container allocation distribution with backgroud jobs BG1, BG2 and Pi-MC MPI job. 83

7.1 The node metrics that **RM** monitors on each node. 93

7.2 The task metrics that **TM** monitors on each node. 93

7.3 Specifications of Hydra cluster nodes. 100

7.4 Studied workloads and input sizes. 101
7.5 Hardware characteristics benchmarks. 102
7.6 Number of tasks with different locality level under default Spark and RUPAM. . . 105

Chapter 1

Introduction

Big data analysis has become an essential technology for industry and science. Transcending the previous big data boom era, now more complex data analysis with machine learning and deep learning techniques are required to draw insights from data in many fields, including business, finance, politics, and science. This demand for data and large-scale analytic tasks has motivated the development of both software and hardware. MapReduce [17] based distributed data processing frameworks are widely deployed in industrial companies like Google, Amazon, Microsoft, etc. More frameworks are being developed since Hadoop [31], including Spark [218] and Dryad [108]. Apart from the general MapReduce frameworks, the distributed learning framework community has recently developed large-scale deep learning frameworks and tools such as Tensorflow [33], PyTorch [32], Caffe2 [110], and Theano [38], with more to come. Software has made data analysis easier and more convenient for users; however, it also has introduced complexity and diversity to big data processing tasks. The development of applications and the scale of data drive the development of new hardware for better and faster execution. For example, there are new processing resources with specialized computing power such as GPUs and ASICs, and improved energy-efficiency such as FPGAs, as well as storage resources such as SSDs and NVMs. It is challenging to manage the resources available in a cost-restricted environment. As a result, heterogeneous setups have been adopted in production to support different tasks. To efficiently support as many applications as possible, resource management plays an important role in extracting the resources' capabilities and orchestrating them to serve the applications.

1.1 Motivation

Typical big data frameworks are deployed either in standalone distributed mode or on top of a cluster resource manager such as YARN [187] and Mesos [103]. In standalone mode, the framework adopts a master-slave scheme in which the master collects resource information

from the slaves running on each node and schedules tasks. In such a setting, the resources are usually partitioned into fixed amounts of units (i.e., size of memory and CPU cores) for the tasks to run inside. Neither the master nor the slaves are aware of the heterogeneity of the underlying resources and the master schedules tasks based on data locality. This can lead to the problem of task failure due to insufficient resources (e.g., *OutOfMemory*) or inefficient scheduling decisions (e.g., a GPU task is forced to run with the CPU in a CPU-only node).

Current resource managers like YARN and Mesos have limited support for heterogeneous hardware resources. They either virtualize the resources into homogeneous units or require manual labeling for each resource. Furthermore, they entail a two-level mechanism in which they are agnostic about application characteristics and require the application to request the right resource to be allocated. This approach burdens users on both sides; often, it is impossible to predict resource usage beforehand. Moreover, without an overview of the whole cluster, the scheduler might lose the opportunity for dynamic scheduling and resource overlapping. This causes inefficient utilization of resources and a lack of flexibility to reschedule tasks during suboptimal task allocation.

To address the above issues, this dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks to realize workload-aware resource management and scheduling. This dissertation demonstrates our techniques for efficient resource management across memory resources for in-memory data analytic platforms and processing resources for compute-intensive machine learning applications. We designed and developed a workload- and heterogeneity-aware scheduler for general big data platforms. The overarching goal of this dissertation is to improve the efficiency and flexibility of modern storage applications using resource and functional partitioning.

In the next sections, we briefly describe the research problems, proposed research methodologies, and evaluation results of the work included in this dissertation.

1.1.1 Memory Management for In-memory Data Analytics

MapReduce [17] has been the de-facto distributed processing model for big data analytics due to its ease-of-use and ability to scale. Key to the performance of MapReduce is its ability to cache intermediate data, especially for many iterative applications [218]. In-memory data analytic platforms such as Spark [218] and M3R [174] utilizes memory resources for data caching and thus outperform Hadoop [31] by more than 10 times by reducing I/O. However, memory is a limited and costly resource in terms of capacity with high demand. Moreover, memory plays different roles, such as data caching, shuffle page caching, and task execution consumption. Memory management in big data analytic platforms affects the utilization of all these roles, thus impacting application performance in complicated ways. Hence, it is crucial to optimally manage memory resources for different uses to achieve high performance.

In this work, we discovered that it is impractical to find an optimal solution for all workloads.

Therefore, we adopted a dynamic approach based on heuristics 3 and machine-learning-enabled knowledge 4 to provide adaptive memory management for each workload.

1.1.2 Exploiting Specialized Accelerators for Big Data Analytics

With the prevalence of machine learning and deep learning applications, more data centers are adopting hardware accelerators such as GPUs, ASICs, FPGAs, and specialized CPU instructions [104] for matrix operations to achieve high performance [123, 144, 145]. However, it is non-trivial to exploit accelerators available in a cluster for scale-out data processing frameworks due to the fundamental design choices made by each. Scale-out data processing frameworks such as Spark often focus on scalability, fault tolerance, and workload balancing, but do not typically factor in techniques to support individual nodes' hardware accelerators. However, scale-up solutions focus on parallelism at fine granularity, such as SIMD and SIMT models, and each scale-up solution typically targets specialized operations for the specific target hardware accelerator. Thus, it is challenging to integrate different hardware acceleration techniques into general propose big data processing frameworks through a single run-time solution.

Hence, this work provides a run-time solution for utilizing off-chip accelerators in big data platforms, as well as an online algorithm to determine the best processors for a task.

1.1.3 Workload-aware Heterogeneous Resource Scheduling

Big data processing systems such as Spark are employed in an increasing number of diverse applications, such as machine learning, graph computing, streaming, and scientific computing, each of which has dynamic and different resource needs. Furthermore, some jobs comprise rich multifaceted workflows with both compute-intensive and data-intensive tasks. Such workflows might involve not only one programming paradigm, such as MapReduce, but also other programming models, such as MPI [15], for compute-intensive jobs with complicated communication patterns. It is important to orchestrate underlying resources for different applications' requirements to achieve better resource utilization, workload balance, and performance for applications. However, it is challenging to make an optimal scheduling decision at run-time, especially when applications with heterogeneous resource demands run in a heterogeneous environment in which each node has different hardware capabilities, which has been proven to be NP hard [98].

This part of the dissertation tackles the above problems by presenting Gerbil, which can schedule different workloads such as MPI and MapReduce in the same set of resources. Moreover, we took a step forward into fine-grained task-level scheduling with Rupam, a heterogeneity-aware task scheduling system for big data platforms that considers both task-level resource characteristics and underlying hardware characteristics, and preserves data

locality.

1.2 Research Contributions

From the above three aspects, *we demonstrate that we can improve the performance of big data analytics by carefully managing available resources and scheduling tasks according to the workload characteristics and resource availability and utilization.*

Overall, this dissertation presents innovative systemic and algorithmic approaches to tackle resource management problems in existing big data analytic systems.

We discovered contention on multi-dimensional resources at framework run-time and mitigated the impact of resource contention and task skewness with dynamic resource provisioning and task scheduling. We explored heuristics for scheduling heterogeneous tasks in heterogeneous environments. We also studied different scheduling policies to minimize the job wait time in a multi-tenant environment. We opportunistically utilized available out-of-core accelerators for compute-intensive jobs such as machine learning and deep learning applications. As a result, we were able to demonstrate that our approach provides performance improvement according to metrics including overall execution time for different kinds of applications, with a detailed breakdown of each stage/operation, as well as system metrics (e.g., system utilization, cache hit ratio, etc.). In the following, we highlight this dissertation's specific research contributions.

Memory Management for In-memory Data Analytics In this work, we first present the problem of the existing memory management scheme that partitions memory space into static regions for different uses in Spark. We empirically detected memory contention among different usages, which leads to the *OutOfMemory* error and greatly degrades overall performance. Based on our observation of the dynamic memory consumption and the need for different usages of a workload during execution, we designed MEMTUNE, a dynamic memory manager for in-memory data analytics. MEMTUNE dynamically tunes computation/caching memory partitions at run-time based on workload memory demand and in-memory data cache needs. Moreover, if needed, the scheduling information from the analytic framework is leveraged to evict data that is unneeded in the near future. Finally, MEMTUNE also supports task-level data prefetching with a configurable window size to more effectively overlap computation with I/O. Our experiments show that MEMTUNE improves memory utilization, yields an overall performance gain of up to 46%, and achieves a cache hit ratio of up to 41% compared to standard Spark.

We further studied the key factors that affect memory consumption during the execution of a workload. We found that memory usage, including shuffle and task execution, is heavily influenced by data partitioning. This is because partitioning effectively de-

termines task granularity and parallelism. Moreover, the different phases of workload execution can have different optimal partitions. However, in current implementations, the tuning knobs controlling partitioning are either configured statically or involve a cumbersome programmatic process for affecting changes at run-time. Therefore, we propose Chopper, a system for automatically determining the optimal number of partitions for each phase of a workload and dynamically changing the partition scheme during workload execution. Chopper adopts a machine-learning-based approach to decide the best partition scheme as a workload runs. As a result, Chopper further improves workload performance by up to 35.2%.

Exploiting Specialized Accelerators for Big Data Analytics In this work, we provide a run-time approach to utilizing out-of-core accelerators such as GPUs for linear algebraic operations that are widely used in data analytics. We discovered that it is not always beneficial to utilize specialized accelerators that are well known for performance. However, the benefits of offloading computation depend on the task characteristics and input data. Based on this, we propose Arion, a hardware-acceleration-based approach for scaling-up individual tasks in Spark. Arion dynamically schedules tasks to different processing variants based on both workload statics and run-time resource utilization to improve resource utilization and avoid resource contention. We demonstrate the benefits of Arion for general matrix multiplication operations over large matrices with up to four billion elements by using Gramian matrix computation that is commonly used in machine learning. Experiments show that our approach achieves more than 2× and 1.5× end-to-end performance speed-ups for dense and sparse matrices, respectively, compared to MLlib, a state-of-the-art Spark-based implementation.

Workload-aware Heterogeneous Resource Scheduling In our research, we noticed the demand for incorporating multi-faceted workflows with both compute-intensive and data-intensive tasks, as well as intricate communication patterns. However, extant multi-cluster approaches are inefficient, as they require manual effort, including codes, to interact with APIs provided by resource managers, with data movement across clusters, and with porting across data formats. To this end, we designed Gerbil, a framework for transparently co-hosting multiple applications on the same cluster. We identified the wide needs for MPI and MapReduce applications and showcase Gerbil with these two by addressing the fundamental mismatch between MapReduce-based resource managers and MPI applications.

In later work, we further extended Gerbil by supporting more diverse applications, such as machine learning, graph computation, and scientific computing, on heterogeneous clusters with Rupam. Rupam is a heterogeneity-aware task scheduling system for big data platforms that considers both task-level resource characteristics and underlying hardware characteristics, and also preserves data locality. Rupam can support all kinds of applications due to its task-level scheduling. It also supports clusters with heterogeneous setups by run-time resource and bottleneck detection. Our experiments

show that RUPAM can improve the performance of representative applications by up to 62.3% compared with the standard Spark scheduler.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. Chapter 3 presents a solution of effective memory management for in-memory big data analytics. Chapter 4 presents method for optimal data partitioning scheme for a given workload. Chapter 5 introduces a hardware acceleration based approach for scaling-up individual tasks of Spark, and how we dynamically schedule tasks to different processors for overall best performance. Chapter 6 describes the fundamental mismatch between MPI applications and existing resource managers for MapReduce, and how we bridge the gap between these two to transparently co-host both applications in the same cluster. Chapter 7 presents a heterogeneity-aware task scheduling system for big data platforms. Chapter 8 concludes and discusses the future directions.

Chapter 2

Background

In this chapter, we provide background required for various aspects of our dissertation. This dissertation focuses on leveraging workload information for effective and efficient resource management to support the workload needs. This chapter summarizes the state-of-the-art research that is closely related to the major theme described above. We also compare them against our work by emphasizing the effectiveness, novelty, and benefits of proposed techniques and algorithms in this dissertation.

2.1 In-Memory Data Analytic Frameworks

Memory Management for In-Memory Data Analytic Frameworks Memory management is a well studied topic. LRU is widely used in caches [34, 54, 153, 177] and also the default in Spark, however, it does not factor in available dependency information as in MemTune. A number of distributed memory cache systems have also been designed for data intensive parallel processing frameworks [13, 22, 41, 51, 53, 60, 100, 147]. MemTune learns from these projects, but goes further to obtain tasking scheduling orders from the data analytics frameworks, and uses the information to prefetch and evict blocks as needed, and manage memory across the different system components.Systematic approaches such as iTask [87] resolve memory pressure by suspending and resuming running tasks. In contrast, MemTune focuses on memory resizing instead of managing the degree of parallelism and thus avoids the issues of task interference.

Memcached [13] and Redis [22] are highly available distributed key value stores, which are used to cache data in memory for large scale web applications [4, 27]. Megastore [60] offers a distributed storage system with strong consistency guarantees and high availablility for interactive online applications. These systems are complementary to MemTune but are designed for specific application domains and not general purpose framework such as Spark. Grappa [147] provides a distributed shared memory allowing programmers to use the aggre-

gated cluster resources as a large single machine. This is orthogonal to MEMTUNE as it does not use memory cache, and iterative applications still use disk for intermediate results.

Dynamic JVM heap management [66, 212] enables multiple JVMs to concurrently run on the same machine with reduced contention. MEMTUNE also dynamically changes the JVM heap size, but uses different JVM metrics as the contention indicator. Moreover, MEMTUNE's goal is to reduce contention between different application tasks and operation within a JVM to improve overall performance.

Parameter and memory tuning for large-scale data parallel computation have also been explored [30, 102, 125, 128, 132, 173, 196]. These systems and performance tuning guides propose either automatically change workload configurations or suggest configuration heuristics for MapReduce frameworks, and are orthogonal to MEMTUNE design and aims.

Tachyon [127] offers a reliable in-memory distributed caching layer that caches intermediate data across multiple frameworks. The key idea is using lineage to recompute lost data in case of failures. In contrast, MEMTUNE focuses on dynamically adjusting the memory allocation of task, shuffle and data caching based on application characteristics. Project Tungsten [26] has been proposed to move the memory management from JVM heap to off heap management. However, even when the memory is allocated off heap, the issue of deciding how much memory to allocate to tasks, shuffle and intermediate data remains. MEMTUNE is orthogonal and can be applied on top of project Tungsten.

The Spark open source community [28] has proposed to unify memory allocation within Spark. However, the focus is on how to reallocate RDD cache capacity for shuffle usage but not vice versa, and RDD cache management is left as is. In contrast, MEMTUNE exploits DAG task scheduling information, and offers a comprehensive solution for memory management of in-memory data analytic frameworks.

Optimizing Data Partitioning for In-Memory Data Analytic Frameworks A number of recent works have focused on improving the performance of big data processing frameworks by designing better built-in data partitioning [43, 44, 50, 119, 120]. In the following, we discuss projects that are closely related to CHOPPER.

Shark [201] supports a column-oriented in-memory storage, using RDDs [219], to efficiently run SQL queries and iterative machine learning functions. This is achieved by re-planning query execution mid-query if needed. Although Shark optimizes execution plan of a workload while the workload is running, unlike CHOPPER, Shark does not alter the number of partitions at each Spark compute phase, which can help optimize data movement between cluster nodes and load balance between tasks.

Spartan [106] automatically partitions the data to improve data locality in processing large multi-dimensional arrays in a distributed setting. It transforms the user code into an expression graph based on high-level operators, namely map, fold, filter, scan and join_update,

to determine the communication costs of data distribution between the compute nodes. Although Spark can be used to implement Spartan, in contrast to Chopper, the partitioning approach proposed in Spartan can not be applied directly to determine the optimal number of partitioning in Spark at each stage (where a stage may present unique execution characteristics that may not be represented by high-level operators).

Repartitioning MapReduce tasks has been actively studied [81, 89, 124, 133]. PIKACHU [89] improves load balancing of MapReduce [81] workloads on clusters with heterogeneous compute capabilities. It specifically targets the reduce phase of MapReduce execution and schedules jobs on slow and fast nodes such that jobs completion times are evened out across all nodes. On the other hand, Stubby [133] optimizes the given MapReduce workflow by combining multiple MapReduce jobs into a single job. It searches through the plan space of a given workflow and applies multiple optimizations, such as vertical packing to combine map and reduce operations from multiple jobs for reducing the network traffic. Similarly, SkewTune [124] mitigates the skewness in MapReduce applications by first detecting if a node has become idle in a cluster and then by scheduling the longest job on the idle node. The optimizations proposed in these systems share the goal of repartitioning with Chopper, but are specific to MapReduce programming model, and can not be applied directly to improve the partitioning scheme in a Spark application. This is because, the data partitioning for Spark need to consider characteristics across multiple compute phases within a workflow, and not just Map and Reduce phase. This concept can also help in virtual machine management in cloud computing [158].

The Hardware Accelerated Range Partitioning (HARP) [197] technique leverages specialized processing elements to improve the balance between memory throughput and energy efficiency by eliminating the compute bottlenecks of the data partitioning process. HARP makes the case that using dedicated hardware for data partitioning outperforms its software counterparts and achieves higher parallelism. The approach proposed in HARP is orthogonal to our work, and can be used in conjunction with Chopper.

Several other works also propose solutions to optimize the data partitioning problem in order to improve the processing and storage performance of multi-processor systems [70, 78, 121, 184, 226], cloud storage systems [45, 46, 47, 168, 190, 211], database systems [42, 48, 49, 52, 74, 146, 160, 164, 198, 225], and graph processing systems [72, 95, 170, 183]. While not directly applicable to the context of Spark and Chopper, the techniques proposed in these works can be leveraged in Chopper to further improve the data partitioning approaches inline with other system-level constraints, e.g., storage optimization.

2.2 Hardware Acceleration in Big Data Analytics

Advancing the computational capability of fundamental mathematical operations such as matrix operations is critical to brining large scale data analytics to bear upon knowledge

discovery across domains [63, 150]. For instance, scalable matrix inversion has been studied both with Hadoop [200] and Spark [135]. Elgamal et al. [85] show that optimizing matrix operations can improve both scalability and performance of ML algorithms. Gittens et. al [94] investigated both performance and scalability of the randomized CX low-rank matrix factorization on Spark. Zadeh et. al [215] optimize Singular Value Decomposition (SVD)—which can be considered as a form of matrix factorization—in Spark by distributing matrix operations to the cluster while keeping vector computations local to the driver node. In addition, scaling up individual machines for enhancing the throughput of matrix computations is also promising. For instance, numerous works have focused on optimizing GEMM operations [123, 144, 145] and, in turn, ML algorithms [35, 110, 136], through exploiting GPUs.

The Spark community has also initiated discussions regarding utilizing hardware accelerations from within the Spark [55, 56] platform, so as to realize the benefits from hardware acceleration for increased throughput of matrix computations on individual nodes [71, 113, 165]. To this end, a preliminary study about using hardware optimized libraries for both CPU and GPU on a single machine for matrix multiplication in Scala has shown promising results [185]. Similarly, HeteroSpark [131] showed that RMI can be used to reduce communication overheads between CPU and GPU in Spark. SparkNet [142] provides a Spark interface to use Caffe framework [110] for training large-scale deep neural networks, where the instance of Caffe framework on each node can use GPUs, and Spark maintains data on system memory, managed by CPUs. However, many challenges remain when employing combining the above scale-out and scale-up techniques. Some approaches are not applicable to general data analysis [142], and further, utilizing hardware optimized libraries often requires an in-depth understanding of the hardware characteristics for each computation in data analysis pipeline, which tends to be cumbersome and impractical. Consequently, efficient utilization of hardware acceleration in a cluster is not available in popular distributed matrix computation packages such as ScaLaPACK [64], PLAPACK [40], CombBLAS [68], and Elemental [162]. While the focus of the above approaches is different, they essentially offer C/C++ programming libraries atop MPI library. Our approach is complementary to these works and aims to reconcile the Spark philosophy of providing a general-purpose, fault-tolerant data analysis platform with benefits of using hardware optimized libraries for extracting higher performance, specifically for crucial matrix multiplication operations.

2.3 Heterogeneity-aware Resource Management

Resource Management for Heterogeneous Workflows A number of recent works focus on bringing MPI and MapReduce models together. Ye et al. [213] proposed a modified OpenMPI that allows MPI jobs to run on Hadoop clusters via the Hadoop streaming [24] interface. Bai [59] implemented a hybrid framework of iterative MapReduce and MPI. These approaches have used Hadoop 1.0, and are limited in scope or are specialized. In contrast,

Gerbil leverages the state-of-the-art YARN and provides general-purpose support for unmodified MPI applications on YARN. Conversely, projects such as MapReduce-MPI [18], DataMPI [137], and MR+ [19] supports MapReduce or Hadoop-friendly data-intensive jobs on MPI resources. However, these approaches do not port the well-established ecosystem around Hadoop, and thus are limited in utility. In contrast, Gerbil brings the entire MPI support to YARN and thus is expected to have easier transition and faster adaptation. The closest concept to Gerbil is Hamster [6], which was originally left unfinished but has recently been worked on by Yang et al. [29]. While Hamster also aims to support MPI applications on YARN, Gerbil is different and unique in its focus on efficiently provisioning and allocating resources. Gerbil also provides different resource allocation strategies that allow users to better match the resource allocation to the needs of their applications.

A number of works [9, 39, 193] also provide better application workflow processing by generating multi-language and multi-environment sub-workflows that can then be executed on clusters supporting the language/environment, and are complementary to Gerbil. However, Gerbil supports execution of two different models on the same set of resources and offers an integrated solution that avoids issues of supporting multiple types of resources/clusters and unnecessary data movement.

Task Scheduling for Big Data Frameworks in Heterogeneous Environment Rolling server upgrades is a common practice in large scale clusters and data centers, which inherently make the systems more heterogeneous [126, 155]. Thus, extensive work [65, 93, 103, 161, 171, 179, 180, 182] has been conducted for sharing heterogeneous clusters among multiple applications in both cloud environments and local clusters. These works focus on multi-tenancy environment and allocate resources in a heterogeneous environment to an application based on application types. In contrast, Rupam monitors and captures the diverse task and hardware characteristics, and thus aims to create a fine-grained resource scheduler with the aim to improve performance for individual tasks instead of just at application level (as applications can have tasks with varying demands). The design of Rupam is orthogonal to works on heterogeneous resource management, and can co-exist and benefit from them as a second-level scheduler.

Scheduling optimization for MapReduce jobs in heterogeneous environment is also extensively studied both for Hadoop [36, 86, 90, 188, 192, 216] and Spark [209]. Here, techniques such as straggler detection via progress rate and time estimation for better schedule and reschedule tasks are used. These approaches rely on application monitoring to make reactive scheduling decisions. In contrast, Rupam considers task characteristics as well as resource heterogeneity and utilization when scheduling a task. Rupam also adopts the idea of reactive scheduling as needed to avoid suboptimal decisions. Another aspect is taken up by works such as [36, 65, 75, 92, 172], which consider the resource utilization when scheudling tasks. However, heterogeneous cluster scheduling that considers multiple resources is a hard problem, both in theory [96] and practice[36, 98, 134]. The challenges here stem from the dynamic

interactions between processes (or tasks) within a single application such as synchronization, load balancing, and the inter-dependency between processes; and the dynamic interactions between applications such as the interleaving resource usages with other collocated independent workloads [210]. To reduce the complexity of the problem, these works focus on dominant resource optimization per a stage and assumes all tasks in the same Map/Reduce stage share the same resource characteristics. In contrast, RUPAM considers resource usage pattern for each task and adopts a heuristic to reduce complexity and thus improve performance. Tetris [98] proposes to pack multiple resources in a cluster for scheduling tasks. It applies the heuristics for the multi-dimensional bin packing algorithm. However, Tetris works with YARN in a homogeneous environment and assumes that tasks of a job have the same resource requirements to reduce the search space. In contrast, RUPAM takes a step further to consider both heterogeneous cluster resources and heterogeneous tasks, within Spark's own task scheduler.

Chapter 3

Dynamic Memory Management for In-memory Data Analytic Platforms

3.1 Introduction

Emerging in-memory distributed processing frameworks such as Spark [217] and M3R [174] are experiencing rapid growth and adoption due to their use in big data analytics. A crucial reason for the success of these frameworks is their ability to persist intermediate data in memory between computation tasks, which eliminates significant amount of disk I/Os and reduces data processing times. Consequently, for iterative jobs, in-memory processing has been shown to outperform the well-established Hadoop [31] model by more than ten times [217]. Such performance boost has led to the establishment of in-memory processing as the enabling technology for different data analytic platforms. For instance, a comprehensive ecosystem with a rich set of features has been developed atop Spark, including SQL query [57, 201], machine learning [139], graph computing [202] and streaming [220].

The key resource enabling the performance acceleration of the above platforms is *memory*. Failing to persist the whole working set in memory causes disk I/O or re-computation that leads to performance degradation. However, not all memory can be used for caching; applications also require memory for processing and data shuffling. Thus, there are opposing demands of caching and processing on the memory, which are growing with larger data sets and complex analysis tasks. Reconciling these demands is non-trivial. The current approach adopted in Spark is to statically configure memory partitions based on user specifications. However, this entails that users have deep knowledge about their workloads including process working set size, input data size, and data dependency. Given that the frameworks are general purpose, such a "best configuration" differs significantly across workloads. Moreover, determining a best configuration is hard and cumbersome and often not even possible as a users may be simply employing a prepackaged analytics application and not intimately aware

of its system-level characteristics. This is problematic, as we show in our evaluation that there is a large penalty for using mismatched configurations. Furthermore, we have observed that even in a single workload, the memory usage changes during execution. This is due to the change of data dependency and task working set size. Thus, a static configuration approach cannot capture such varying workload behavior, consequently leading to degraded performance.

Dynamic memory tuning at runtime can help improve memory resource utilization and reduce memory contention between data cache and process and shuffle memory. While promising, this is a challenging task due to two reasons. First, dynamic tuning requires accurate accounting information about both shuffle and tasks memory consumption, which is not available, and memory tuning can be counterproductive when such information is lacking. Second, it is difficult to decide which data to keep in memory and which data to evict, especially when multiple datasets, e.g., Resilient Distributed Dataset (RDD) [217], needed by the same processing stage cannot be fit into memory. Similarly, if consecutive stages use different RDDs, caching of data from a previous stage may be useless and unnecessarily increase the pressure on memory.

In this paper, we address the above problems and propose MEMTUNE, an approach that uses dynamic Directed Acyclic Graph (DAG) [194]-aware memory tuning for DAG-based in-memory distributed processing platforms. The goal of MEMTUNE is to improve overall memory utilization and reduce performance-degrading memory contention between data cache, process and shuffle memory. MEMTUNE monitors the task memory consumption using statistics such as garbage collection duration and memory paging frequency, and uses the information to dynamically change the data cache size. We also exploit the DAG execution graph of tasks to prefetch data that would be needed by the next stages, thus overlapping the computation and I/O to improve performance.

Specifically, this paper makes the following contributions.

1. We empirically study and demonstrate the impact of memory contention, caching policy and data cache size on performance for workloads running on in-memory data analytic platforms.

2. We design and implement MEMTUNE, a dynamic memory manager atop Spark, which adjusts the cache size and cached data at runtime to capture varying workloads demands and enhance performance.

3. We design and implement an automated algorithm that uses monitored memory statistics and DAG execution flow to determine efficient data cache size and caching policy.

4. We evaluate MEMTUNE in Spark using representative and diverse workloads from SparkBench [129]. Our results demonstrate that compared to static configuration, MEMTUNE reduces workload execution time by up to 46%. Moreover, MEMTUNE

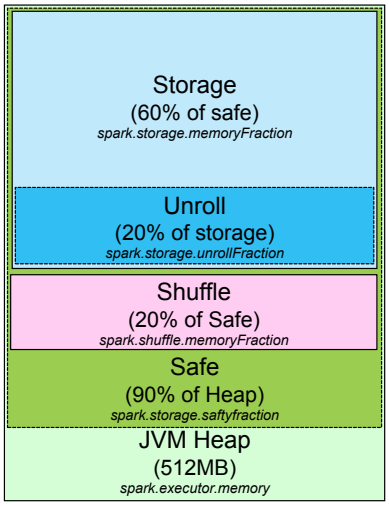


Figure 3.1: Typical memory partitioning used in Spark.

effectively detects the desired memory demand of tasks in each stage and change the data cache size accordingly, thus improving memory hit ratio by up to 41%.

3.2 Background and Motivation

In this section, we first discuss memory management used in Spark. Next, we motivate the need for our proposed solution through an empirical evaluation of Spark on a local cluster, SystemG [25].

3.2.1 Spark Memory Management

In Spark, data is managed as an easy-to-use memory abstraction called resilient distributed datasets (RDDs) [217]. An RDD is a collection of objects partitioned across a set of machines. Each machine retains several partitions (blocks) of an RDD in memory. An RDD block can be replicated across nodes for resiliency, and blocks can also be recomputed based on the associated dependencies if the data is lost due to machine failure. Computation is done in the form of RDD actions and transformations, which can be used to capture the lineage of a dataset as a DAG of RDDs, and help in RDD (re)creation as needed. Such DAGs of RDDs are maintained in a specialized component, DAGScheduler, which also schedules the tasks as needed.

Spark is deployed using a *driver program* running on a master node of a resource cluster and several *executors* running on worker nodes. Executors are launched as JAVA processes within which all tasks are executed. Each executor allocates its own heap memory space for

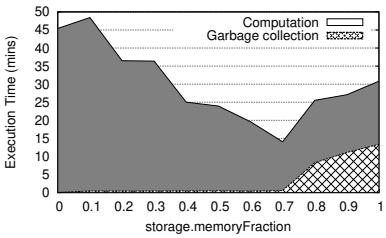


Figure 3.2: Total execution time and garbage collection time of *Logistic Regression* under `MEMORY_ONLY`.

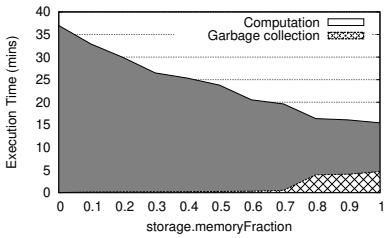


Figure 3.3: Computation time and garbage collection time of *Logistic Regression* under `MEMORY_AND_DISK`.

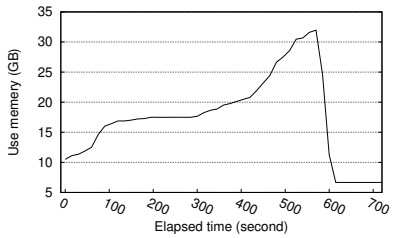


Figure 3.4: Memory usage of *TeraSort*.

caching RDDs. Figure 3.1 shows the memory partitions used by an executor. By default, Spark allocates a maximum of 90% of the heap memory size as *safe space* for RDD cache and shuffle sort operations, and reserves the remaining 10% of the heap for tasks processing. The figure shows how the safe space is further partitioned for RDD storage, shuffle sort operations, and RDD serialization/deserialization. If the assigned RDD cache is full, any remaining RDD blocks will either be re-computed or spilled to disk based on user specification, i.e., under Spark options `MEMORY_ONLY` and `MEMORY_AND_DISK`, respectively.

3.2.2 Empirical Study

The memory management adopted in Spark is static. However, as discussed earlier, workload variance implies that such fixed memory partitioning may not offer the best performance. To quantify the impact of memory management on workload performance, we conduct an empirical study using SparkBench [129], a comprehensive benchmark suite for Spark. We use Spark version 1.5, the latest release of Spark, with Hadoop version 2.6 providing the storage layer. All experiments are done on 6 nodes of our SystemG cluster. Each node has two 4-core 2.8 GHz Intel Xeon processors and 8 *GB* memory, and boosts a 1 *Gbps* Ethernet interconnect. One of the nodes is configured to be the master node and the others as workers for both HDFS and Spark. We configure each worker node to have one executor with 6 *GB* memory, leaving the remaining memory for OS buffer and HDFS data node operations. Each executor has 8 task slots, one for each CPU core. We repeated each experiment 5 times, and in the following report the average results.

Memory Contention

In our first test, we study the performance under varying configurations such as persistence level and `spark.storage.memoryFraction` values. We study two persistence levels, the default `MEMORY_ONLY` and `MEMORY_AND_DISK`. For this test, we use the *Logistic Regression*

workload. We ran the workload with an input data size of 20 GB, with 40 GB total system memory capacity. We change the configuration parameter from 0 to 1, i.e., from no memory to cache RDDs to all of the memory used for caching. We set the workload iteration limit to three to ensure that the experiment can finish in a reasonable amount of time without sacrificing the accuracy of the inferences drawn.

Figure 3.2 shows the overall execution time that includes compute time and garbage collection (GC) time of the workload under the default level of `MEMORY_ONLY` with increasing `spark.storage.memoryFraction`. We record the GC time in each executor during the workload execution and report the average here. We can see that the overall performance is the best when the parameter is configured to a value of 0.7. Lower values result in larger compute time. This is because there is not enough memory for caching RDDs, forcing needed RDDs to be recomputed. We also found that when the parameter value is configured to a higher value (from 0.8 to 1), the overall execution time is once again increased. Upon closer inspection, we find that the reason for this increase is that the GC time is now increased due to less memory being available to the JVM, which causes more frequent garbage collection. Thus, simply allocating more memory to RDD caching is not beneficial.

Note that the compute time includes RDD computation, task computation and framework overhead. Increasing the memory fraction for RDD increases contention with the executor's other tasks. This is because Spark utilizes part of the executor memory for RDD storage, meanwhile tasks are also launched inside the same executor. If too much memory is allocated for RDD caching, the executor will again incur a huge GC overhead, resulting in degraded performance.

Figure 3.3 shows that a similar behavior is observed under `MEMORY_AND_DISK` as well. With this persistence level, the GC overhead is not as pronounced as the default memory-only level. This is because spilling RDDs to disk avoids recomputation, which in turn decreases memory contention.

From this test, we see that to get the best performance, we need to balance the memory fraction between RDD cache and compute memory consumption. For the above case, the best fraction turned out to be 0.7, but this may not hold for other real workloads that exhibit different characteristics. For example, memory-intensive workloads such as *Logistic Regression* require more memory for tasks to execute, while I/O-intensive workloads such as *TeraSort* require less memory but have frequent accesses to data. Static configuration relies heavily on users' knowledge about the workload characteristics as well as the deployment setup to find the best configuration. To remedy this, the Spark community recommends using a value of 0.6 for `spark.storage.memoryFraction`, which works for general workloads with modest input data sizes and system set up. However, this approach fails to handle big input data sizes (`OutOfMemory` error was thrown during our test runs with big data sizes). The value is also not suitable for long running jobs with varying characteristics. To investigate this further, we ran several experiments with different workloads and input sizes and show the results in Table 3.1. The table points out the maximum input data size that Spark

Workload	Input size (GB)
Logistic Regression	20
Linear Regression	35
Page Rank	2
Connected Components	< 1 (16M nodes, 99M edges)
Shortest Path	6

Table 3.1: Maximum input size under Spark with default configurations.

was able to handle using the community-suggested parameter values without `OutOfMemory` errors. Note that for some workloads the problem started with as small an input as 1 GB, which is a worrisome observation for a big data processing framework.

Static Configuration

Once set, static configuration is effective throughout an application execution, which makes it hard to adapt to workloads with dynamic memory demands. For example, Figure 3.4 shows the memory use of *TeraSort*. We set the RDD cache size to 0 in order to observe the task memory consumption. We observe a burst in the memory usage after about 8 minutes. Under static configuration, a user would have to configure the RDD cache size to a small number throughout the execution to accommodate such a burst in task memory requirement, thus losing the opportunity to utilize the memory for RDD cache in earlier stages. An ideal dynamic approach, on the other hand, can start with more memory for RDD, reap the benefits of caching, and then reduce the RDD memory to accommodate the burst and so on. MEMTUNE aims to realize such dynamic management.

Memory Management Policy

Spark uses LRU [153] policy for evicting RDD blocks from memory. If a block is evicted, it is not brought back to memory again, and accessed directly from disk or re-computed in cases it is accessed again. The policy is effective, but does not consider an RDD's dependency and need for future stages, information that is available in the workflow DAG. For example, in the *Shortest Path* workload, there are 7 stages and 5 RDDs (RDD3, RDD16, RDD12, RDD14, and RDD22) need to be cached. Among the 7 stages, 5 stages have RDD dependencies. Table 3.2 shows the RDD dependencies and the total sizes of the RDDs for this workload. Figure 3.5 shows the RDD sizes in the beginning of the 5 stages that have RDD dependencies. In contrast, Figure 3.6 shows the ideal RDD sizes that the stage needs. Note that with default configuration in our test cluster, we have 14 GB in total for RDD storage.

We can see that for stage 3 and stage 4, LRU works well. However, stage 5 is solely dependent

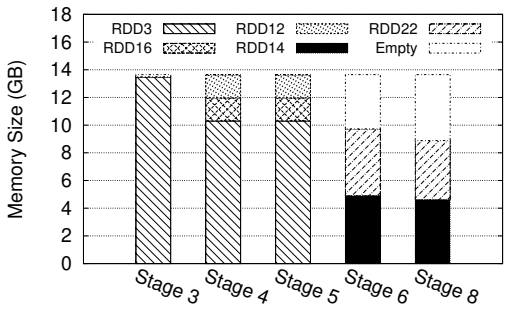


Figure 3.5: RDD sizes in memory in different stages of *Shortest Path* under default configurations.

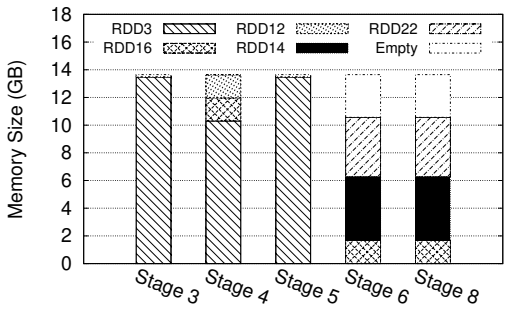


Figure 3.6: Ideal RDD sizes in memory based on stage RDD dependencies of *Shortest Path*.

Stages	**RDD3 (18.7G)**	**RDD16 (4.8G)**	**RDD12 (4.8G)**	**RDD14 (11.7G)**	**RDD22 (12.7G)**
Stage 3	×	·	·	·	·
Stage 4	·	×	×	·	·
Stage 5	×	·	·	·	·
Stage 6	·	×	·	·	·
Stage 8	·	×	·	·	·

Table 3.2: RDD dependencies for different stages of *Shortest Path*. '×' and '·' denote whether a stage is dependent on an RDD or not, respectively.

on RDD3, but some RDD3 blocks are evicted in stage 4. Also, stage 6 and stage 8 are dependent on RDD16, while no RDD16 is cached in memory because it is completely evicted from memory after stage 5. Since the evicted RDDs will not be brought back to memory again, there is extra empty room left in both stages. This RDD placement policy leads to an inefficient use of memory resource. MemTune aims to address this by designing better workload-aware memory management.

3.3 System Design

In this section, we first describe the architecture of MemTune, followed by how we achieve dynamic memory tuning and RDD cache management.

3.3.1 Architecture Overview

Figure 4.5 shows the overall architecture of MemTune. Although we have implemented MemTune atop Spark, MemTune can also work in multi-tenancy environments with other

API	Description
`double getRDDCache(AppID aid)`	Returns the current RDD cache ratio for the application with ID `aid`.
`void setRDDCache(AppID aid, double rddCacheRatio)`	Sets the RDD cache ratio to value `rddCacheRatio` for the application with ID `aid`.
`void setPrefetchWindow(AppID aid, double prefetchWindow)`	Sets the prefetch window with a value `prefetchWindow` for the application with ID `aid`.
`void setEvictionPolicy(AppID aid, EvictionPolicy ep)`	Sets the RDD eviction policy for the application with ID `aid`.

Table 3.3: Key APIs provided by MEMTUNE.

cluster resource managers such as YARN [187] and Mesos [103]. This is because MEMTUNE manages resources that are provided to it, and thus can naturally extend to containers supported by YARN or Mesos like systems.

MEMTUNE has two key centralized components, *controller* and *cache manager*, and a distributed component, *monitor* that is implemented within each executor on participating nodes. The controller implements the main logic flow of MEMTUNE, such as determining and setting the RDD cache size for each stage, the RDD eviction policy, and the prefetch window size. The cache manager implements the APIs described in Table 3.3. The distributed monitors are responsible for gathering runtime statistics such as garbage collection time, memory swap, task execution time per stage, and input and output dataset sizes. The monitor is designed to be an extensible component so that additional information can be easily captured as needed.

After an application is submitted through spark-submit scripts, Spark launches a Spark driver program together with a *SparkContext* object. Within SparkContext, MEMTUNE's controller and cache manager are instantiated along with the *DAGscheduler* and *BlockManagerMaster*. Next, Spark launches its executor components on the participating nodes, which results in the MEMTUNE monitors being deployed on the cluster as well. The controller periodically gathers data from each monitor and uses the information to adjust RDD cache sizes on nodes, and if needed, selects blocks to evict or prefetch. The controller then communicates this information to the cache manager, which in turn invokes the BlockManagerMaster requests to perform the needed operations and execute the commands on the working nodes.

To support dynamic memory configuration, we modify BlockManagerMaster to allow dynamically changing of RDD cache sizes and triggering RDD eviction if the cache is now smaller than the cached data. MEMTUNE supports a set of APIs for this purpose, as shown in Table 3.3. Typically, MEMTUNE will use these APIs to manage RDD cache automatically. However, the APIs also allow users to explicitly control RDD cache ratios, RDD eviction policy and prefetch window during application execution. MEMTUNE automatically manages the RDD cache by efficient eviction and prefetching with a dynamically-adjusted

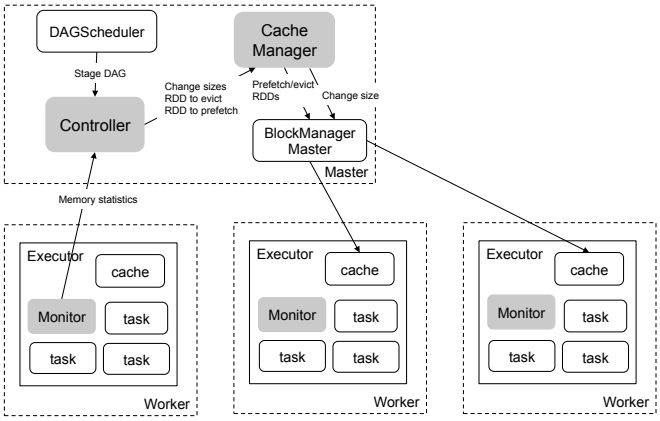


Figure 3.7: System architecture of MemTune.

prefetch window. By considering application I/O demands in controlling the prefetch window size—which determines how much data to be prefetched from disks to memory—MemTune effectively overlaps computation with disk I/Os and avoids I/O contention.

3.3.2 Dynamically Tuning RDD Cache and JVM Heap Size

As shown in Figure 3.1, the JVM memory is used by task execution, shuffle sort operations, and RDD cache. It is crucial to coordinate the different needs, especially, when there is a contention. Furthermore, node memory outside of JVM provides buffer space for shuffle reads and writes. If there is not enough space to buffer the shuffle data, significant disk I/O would occur, degrading performance. Thus, if the workload is not memory-intensive rather shuffle-intensive, we can enlarge such buffer space by shrinking the JVM heap size. The memory contention between such different needs is shown in Table 3.4. We observe that there is no contention between shuffle and task execution. This is because shuffle operations usually happen in the end (writes) or start (reads) of a stage. Spark determines a new stage based on whether there is a shuffle operation or not, and the new stage does not have RDD cache dependencies, rather the RDDs are obtained through shuffle reads. Although during shuffle writes, there are RDD cache dependencies, data can only be written when tasks are finished with their computation.

Moreover, a challenge here is that we need to ensure that applications can finish without errors, because task failures due to memory errors are not recoverable. Consider how such errors are handled in Spark. Upon getting a memory error, a user has to either reduce the RDD cache ratio or increase the task's parallelism, and then re-launch the entire application. However, this is a trial-and-error approach, and the process may have to be repeated until the out-of-memory errors disappear. This is cumbersome and frustrating especially if the application is a long running one.

Case#	Shuffle	Task	RDD	Contention	Action
0	N	N	N	N	N/A
1	N	N	Y	RDD	↑JVM, ↑cache
2	N	Y	N	Task	↑JVM
3	N	Y	Y	Task, RDD	↑JVM, ↓cache
4	Y	N	N	Shuffle, RDD	↓cache, ↓JVM

Table 3.4: Cases of memory contention and corresponding actions taken by MEMTUNE. Yes (Y) and No (N) under the Shuffle, Task and RDD columns denote whether there is memory contention detected for that usage.

We leverage the above observations in MEMTUNE to prioritize and first allocate sufficient task memory, then shuffle sort memory, shuffle buffer, and finally RDD cache. It is challenging to determine both the task and shuffle memory demands. Currently MEMTUNE adopts indicators of GC ratio and swap ratio, as well as three different thresholds, Th_GCup, Th_GCdown, and Th_sh to determine whether there is enough memory for task execution and shuffle operations. The indicators can be extended to other indicators with more accuracy such as task memory footprint in the future. Currently, the thresholds are set based on observations from our experimentation, but they can also be exposed to users in the form of parameters. For the RDD cache, we start with the maximum fraction of 1 instead of the default of 0.6, and adjust it dynamically as needed to accommodate other demands as follows. If the monitors detect a shuffle operation, it implies that there is currently no need to access the RDD cached data. In this case, we prioritize shuffle operations over RDD cache and reduce the cache size. Conversely, if the tasks are not in the shuffle phase, we allocate more memory for the RDD cache while also ensuring (using the GC etc. indicators described above) that the tasks have enough memory for execution.

Table 3.4 classifies five cases of contention and the corresponding actions performed by MEMTUNE. There are two tuning knobs to mitigate the contention, namely the JVM size and the RDD cache size. By default, we set the JVM heap size to the maximum available memory of each physical node to maximize utilization. We tune the JVM size asymmetrically, in the sense that we prefer to only reduce the JVM size temporally when we detect shuffle contention. We always first increase JVM size whenever we detect task or RDD memory contention, if JVM heap size has been set to less than the maximum memory allocation in a prior epoch. If the JVM heap is already at its maximum value, we proceed as follows. If there is RDD contention only, we conservatively increase the RDD cache size by one unit. If there are both Task and RDD contention, priority is given to Tasks and the RDD cache size is reduced by one unit. Finally, if there is shuffle contention, both the RDD cache and JVM heap size are reduced by the same amount α_{sh}, i.e., we give a portion of the memory allocated to RDD cache to shuffle (before giving up the memory allocated for tasks). Note that if there is no contention, MEMTUNE does not perform any actions in the current epoch.

The main loop (line 4 to line 20) of the Algorithm 20 shows the steps taken by the controller when we cannot simply increase the JVM heap size to mitigate memory contention. Using

Algorithm 1: Controller workflow algorithm.

Input: *block_size*, *RDD_size*, *shuffle_size*

begin

```
1   RDD_list ← calculate dependent RDD list of the stage;
2   if RDD_size <sizeof (RDD_list) then
3       prefetch(window_size);
    end
4   while true do
5       {gc, swap} ← get GC and page swap information from monitor;
6       gc_ratio ←caculate_gc_ratio(gc);
7       swap_ratio ←calculate_swap_ratio(swap);
8       if gc_ratio > Th_GCup then
9           RDD_size− = block_size;
10          evict_rdd(block_size);
11
        end
12      if swap_ratio > Th_sh then
13          α_sh ← block_size × number_of_tasks;
14          RDD_size− = α_sh;
15          evict_rdd(α_sh);
16          shuffle_size+ = α_sh;
17          jvm_size− = α_sh
        end
18      if gc_ratio < Th_GCdown then
19          RDD_size+ = block_size;
        end
20      sleep(5);
    end
end
```

the periodically gathered runtime statistics (GC time and page swap amount) from the monitor, MemTune calculates the GC ratio and swap ratio, and checks if the GC ratios exceeds Th_GCup, i.e., an upper threshold for GC ratio. If this is the case, MemTune determines that there is a memory shortage for tasks and reduces the RDD cache size by one unit size. We choose one RDD block size as the unit size because this is the minimum amount of RDD cache size that we can evict to release memory. Next, if the swap ratio for an executor exceeds the Th_sh, it means that memory is needed by the N_s tasks that are performing shuffle in the executor. To provide the memory, we reduce the RDD cache by N_s units to ensure that none of the shuffle tasks suffer from swapping. We also increase the shuffle sort size and decrease the JVM heap size to give more memory for I/O buffers. Finally, if the GC ratio is too low (less than Th_GCdown) implying that the tasks are not

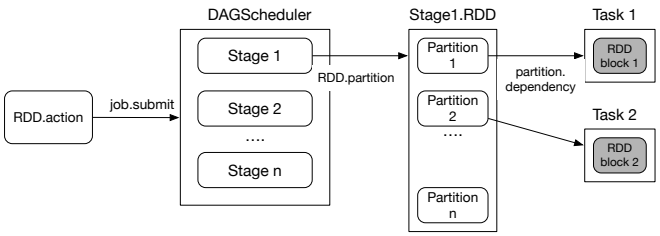


Figure 3.8: Generation of tasks based on RDD dependency after an RDD action is submitted to DAGSchduler.

using much memory, we increase the RDD cache size by one unit to give more memory to the RDD cache. We conservatively set Th_GCdown smaller than Th_GCup to give priority to task execution memory. Note that the controller triggers these steps periodically so that the size can be changed gradually. Even if the controller makes a sub-optimal decision in the current epoch, it can be improved/corrected in the next epochs.

3.3.3 RDD Eviction Policy

Our automatic RDD cache manager may entail that some RDDs need to be evicted. For this purpose, we do not employ the default LRU policy for RDD cache. Instead, MEMTUNE leverages the DAG information generated by Spark task scheduler for selecting eviction candidates.

As seen in Figure 3.8, an RDD action triggers *SparkContext* to submit a job, which is then handled by the *DAGScheduler*. The scheduler divides a job into stages based on RDD dependencies, and submits the stages one by one. Each stage has a group of tasks that actually performs the computation. These tasks are generated based on RDD blocks that the tasks need to produce. Here, the controller calculates the RDD block dependency for each block (gray blocks in Figure 3.8) and associates this information with the tasks for supporting task-level prefetching later. Thus, in each stage of a job, we have a collection of tasks with their dependent RDD blocks. We refer to this group of RDD blocks as *hot_list*. Note that all RDD eviction and prefetching are within fine-grained block level. This enables the controller component to have the DAG information for each stage. Moreover, the controller can commence prefetching with a *hot_list* before the associated tasks are submitted. During task execution, finished tasks are also tracked to help eviction as needed by adding the RDD blocks of such tasks to a $finished\_list$.

Two scenarios can occur that can trigger evicting RDD blocks. The first is when the controller reduces the RDD cache size. In this case, the controller first scans the current in-memory cached RDD blocks ($memory\_list$) obtained from the cache manager. If a block is found that does not belong to the *hot_list*, that block is chosen for eviction. Otherwise, a block from the $finished\_list$ is evicted if it is not empty. If no eviction candidate is available

so far, a block in *memory_list* with the highest partition (block) number is chosen since it is least likely to be used right away. The choice of evicting high-partition number blocks is based on the observation that Spark schedules tasks according to partition numbers in an ascending order, so we are evicting a block to be used farthest in the future, i.e., effectively an LRU policy. This eviction policy prevents eviction of current blocks that are in use. Finally, the controller uses the cache manager to evict the chosen block and reduce the RDD cache size.

The second eviction case arises when a new RDD block needs to be placed in memory, but the cache is full or does not have sufficient room. The default behavior in Spark is to first check if there are blocks of other RDDs in memory, if so, those are evicted. Otherwise, a warning message is generated to indicate that blocks from the same RDD are being evicted, and then LRU RDD blocks are spilled to disk. This is not desirable as blocks that may be needed soon may be evicted even though the cache contains blocks on the $finished\_list$ that are no longer needed. To remedy this, we changed the default policy to first evict $finished\_list$ blocks before spilling others. The goal of MemTune eviction policy is to utilize RDD dependency information to achieve better caching performance. However, the users can still use the explicit control APIs of MemTune to implement their own custom policies as needed.

3.3.4 Prefetch and Prefetch Window

The controller checks to see if all the dependent RDD blocks are cached at the beginning of each stage (Algorithm 20 line 1 to line 3) and triggers prefetching as needed. MemTune goes further to also aggressively prefetch RDD blocks from disk to effectively overlap the task computation and I/O. The intuition is that since we know the task scheduling sequence and the task running status per machine, we have exact knowledge about which RDDs will be accessed in the next epoch and can prefetch them into memory. One exception is that when the tasks are determined to be I/O bound, indicating that there is little additional disk bandwidth for prefetching, in which case prefetching is not done.

To achieve prefetching, the cache manager creates a prefetch thread running on each executor. The thread continuously prefetches data as long as the prefetch window is not filled. MemTune uses an initial prefetch window size that is twice the degree of task parallelism (number of tasks in each executor). This is because tasks are executing in parallel, and data are consumed in a wave (number of tasks). If the controller detects memory contention and decides to drop an RDD block from memory, the window size will be decreased by one wave. If no contention is detected for tasks and shuffle, the size is increased again to the maximum used as the initial value. This policy also ensures that memory priority is given to task execution.

MemTune keeps a list of blocks to prefetch ($prefetch\_list$) and a list of blocks that are prefetched ($cached\_list$). Upon trigger, the prefetching thread scans the RDD blocks that

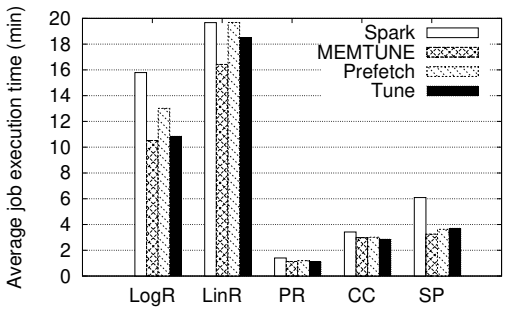


Figure 3.9: Execution time of studied workloads under the default Spark, MEMTUNE, MEMTUNE with prefetch only, and MEMTUNE with tuning only. LogR: *Logistic Regression*, LinR: *Linear Regression*, PR: *Page Rank*, CC: *Connected Components*, SP: *Shortest Path*.

are present in a node disk (*disk_list*), finds any RDD blocks that are in the *hot_list*, and puts them on the $prefetch\_list$. The $prefetch\_list$ blocks are then read one by one in ascending order of partition numbers and placed on the *cached_list*. The prefetching continues until the size of the *cached_list* is equal to the prefetch window size, which is a configurable parameter as indicated above. When a task is started, access to any block on the *cached_list* results in it being moved to the standard cached block (*memory_list*). This makes room for further prefetching. The prefetching thread keeps track of the size of the *cached_list* and perform prefetching whenever the size is less than the prefetch window.

3.3.5 Discussion

In a multi-tenant environment where there are multiple applications running at the same time, we also need to consider other factors such as service level agreements (SLAs), job priority, and overall system utilization when deciding the memory allocation. MEMTUNE does not have a global system view, however, the underlying resource managers can instruct MEMTUNE by setting a hard limit of JVM size so that MEMTUNE will not expand its memory for an application beyond what is allowed. While inside this hard limit, MEMTUNE strives to best utilize the memory resource. This would ensure that MEMTUNE improves individual allocated memory utilization of each application.

3.4 Evaluation

In this section, we demonstrate the efficacy of MEMTUNE on our SystemG setup (described in Section 3.2). We have implemented MEMTUNE in Spark by modifying about 20 classes. Mainly, we modified the Spark *DAGScheduler*, *BlockManagerMaster*, *BlockManager* classes to realize the controller, cache manager, and prefetcher components, respectively. We use

the built-in function `dropFromMemory` to evict RDD blocks, and implemented a new helper function `loadFromDisk` to load RDD blocks from disk to memory.

3.4.1 Overall Performance of MemTune

In our first test, we study the overall performance impact of MemTune. For this purpose, we use five workloads from the SparkBench [129] suite as shown in Table 3.1, with the maximum input sizes that can be run on Spark without errors. Among these five workloads, *Logistic Regression* and *Linear Regression* both have RDDs whose size is larger than the aggregated cluster RDD capacity. On the other hand, the graph computation workloads (*Page Rank*, *Shortest Path* and *Connected Components*) have smaller RDDs that can completely fit into the RDD cache under the default Spark configuration. However the graph workloads cannot complete successfully if we increase the input data size. Figure 3.9 shows the overall workload execution time under four scenarios: Spark with default configuration, MemTune with dynamic memory tuning only, MemTune with prefetch only, and MemTune with both dynamic memory tuning and prefetching enabled. We can see that MemTune performs comparable or faster than the default Spark (up to 46.5% improvement) for all workloads. Note that this performance improvement is compared against Spark with the default configuration (`storage.memoryFraction=0.6`). MemTune performs better than the optimal configuration (`storage.memoryFraction=0.7`) shown in Figure 3.2. *Page Rank*, *Connected Components*, and *Shortest Path* do not benefit much from MemTune because the input data size is not big enough to exhaust the memory and no RDD block is spilled to disk under the default Spark setup. When we increased the data size, the default Spark emitted `OutOfMemory` errors and failed the execution, while MemTune was able to finish execution without errors even with larger data set sizes. Moreover, except for *Shortest Path*, we see that most workloads benefit from dynamic configurations more than data prefetching. This is because most of these workloads are not CPU-intensive, resulting in short task execution times, and thus do no leave enough time to overlap I/O with computation through task-level prefetching. However, prefetching was able to help *Shortest Path* by overlapping task computation with I/O, and reduced the workload execution time by 46.5%. The overall average performance gain across the studied workloads achieved by MemTune is 25.7% compared to the default Spark.

3.4.2 Impact of Garbage Collection

Next, we repeat the previous set of experiments and collect the total garbage collection (GC) time on each executor during the entire application execution. We report the average ratio of GC time to overall application execution time. Figure 3.10 shows that MemTune imposes bigger GC ration than Spark with default configuration. This is because of two reasons: Spark does not exhibit a huge GC overhead under the default configuration of 0.6

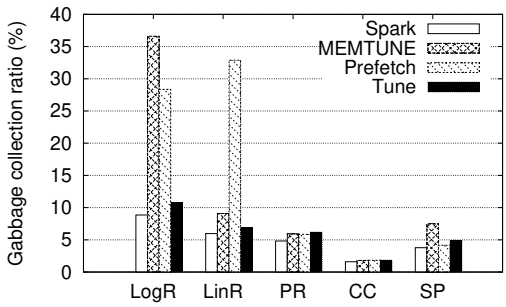


Figure 3.10: Gabbage collection ratio of studied workloads under different scenarios.

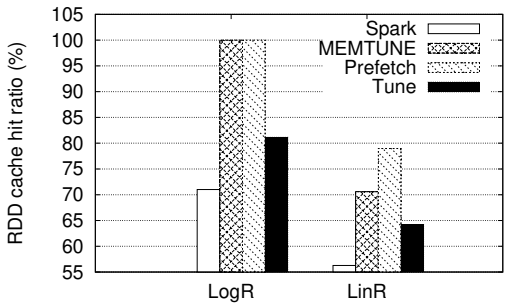


Figure 3.11: RDD memory cache hit ratio of studied workloads under different scenarios.

as shown in Figure 3.3. Secondly, dynamic memory tuning of MEMTUNE tends to increase RDD storage size when GC is not significant. Finally, data prefetching provided by MEMTUNE tends to bring RDD blocks to memory, which also increases the memory utilization. For *Linear Regression*, MEMTUNE has lower GC ratio compared to the Prefetch only case since MEMTUNE decreases RDD cache sizes for task computation while RDD blocks are prefetched.

3.4.3 Impact on Cache Hit Ratio

In our next test, we use *Logistic Regression* and *Linear Regression* to study the RDD cache hit ratio under MEMTUNE. We do not use the graph computing workloads as they fit in memory and have a 100% hit rate for the four studied scenarios. We can see in Figure 3.11 that prefetching under MEMTUNE results in the highest RDD memory cache hit ratio—up to 41% improvement compared to the default Spark. In contrast, dynamic tuning yields better hit ratios than the default Spark but not as higher as under the other two MEMTUNE scenarios. This is because dynamic memory tuning increases RDD storage sizes for both workloads, thus increasing the number of RDD blocks that are cached. Prefetching has the best cache hit ratio because it prefetches the RDD blocks into memory before the blocks are accessed. For *Logistic Regression*, MEMTUNE with both features enabled achieves the same cache hit ratio as prefetching. However, for *Linear Regression*, MEMTUNE with both features enabled achieves less than prefetching alone. This is because *Linear Regression* has a higher task memory consumption, thus when prefetching the data, dynamic memory tuning reduces the RDD cache size, thus reducing the amount of RDD blocks that are cached.

Discussion: Considering the above three experiments, we see that *Logistic Regression* shows less task memory consumption and benefits more from prefetching. On the other hand, *Linear Regression* shows more task memory contention, thus prefetching alone shows bigger GC overhead and benefits both from prefetching and dynamic tuning. The other three

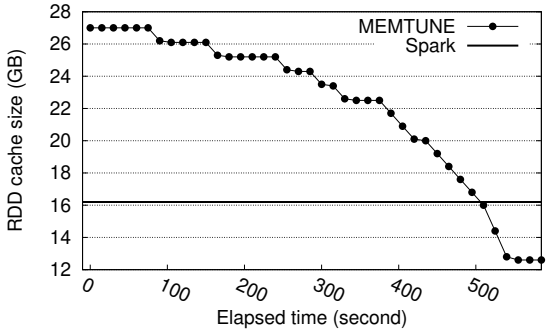


Figure 3.12: Dynamic change in RDD cache size during execution.

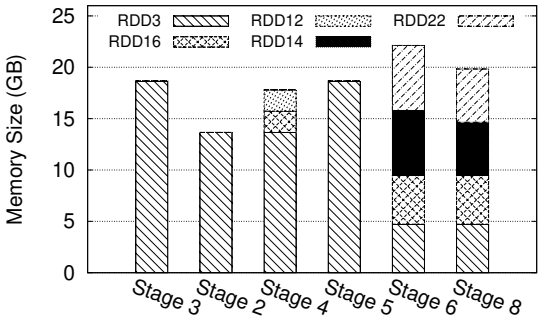


Figure 3.13: RDD cache sizes in memory with the default configuration under different stages with MemTune.

graph computation workloads have small input data sizes, thus show a modest resource consumption, and both the default Spark and MemTune performs similarly. Yet, MemTune is better as it can also process larger input data set sizes where the default Spark fails. MemTune's effectiveness is heightened whenever there is memory contention. RDD prefetch aims at increasing cache hit ratio of RDD blocks in memory, while dynamic memory tuning aims at increasing the memory usage while mitigating memory contention among different demands. Combining these two techniques, MemTune increases the memory utilization and the efficiency of RDD cache by overlapping computation with I/O.

3.4.4 Dynamic RDD Cache Size Tuning

As discussed in Section 3.2.2, *TeraSort* exhibits a dynamic memory usage demand that cannot be fully satisfied by a static configuration. Another characteristic of *TeraSort* is that it is shuffle-intensive. Most of its stages involve heavy shuffle I/Os. In our next experiment, we run the *TeraSort* workload with MemTune, and monitor the dynamic RDD size changes over the execution time. Figure 3.12 illustrates that MemTune starts with a high RDD configuration in the beginning, and decreases gradually throughout the execution. Recall that in Figure 3.4, we observed a large memory usage burst in the final stage of *TeraSort*. Although the algorithm of MemTune is conservative for catching such bursty memory consumption, by periodic tuning, MemTune is able to keep dropping the RDD cache size until it detects the contention falls below the defined threshold. Increasing the checking and tuning frequency would enable MemTune to react to memory contention more aggressively (though it can add monitoring overhead and may also cause thrashing, which underscores our current conservative approach to tuning).

3.4.5 DAG-Aware RDD Prefetching

As shown in Section 3.2.2, LRU RDD eviction policy works for some stages in a workload, but not for other stages that have RDD dependencies. In our next experiment, we run *Shortest Path* with 4 *GB* input graph data size under MEMTUNE. We show the RDD memory size in the beginning of the 7 stages in Figure 3.13. Table 3.2 shows the RDD dependencies of the 5 stages that are also applicable here. We see that unlike the default Spark (Figure 3.5), MEMTUNE brings RDD3 back to memory in stage 5 because MEMTUNE detects that stage 5 is dependent on RDD3. Likewise, MEMTUNE also brings RDD16 to memory in both stage 6 and stage 8. Moreover, on average RDD sizes in memory are also more than the default Spark, and there is no empty space left in the RDD cache. This is because MEMTUNE dynamically changes the RDD cache size and increases it compared to the default 0.6. This is also the reason for the changes in total RDD memory sizes.

3.5 Chapter Summary

We have presented the design of MEMTUNE, a dynamic memory management approach for in-memory data analytic platforms. MEMTUNE detects memory contention at runtime and dynamically adjusts the memory partitions between in-memory data cache, task execution, and in-memory shuffle sort operations. By leveraging workload DAG information, MEMTUNE proactively evicts and prefetches data with a configurable prefetch window and also employ task-level prefetching. Experiments with an implementation of MEMTUNE in the popular Spark framework shows an overall performance improvement of up to 46% for representative workloads. In our future work, we plan to improve memory usage estimations of MEMTUNE, and expand our system to multi-tenant environments.

Chapter 4

Optimizing Data Partitioning for In-Memory Data Analytics Frameworks

4.1 Introduction

Large scale in-memory data analytic platforms, such as Spark [217, 219], are being increasingly adopted by both academia and industry for processing data for a myriad of applications and data sources. These platforms are able to greatly reduce the amount of disk I/Os by caching the intermediate application data in-memory, and leverage more powerful and flexible direct acyclic graphs (DAG) based task scheduling. Thus, in-memory platforms outperform widely-used MapReduce [81]. The main advantage of a DAG-based programming paradigm is the flexibility it offers to the users in expressing their application requirements. However, the downside is that complicated task scheduling makes identifying application bottlenecks and performance tuning increasingly challenging.

There is a plethora of application-specific parameters that impact runtime performance in Spark, such as tasks parallelism, data compression and executor resource configuration. In typical data processing systems, the input (or intermediate) data is divided into logical subsets, called partitions. Specifically, in Spark, a partition can not be divided between multiple compute nodes for execution, and each compute node in the cluster processes one or more partitions. Moreover, a user can configure the number of partitions and how the data should be partitioned (i.e., hash or range partitioning schemes) for each Spark job. Suboptimal task partitioning or selecting a non-optimal partition scheme can significantly increase workload execution time. For instance, if a partition strategy launches too many tasks within a computation phase, this would lead to CPU and memory resource contention, and thus lose performance. Conversely, if too few tasks are launched, the system would have

low resource utilization and would again result in reduced performance.

Moreover, inferior partitioning can lead to serious data skew across tasks, which would eventually result in some tasks taking significantly longer time to complete than others. As data processing frameworks usually employ a global barrier between computation phases, it is critical to have all the tasks in the same phase finish approximately at the same time, so as to avoid stragglers that can hold back otherwise fast-running tasks. The right scheme for data partitioning is the key for extracting high performance from the underlying hardware resources. However, finding a data partitioning scheme that gives the best or highest performance is non-trivial. This is because, data analytic workflows typically involve complex algorithms, e.g., machine learning and graph processing. Thus, the resulting task execution plan can become extremely complicated with increasing number of multiple computation phases. Moreover, given that each computation phase is different, the optimal number of partitions for each phase can also be different, further complicating the problem.

Spark provides two methods for users to control task parallelism. One method is to use a workload specific configuration parameter, $default.parallelism$, which serves as the default number of tasks to use for when the number of partitions is not specified. The second method is to use repartitioning APIs, which allow the users to repartition the data. Spark does not support changing of data parallelism between different computation phases except via manual partitioning within a user program through repartitioning APIs. This is problematic because the optimal number of partitions can be affected by the size of the data. Users would have to change and recompile the program every time they process a different data set. Thus, a clear opportunity for optimization is lost due to the rigid partitioning approach.

In this paper, we propose CHOPPER, an auto-partitioning system for Spark[1] that automatically determines the optimal number of partitions for each computation phase during workload execution. CHOPPER alleviates the need for users to manually tune their workloads to mitigate data skewness and suboptimal task parallelism. Our proposed dynamic data partitioning is challenging due to several reasons. First, since Spark does not support changing tasks parallelism parameters for each computation phase, CHOPPER would need to design a new interface to enable the envisioned dynamic tuning of task parallelism. Second, the optimal data partitions differ across different computation phases of workloads. CHOPPER needs to understand application characteristics that affect the task parallelism in order to select an appropriate partitioning strategy. Finally, to adjust the number of tasks, CHOPPER may introduce additional data repartitioning, which may incur extra data shuffling overhead that has to be mitigated or amortized. Thus, a careful orchestration of the parameters is needed to ensure that CHOPPER's benefits outweigh the costs.

To address the above challenges, CHOPPER first modifies Spark to support dynamically changing data partitions through an application specific configuration file. CHOPPER checks

[1]We use Spark as our evaluation DAG-based data processing framework to implement and showcase the effectiveness of CHOPPER. We note that the proposed system can be applied to other DAG-based data processing framework, such as Dryad [108].

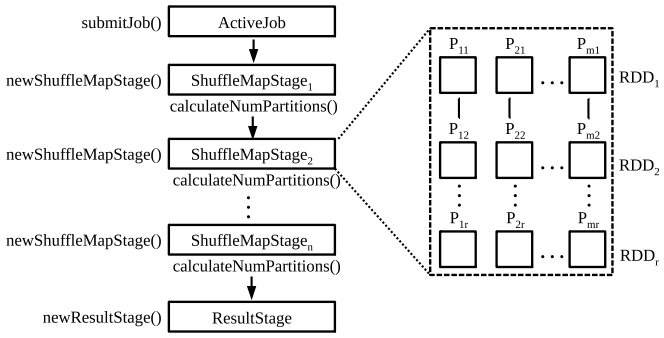


Figure 4.1: Overview of Spark DAGScheduler.

different numbers of data partitions before scheduling the next computation phase. Information gathering about the application execution is achieved via several lightweight test runs, which are then analyzed to identify task profiles, data skewness, and optimization opportunities. Chopper uses this information along with a heuristic to compute a data repartitioning scheme, which minimizes the data skew, determines the right tasks parallelism for each computation phase, while minimizing the repartitioning overhead.

Specifically, this paper makes the following contributions.

1. We enable dynamic tuning of task parallelism for each computation phase in DAG-based in-memory analytics platforms such as Spark.

2. We design a heuristic to carefully compute suitable data repartitioning schemes with low repartitioning overhead. Our approach successfully identifies the data skewness and optimization opportunities and adjusts task parallelism automatically to yield higher performance compared to the default static approach.

3. We implement Chopper on top of Spark and evaluate the system to demonstrate its effectiveness. Our experiments demonstrate that Chopper can significantly outperform vanilla Spark by up to 35.2% for the studied workloads.

4.2 Background and Motivation

In this section, we first discuss current data partitioning methodologies in Spark. Next, we present the motivation for our work by studying the performance impact of different number of data partitions on a representative workload, *KMeans* [101].

4.2.1 Spark Data Partitioning

In Spark, data is managed as an easy-to-use memory abstraction called resilient distributed datasets (RDDs) [219]. To process large data in parallel, Spark partitions an RDD into a collection of immutable partitions (*blocks*) across a set of machines. Each machine retains several blocks of an RDD. Spark tasks, with one-to-one relationship with the partitions, are launched on the machine that stores the partitions. Computation is done in the form of RDD actions and transformations, which can be used to capture the lineage of a dataset as a DAG of RDDs, and help in the recreation of an RDD in case of a failure. Such DAGs of RDDs are maintained in a specialized component, DAGScheduler, which schedules the tasks for execution.

Fig. 4.1 shows an overview of the Spark DAGScheduler. The input to DAGScheduler are called jobs (shown as ActiveJob in the figure). Jobs are submitted to the scheduler using a *submitJob* method. Every job requires computation of multiple stages to produce the final result. Stages are created by shuffle boundaries in the dependency graph, and constitute a set of tasks where each task is a single unit of work executed on a single machine. The narrow dependencies, e.g., *map* and *filter*, allow operations to be executed in parallel and are combined in a single stage. The wide dependencies, e.g., *reduceByKey*, require results to be combined from multiple tasks and cannot be confined to a single stage. Thus, there are two types of stages: ShuffleMapStage (shown in Fig. 4.1 as $ShuffleMapStage_1$, $ShuffleMapStage_2$, ..., $ShuffleMapStage_n$), which writes map output files for a shuffle operation, and ResultStage, i.e., the final stage in a job. ShuffleMapStage and ResultStage are created in the scheduler using *newShuffleMapStage* and *newResultStage* methods, respectively.

In Spark, tasks are generated based on the number of partitions of an input RDD at a particular stage. The same function is executed on every partition of an RDD by each task. In Fig. 4.1, $ShuffleMapStage_2$ is expanded to show the operations involved in a particular stage. Different operations in a stage form different RDDs (RDD_1, RDD_2, ... , RDD_r). Each RDD consists of a number of tasks that can be operated in parallel. In the figure, $P_{\alpha\beta}$ represents $partition_\alpha$ of RDD_β. There are m partitions for an RDD. Each RDD in a stage consists of a narrow dependency on the previous RDD, which enables parallel execution of multiple tasks. Thus, the number of partitions at each stage determins the level of parallelism.

Currently, Spark automatically determines the number of partitions based on the dataset size and the cluster setups. However, the number of partitions can also be configured manually using `spark.default.parallelism` parameter. In order to partition the data, Spark provides two data partitioning schemes, namely hash partitioner and range partitioner. Hash partitioner divides RDD based on the hash values of keys in each record. Data with the same hash keys are assigned to the same partition. Conversely, range partitioner divides a dataset into approximately equal-sized partitions, each of which contains data with keys within a specific range. Range partitioner provides better workload balance, while hash partitioner ensures that the related records are in the same partition. Hash partitioner is the default

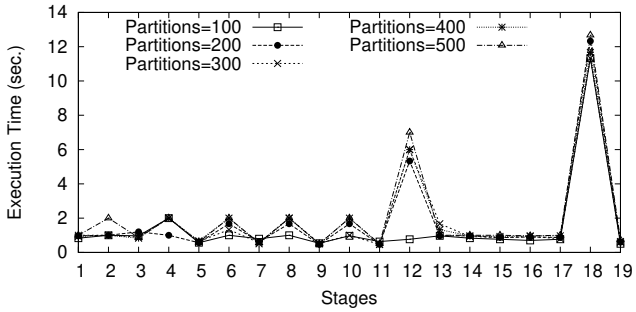


Figure 4.2: Execution time per stage under different number of partitions.

partitioner, however, users can opt to use their own partitioner by extending the *Partitioner* interface.

Although Spark provides mechanisms to automatically determine the number of partitions for a given RDD, it lacks application related knowledge to determine the best parallelism for a specific job. Moreover, the default hash partitioner is prone to creating workload imbalance for some input data. Spark provides the flexibility to tailor the configurations for workloads, however, it is not easy for users to determine the best configurations for each stage of a workload, especially when workloads may contain hundreds of stages.

4.2.2 Workload Study

To show the impact of data partitioning on application performance, we conduct a study using *KMeans* workload from SparkBench [129] and the latest release of Spark (version 1.6.1), with Hadoop (version 2.6) providing the HDFS [176] storage layer. Our experiments execute on a 6-node heterogeneous cluster. Three nodes (A, B, C) have 32 cores, 2.0 GHz AMD processors, 64 GB memory, and are connected through a 10 Gbps Ethernet interconnect. Two nodes (D, E) have 8 cores, 2.3 GHz Intel processors, 48 GB memory. The the remaining node (F) has 8 cores, 2.5 GHz Intel processor, and 64 GB memory. Nodes D, E and F are connected via a 1 Gbps Ethernet interconnect. Node F is configured to be the master node, while nodes A to E are worker nodes for both HDFS and Spark. Every worker node has one executor with 40 GB memory, and the remaining memory can be used for the OS buffer and HDFS data node operations. While our cluster hardware is heterogeneous, we configure each executor with the same amount of resources, essentially providing same resources to each component to better match with Spark's needs, and alleviating the performance impact due to the heterogeneity of the hardware. Note that, given the increasing heterogeneity in modern clusters, we do take the heterogeneity of cluster resource into account when designing Chopper. We repeated each experiment 3 times, and report the average results in the following.

First, we study the performance impact of different number of partitions. For this test, we

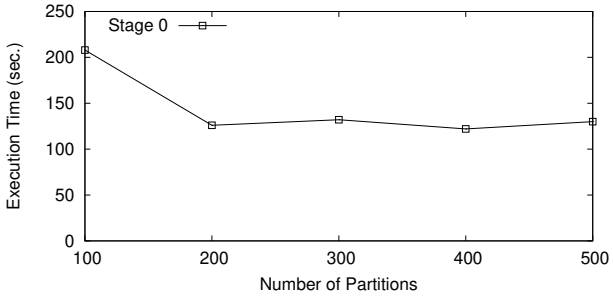


Figure 4.3: Execution time of stage 0 under different partition numbers.

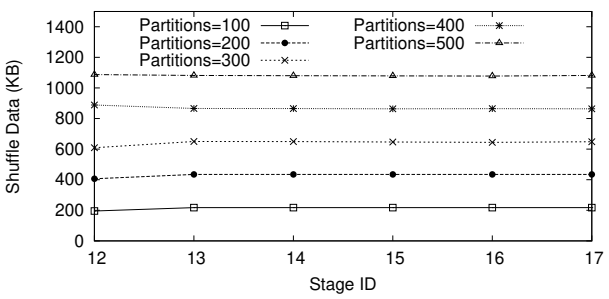


Figure 4.4: Shuffle data per stage under different partition numbers.

use KMeans workload with 7.3 GB input data size. KMeans has 20 stages in total, and we change the number of partitions from 100 to 500 and record the execution time for each stage. Fig. 4.2 shows the results. For every stage, the number of partitions that yields minimum execution time varies. This shows that different stages have different characteristics and that the execution time for each stage can vary even under the same configuration. To further investigate the impact on performance, we study stage-0 in more detail. As shown in Fig. 4.3, the execution time of a stage changes with the number of partitions, and we see the worst performance when the number of partitions is set to 100. From this study, we observed that the number of partitions has an impact on the overall performance of a workload. Furthermore, different stages inside a workload can have different optimal number of partitions. In this example, specifying 100 partitions may be an optimal configuration for overall execution (Fig. 4.2), but clearly it is not optimal for stage-0 (Fig. 4.3).

To better understand the above observed performance impact, we investigate the amount of shuffle data produced at each stage with different number of partitions—since shuffle has a big impact on workload performance. For this test, we record the maximum of shuffle read or write data as representative of shuffle data per stage. For KMeans, only stages 12-17 involve data shuffle. As shown in Fig. 4.4, any increase in the number of partitions also increases the shuffle data at each stage. A shuffle stage usually involves repartitioning RDD data. For an equivalent execution time, if repartitioning is not involved, then the amount of shuffle data increases from 434.83 KB for 200 partitions to 1081.6 KB for 500 partitions for stage-17, compared to 217.33 KB of shuffle data when repartitioning is done. Also, when compared to a large number of partitions, e.g., 2000, there is a significant increase in the execution time as well as increase in the amount of shuffle data. For 2000 partitions, the execution time is 4.53 minutes and the amount of shuffle data for stage-17 is 4300.8 KB. We observe 38.8% improvement in execution time from repartitioning for similar amount of shuffle data. Also there is 46.1% improvement in execution time, and 94.9% reduction in the amount of shuffle data per stage when compared to large (i.e., 2000) number of partitions.

These experiments show that the number of partitions is an important configuration parameter in Spark and can help improve the performance of a workload. The optimal number of

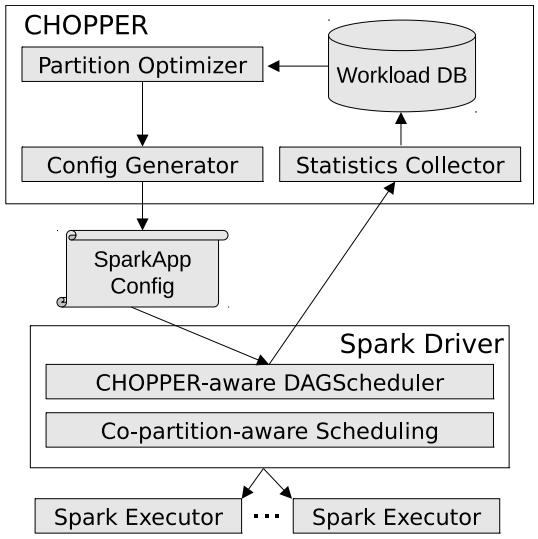


Figure 4.5: System architecture of Chopper.

partitions not only varies with workload characteristics, but is also different among different stages of a workload. We leverage these findings in designing the auto-partitioning scheme of Chopper.

4.3 System Design

In this section, we present the design of Chopper, and how it achieves automatic repartitioning of RDDs for improved performance and system efficiency.

Fig. 4.5 illustrates the overall architecture of Chopper. We design and implement Chopper as an independent component outside of Spark. As Spark is a fast evolving system, we keep the changes to Spark for enabling our dynamic partitioning to a minimum. This reduces code maintenance overhead while ensuring adoption of Chopper for real-world use cases. Chopper consists of a partition optimizer, a configuration generator, a statistics collector, and a workload database. In addition, Chopper extends the Spark's DAGScheduler to support dynamic partitioning configuration and employs a co-partition-aware scheduling mechanism to reduce network traffic whenever possible. *Statistics collector* communicates with Spark to gather runtime information and statistics of Spark applications. The collector can be easily extended to gather additional information as needed. *Workload DB* stores the observed information including the input and intermediate data size, the number of stages, the number of tasks per stage, and the resource utilization information. *Partition optimizer* retrieves application statistics, trains models, and computes an optimized partition scheme based on the current statistics and the trained models for the workload to be optimized. This information is also stored in *Workload DB* for future use. *Partition optimizer* then generates a workload

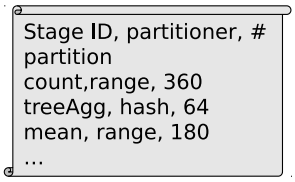


Figure 4.6: An example generated Spark workload configuration in CHOPPER.

specific configuration file. The extended dynamic partitioning DAGScheduler changes the number of partitions and the partition scheme per stage according to the generated Spark configuration file. Finally, the co-partitioning-aware component schedules partitions that are in the same key range on the same machine if possible to decrease the amount of shuffle data. The partition optimizer does not need to consider data locality because the input of the repartitioning phase is the local output of the previous Map phase, and the destinations of the output of the repartitioning phase depend on the designated shuffle scheme. Thus, existing locality is automatically preserved.

Our system allows dynamic updates to the Spark configuration file whenever more runtime information is obtained. CHOPPER modifies Spark to allow applications to recognize, read, and adopt the new partition scheme.

4.3.1 Enable Auto-Partitioning

An example application configuration file produced by CHOPPER is shown in Fig. 4.6. It consists of multiple tuples each containing a signature; the partitioner, and the number of partitions for a particular stage. We use stage signatures to identify stages that invoke identical transformations and actions. This is helpful when the number of iterations within a machine learning workload is unknown, where we can: (a) use the same partition scheme for all the iterations; or (b) use previously trained models to dynamically determine the number of partitions to use if the intermediate data size changes across iterations.

When an application is submitted to a Spark cluster, a Spark driver program is launched in which a SparkContext is instantiated. SparkContext then initiates our auto-partitioning aware DAGScheduler. The scheduler checks the Spark configuration file before a stage is executed. If the partition scheme is different from the current one, the scheduler changes the scheme based on the one specified in the configuration file. Each RDD has five internal properties, namely, partition list, function to compute for every partition's dependency list on other RDDs, partitioner, and a list of locations to compute every partition. CHOPPER changes the partitioner properties to enable repartitioning across stages.

CHOPPER also supports dynamic updates to the Spark application configuration file based on the runtime information of current running workload. In particular, DAGScheduler periodically checks the updated configuration file and uses the updated partitioning scheme if

Algorithm 2: Get Stage level Partition Scheme *getStagePar*.

Input: workload w, stage s, input size d
Output: ($partitioner$, $numPar$, $cost$)
begin
  $rModel$=getRangeParitionModel(w,s)
  $hModel$=getHashPartitionModel(w,s)
  ($numRangePar$,$rCost$)=getMinPar($rModel$,d)
  ($numHashPar$,$hCost$)=getMinPar($hModel$,d)
  if $rCost < hCost$ **then**
    return ($RangePartitioner$,$numRangePar$,$rCost$)
  end
  else
    return ($HashPartitioner$,$numHashPar$,$hCost$)
  end
end

available. This improves the partitioning efficiency and overall performance.

4.3.2 Determine Stage-Level Partition Scheme

The partition optimizer is responsible for computing a desirable partition scheme for each stage of a workload, given the collected workload history, and current input data and size. The optimizer not only considers the execution time and shuffle data size of a stage but also the shuffle dependencies between RDDs. In the following, we describe how this is achieved for the stage-level information. The use of global DAG information is discussed in Section 4.3.3.

Spark provides two types of partitioners, namely range partitioner and hash partitioner. Different data characteristics and data distributions require different partitioners to achieve optimal performance. Range partitioner creates data partitions with approximately same-sized ranges. RDD tuples that have keys in the same range are allocated to the same partition. Spark determines the ranges through sampling of the content of RDDs passed to it when creating a range partitioner. Thus, the effectiveness of a range partitioner highly depends on the data contents. A range partition scheme that distributes a RDD evenly is likely to partition another RDD into a highly-skewed distribution. In contrast, a hash partitioner allocates tuples using a hash function modulo of the number of partitions. The hash partitioner attempts to partition data evenly based on a hash function and is less sensitive to the data contents, and can produce even distributions. However, if the dataset has hot keys, a partition can become skewed in terms of load, as identical keys are mapped to the same partition. Consequently, the appropriate choice between the range partitioner or hash partitioner depends on the dataset characteristics and DAG execution patterns.

Algorithm 3: Get workload partition scheme *getWorkloadPar*.

Input: workload w, DAG dag, input size D
Output: $parList$
begin
  if $dag == null$ **then**
    dag = getDAG(w)
  end
  for *stage* s *in* dag **do**
    d=getStageInput(w, s, D)
    ($partitoner$, $numPar$, $cost$) = getStagePar(w, s, d)
    $parList$.add(s,$partitioner$,$numPar$,$cost$)
  end
  return parList
end

To compute the stage level partition scheme, we aim to minimize both the stage execution time and the amount of shuffle data. Considering stage execution time, and shuffle data that directly affects the execution time, enables us to capture the right task granularity. This prevents the partitions from both growing unexpectedly large—creating resource contention—or becoming trivially small—under-utilizing resources and incurring extra task scheduling overhead. The approach also implicitly alleviates task skew by filtering out inferior partition schemes.

Equations 4.1, 4.2, 4.3 and 4.4 describe the model learned and the objective function used to determine the optimal number of partitions. In particular, D denotes the size of input data for the stage, P denotes the number of partitions, t_{exe} represents the execution time of the stage, and $s_{shuffle}$ is the amount of shuffle data in the stage. t'_{exe} and $s'_{shuffle}$ denote the stage execution time and amount of shuffle data obtained using default parallelism, respectively. Given input data size, Equation 4.4 enables CHOPPER to determine the optimal number of partitions minimizing both execution time and the amount of shuffle data. By normalizing the execution time and the amount of shuffle data with respect to the respective values under default parallelism, we are able to capture both of our constraints into the same objective function. Constants α and β can be used to adjust the weights between the two factors. In our implementation, we set the constants to a default value of 0.5, making them equally important.

We model the execution time and the amount of shuffle data based on the input data size of the current stage and the number of partitions as shown in Equations 4.1 and 4.2. This is a coarse grained model, since it is independent of Spark execution details and focuses on capturing the relationship between input size, parallelism, execution time and shuffle data. In particular, we posit that the execution time increases with the input data size, obeying a combination of cube, square, linear, and sub-linear curves. The amount of shuffle data also

Algorithm 4: Get globally optimized partition scheme.

```
Input: workload w, input size D
Output: parList
begin
    dag=getReGroupedDAG(w)
    for node s in dag do
        d=getStageInput(w, s, D)
        if s isInstanceOf Stage then
            (partitoner, numPar, cost) = getStagePar(w, s, d)
        end
        else
            (partitoner, numPar, cost) = getSubGraphPar(w, s, d)
        end
        if s isFixed then
            curCost = getCost(w, s, getPartitioner(w, s), getNumPar(w, s))
            optCost = cost + getRepartitionCost(w, s, partioner, numPar)
            if optCost < curCost then
                s' = s + "repartitionstage"
                parList.add(s', partitioner, numPar, optCost)
            end
        end
        else
            parList.add(s, partitioner, numPar, cost)
        end
    end
    return parList
end
Function getSubGraphPar
    Input: workload w, DAG dag, input size D
    Output: paritioner, numPar, cost
    parList = getWorkloadPar(w, dag, D)
    min = parList(0)
    for s in parList do
        cost = getCost(w, dag, s.partitioner, s.numPar)
        if cost < min.cost then
            min = s
        end
    end
    return (min.partitioiner, min.numPar, min.cost)
Function getCost
    Input: workload w,DAG dag, partitioner, numPar
    Output: cost
    for stage s in dag do
        if partitioner == range then
            rModel = getRangePartitionModel(w, s)
            cost += Equation 4.3
        end
        else
            hModel = getHashParitionModel(w, s)
            cost += Equation 4.3
        end
    end
    return cost
```

increases or decreases with the number of partitions according to a combination of cube, square, linear, and sub-linear curves. Note that our model can capture most applications observed in the real-world use cases for Spark. However, it may not be able to model corner cases such as those with radically different behavior, e.g., workloads for which execution time grows with D^4. In general, we observe that such a model is simple and computationally efficient, yet powerful enough to capture applications with different characteristics via various coefficients of the model. When the cluster resources and other configuration parameters are fixed, the model fits the actual execution time and amount of shuffle data well with varying size of input data and number of partitions. The data points needed to train the models are gathered by the statistics collector. If the collected data points are not sufficient, CHOPPER can initiate a few test runs by varying the sampled input data size and the number of partitions and record the execution time and the amount of shuffle data produced. CHOPPER also remembers the statistics from the user workload execution in a production environment, which can be further leveraged to better train and model the current application behavior.

$$t_{exe} = a_1 D^3 + b_1 D^2 + c_1 D + d_1 D^{1/2} + e_1 P^3 + f_1 P^2 + g_1 P + h_1 P^{1/2}, \quad (4.1)$$

$$s_{shuffle} = a_2 D^3 + b_2 D^2 + c_2 D + d_2 D^{1/2} + e_2 P^3 + f_2 P^2 + g_2 P + h_2 P^{1/2}, \quad (4.2)$$

$$cost = \alpha t_{exe}/t'_{exe} + \beta s_{shuffle}/s'_{shuffle} \quad (4.3)$$

$$\min cost \quad (4.4)$$

CHOPPER trains two models using Equations 4.1 and 4.2 for every stage of a workload, one for range partitioning and the other for hash partitioning. Algorithm 2 presents how CHOPPER calculates the optimized stage level partition scheme given workload w, stage s and input size d for the stage. The algorithm returns the partitioner, the optimal number of partitions used for stage s and the cost. Specifically, CHOPPER retrieves the trained models of stages for both range partitioning and hash partitioning from the workload database. After this, Algorithm 2 computes the optimal numbers of partitions with minimal cost for both range partitioning and hash partitioning using Equation 4.4. Finally, CHOPPER returns the partitioner that would incur the lowest cost along with the number of partitions to use.

4.3.3 Globally-Optimized Partition Scheme

After we compute the stage level partition scheme, a naive solution is to compute the optimal partition scheme for each stage independently and generate the Spark configuration file. This is shown in Algorithm 3. It gets the DAG information from workload database, iterates

through the DAG, computes the desirable partition scheme for each stage, and adds to a list of partition scheme. Lastly, the algorithm returns the list of partition schemes, which is then used to generate the Spark configuration file for the current workload.

Although Algorithm 3 optimizes the partition scheme per stage, it misses the opportunities to reduce shuffle traffic because of the dependencies between stages and RDDs. For example, if stage-C joins the RDDs from stage-A and stage-B, the shuffle traffic introduced by join can be completely eliminated if the two use the same partition scheme and the joined partitions are allocated on the same machine. However, this cannot be achieved using Algorithm 3. If the computed optimal scheme of stage-A is $(Range, 100)$ and the optimal scheme of stage-B is $(hash, 200)$, the shuffle data cannot be eliminated, as the partition schemes of stage-A and stage-B are different. Even though stage-A and stage-B are optimally partitioned, the shuffle data of stage-C is sub-optimal. Since join and co-group operation are two of the most commonly used operations in Spark applications, poor partitioning will typically introduce significant shuffle overhead. Consequently, it is critical to optimize the join and co-group operation to decrease the amount of shuffle data as much as possible.

Another issue is that since the users are allowed to tune and specify customized partition scheme on their own, Chopper leaves the user optimization intact even when the computed optimal scheme disagrees with the user specified partition scheme. However, Chopper can choose to add an additional partition operation if the benefit of introducing the partition operation significantly outweighs the overhead incurred. For instance, consider a case where stage-B blows up the number of tasks to by a power of two from its previous stage-A (i.e., 100^2 tasks) due to the user-fixed partition scheme of stage-A. If Chopper coalesces the number of tasks of stage-A from 100 to 10, it would significantly reduce the number of tasks in stage-B from 10000 to 100.

To remedy this, Chopper determines the partition scheme by globally considering the entire DAG execution. As described in Algorithm 4, Chopper first groups the DAG graph based on the stage dependencies. The grouping of DAG graph is started from the end stages of the graph and iterated towards the source stages. The grouping is based on the join operations or partition dependencies. The stages with join operations are grouped into a subgraph. The partition dependencies refer to the cases where the number of stages is determined by the previous stage, thus Chopper cannot change the partition scheme. After the DAG of the workload is regrouped, the node within the new DAG can either be a stage or a subgraph that consists of multiple stages. If the node is a stage, the optimal partition scheme is computed using Algorithm 2. Otherwise, the node is a subgraph, where the optimal partition scheme is computed differently. Specifically, we iterate through all the nodes within the subgraph and get the optimal partition scheme for each node, we then compute the cost of applying each partition scheme to all the applicable nodes in the graph and return the partition scheme that has the minimal cost.

Finally, after we compute the globally optimal partition scheme, we check whether the stage partition is allowed to be changed. If not, and the partition scheme is different, we then

Workload	KMeans	PCA	SQL
Input Size (GB)	21.8	27.6	34.5

Table 4.1: Workloads and input data sizes.

check whether it is beneficial to insert a new repartitioning phase by comparing the cost using original partitioning to the cost of the new repartitioning phase together with the cost of optimized partition scheme. If the benefit outweighs the cost by a factor of γ, we choose to insert a new partition phase into the DAG graph. We empirically set γ to 1.5 to tolerate the model estimation error.

4.4 Evaluation

In this section, we evaluate CHOPPER and demonstrate its effectiveness on the cluster described earlier in Section 5.2. We use three representative workloads from SparkBench: KMeans, PCA, and SQL. KMeans [101] is a popular clustering algorithm that partitions and clusters n data points into k clusters in which each data point is assigned to the nearest center point. The computation requirement of this workload change according to the number of clusters, the number of data points, and the machine learning workload that exhibits different resource utilization demand for different stages during the process of iteratively calculating k clusters. PCA [111] is a commonly used technique to reduce the number of features in various data mining algorithms such as SVM [80] and logistic regression [105]. It is both computation and network-intensive machine learning workload that involves multiple iterations to compute a linearly uncorrelated set of vectors from a set of possibly correlated ones. SQL is a workload that performs typical query operations that count, aggregate, and join the data sets. Thus, SQL represents a common real world scenario. SQL is compute intensive for count and aggregation operations and shuffle intensive in the join phase. The input data is generated by the corresponding data generator within SparkBench. The input data size for each workload is shown in Table 4.1. The experiments for vanilla Spark are conducted with the default configuration, which is set to 300 partitions for all the workloads. We run all of our experiments three times, and the numbers reported here are from the average of these runs. Moreoer, we clear the OS cache between runs to preclude the impact of such caching on observed times.

4.4.1 Overall Performance of Chopper

Our first test evaluates the overall performance impact of CHOPPER. Fig. 4.7 illustrates the total execution time of three workloads comparing CHOPPER against standard vanilla Spark. The reported execution time includes the overhead of repartitioning introduced by CHOPPER. We can see that CHOPPER achieves overall improvement in the execution time

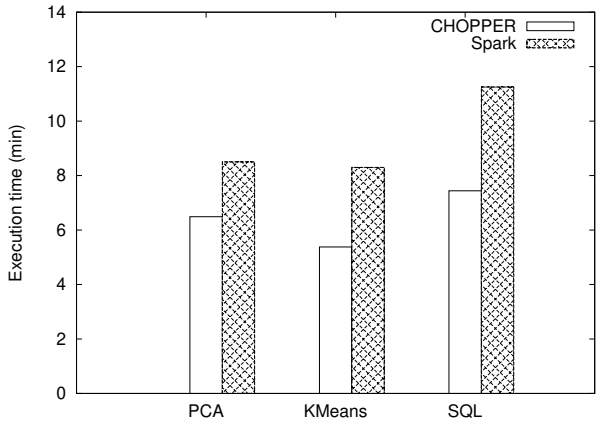


Figure 4.7: Execution time of Spark and Chopper.

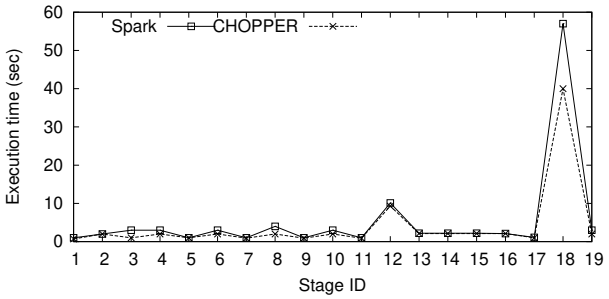


Figure 4.8: Execution time per stage breakdown of KMeans.

Table 4.2: Execution time for stage 0 in KMeans.

	Chopper	**Spark**
Execution Time (sec)	250	372

by 23.6%, 35.2% and 33.9% for PCA, KMeans and SQL, respectively. This is because Chopper effectively detects optimal partitioner and the number of partitions for all stages within each workload. Chopper also performs global optimization to further reduce network traffic by intelligently co-partitioning dependent RDDs and inserting repartition operations when the benefits outweigh the cost. We also observe that Chopper is effective for all types of workloads that exhibit different resource utilization characteristics. The repartition method of Chopper implicitly reduces the potential data skew and determines the right task granularity for each workload, thus improving the cluster resource utilization and workload performance. Thus, Chopper shows significant reduction in the overall execution time for all the three workloads. The model training of Chopper is conducted offline, and thus is not in the critical path of workload execution. Moreover, the overhead of repartitioning is negligible as it involves solving a simple linear programming problem.

4.4.2 Timing Breakdown of Execution Stages

To better understand how dynamic partitioning of Chopper helps to improve overall performance, in our next test, we examine the detailed timing breakdown of individual workload stages. Fig. 4.8 depicts the execution time per stage for KMeans. We show the execution time of stage-0 separately in Table 4.2 since the execution time of stage-0 and that of other stages differs significantly. We see that Chopper reduces execution time of each stage for KMeans compared to vanilla Spark, as Chopper is able to customize partition schemes

Table 4.3: Repartition of stages using CHOPPER.

StageID	0	1	2	3	4	5	6	7	8	9	10	11	12 - 17	18	19
Chopper	210	210	300	720	300	720	300	720	300	720	300	720	210	380	210
Spark	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300

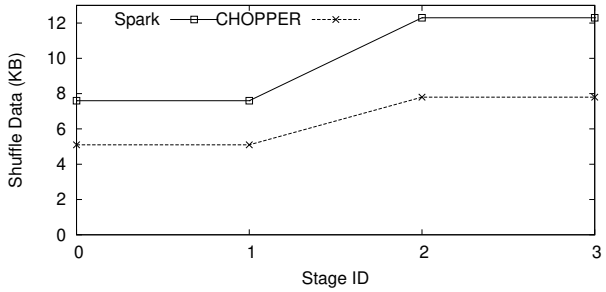


Figure 4.9: Shuffle data per stage for SQL.

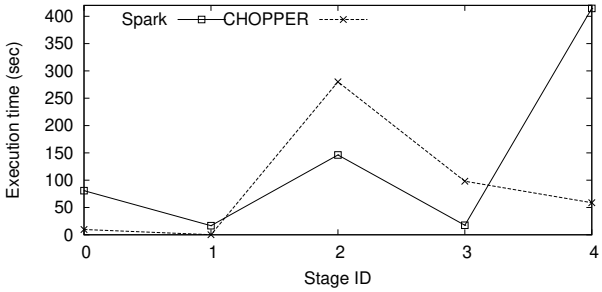


Figure 4.10: Execution time per stage breakdown of SQL.

for each stage according to associated history and runtime characteristics. Table 4.3 shows the number of partitions used by CHOPPER for different stages compared to vanilla Spark. Stages 12 to 17 are iterative, and thus are assigned the same number of partitions. We see that CHOPPER effectively detects and changes to the correct number of partitions for this workload rather than using a fixed (default) value throughout the execution.

4.4.3 Impact on Shuffle Stages

In our next test, we use a shuffle-intensive workload, SQL, to study how CHOPPER reduces the shuffle traffic by automatically recognizing and co-partitioning dependent RDDs. Fig. 4.9 shows that the shuffle data for all four stages is less under CHOPPER compared to vanilla Spark. Stage-4 (not shown in the figure) has the same amount of shuffle data for SQL workload using CHOPPER or Spark (i.e., 4.7 GB). However, as seen in Fig. 4.10, stage-4 takes comparatively shorter time to execute using CHOPPER versus Spark. This is because, stage 4 is divided into 4 sub-stages where the first two sub-stages have a shuffle write data of 1.9 GB and 2.8 GB. CHOPPER combines these two sub-stages for shuffle write and provides the third sub-stage for the shuffle data to be read. This greatly reduces the execution time for stage 4 as seen in Fig. 4.10. Thus, we demonstrate that CHOPPER can effectively detect dependent RDDs and co-partition them to reduce the shuffle traffic and improve the overall workload performance.

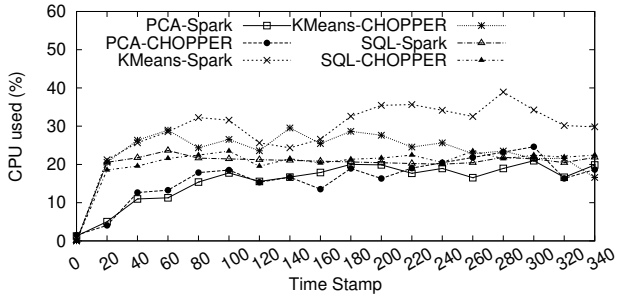


Figure 4.11: CPU utilization.

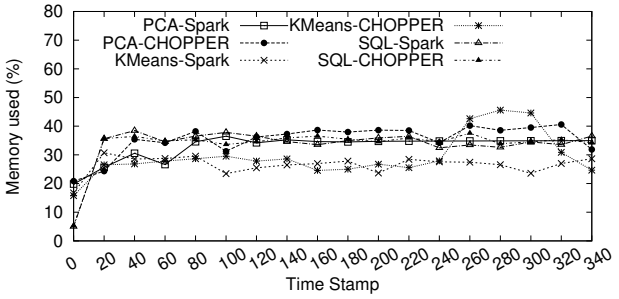


Figure 4.12: Memory utilization.

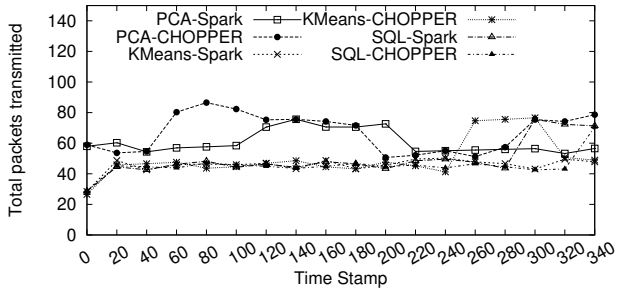


Figure 4.13: Total transmitted and received packets per second.

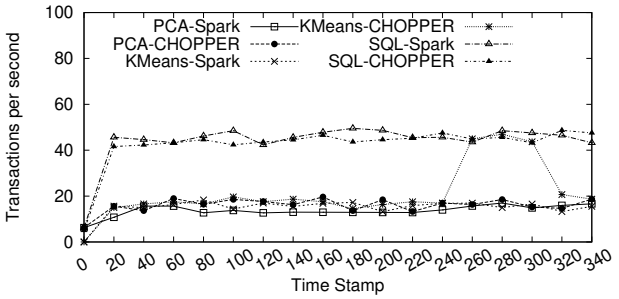


Figure 4.14: Transactions per second.

4.4.4 Impact on System Utilization

In our next experiment, we investigate how Chopper impacts the resource utilization of all the studied workloads. Fig. 4.11, 4.12, 4.13 and 4.14 depict the CPU utilization, memory utilization, total number of transmitted and received packets per second, and the total number of read and write transactions per second, respectively, during the execution of the workloads under Chopper and vanilla Spark. The numbers show the average of the statistics collected from the six nodes in our cluster setup. We observe that the performance of Chopper is either equivalent or in most of the cases better than the performance of vanilla Spark for the studied workloads. In some cases, Chopper shows improved transactions per seconds as compared to vanilla Spark because of the high throughput and improved system performance.

These experiments show that the performance (computed on the basis of execution time and shuffle data) improves under Chopper compared to vanilla Spark. Also, these improvements in Chopper yield comparable or better system utilization compared to vanilla Spark.

4.5 Chapter Summary

In this paper, we design Chopper, a dynamic partitioning approach for in-memory data analytic platforms. Chopper determines the optimal number of partitions and the partitioner for each stage of a running workload with the goal of minimizing the stage execution time and shuffle traffic. Chopper also considers the dependencies between stages, including join and cogroup operations, to further reduce shuffle traffic. By minimizing the stage execution time and shuffle traffic, Chopper implicitly alleviates the task data skew using different partitioners and improves the task resource utilization through optimal number of partitions. Experimental results demonstrate that Chopper effectively improves overall performance by up to 35.2% for representative workloads compared to standard vanilla Spark.

Our current implementation of Chopper has to re-train its models whenever the available resources are changed. In future, we plan to explore the per-stage performance models that can work across different resource configurations, i.e., clusters. We will also explore how Chopper behaves under failures. These will further improve the applicability of Chopper in a cloud environment, where compute resources are failure-prone and scaled as needed.

Chapter 5

Scaling Up Data-Parallel Analytics Platforms

5.1 Introduction

High-dimensional data is commonplace both in science [63, 191] and enterprise [166] applications. In order to discover patterns and relationships in such data, machine learning (ML) is widely used, typically through a core set of linear algebraic operations, called the analysis kernel [76]. The fundamental need of scalable data analysis, thus, is the ability to process a large number of data samples timely, i.e., to have a high analysis kernel computing throughput for matrix operations.

Extant practices for fast analysis kernel computing fall into two broad groups: (1) enabling scaling out on commodity hardware via the use of data parallel computation software platforms such as MPI [162], Hadoop[114], and Apache Spark [85, 135, 200]; and (2) enabling scaling up the computational power of individual nodes/machines for data analysis [71, 79]. Scaling up approaches have been proven to be beneficial to data analysis algorithms, since the approaches entail optimization algorithms [76] that can exploit hardware accelerators, e.g., GPUs, ASICs, FPGAs, and specialized CPU instructions [104] for matrix operations [123, 144, 145].

However, advantages from both scale-out and scale-up techniques come at a price to achieve higher performance. Scale-out options can suffer communication networking overheads and higher costs (energy, labor, machine failures, etc.) as the size of a cluster grows. Due to the constantly growing data acquisition throughput, such overheads lead to a constant pressure on scale-out options. On the other hand, scale-up options face a hard-wall of scalability, limited by the resources available in a single machine. Hence, *how best to combine and reconcile the scale-out and scale-up approaches* has become an important topic in processing matrix operations at scale.

Although it is promising to combining scale-out and scale-up techniques, it is non-trivial to realize the full potential of both approaches, due to the fundamental design choices made under each approach. Scale-out approaches, such as Spark, often focus on scalability, fault tolerance, cluster utilization, and workload balancing, but do not typically factor in techniques for improving an individual node's performance via hardware acceleration to support a wide range of distributed environments. For instance, Spark maps a job to a number of tasks that can later be run on any participating nodes without any regard to the unique hardware capabilities of the nodes or the ability of a task to leverage the available capabilities.

Let us look into the details of the challenges in exploiting node-local accelerators for matrix operations in the context of Spark. Spark MLlib [140], a widely used machine learning library built on top of Spark, often applies an ad-hoc Scala implementation for performing matrix operations without regards to node-local accelerators. Thus, users have to understand and implement system-level details if they want to scale-up Spark tasks. Moreover, scale-up approaches are not designed to consider the wide variance of tasks within Spark applications.

Scale-up solutions focus on parallelism at a fine granularity, such as SIMD and SIMT models, but with limited consideration of distributed environments. Each scale-up solutions for matrix operations, also known as Basic Linear Algebra Subroutine (BLAS) [148] operations, typically target specialized operations for the specific target hardware acceleration, with minimal consideration, if any, on distributing tasks across systems. For example, OpenBLAS [224] is optimized for BLAS operations for dense matrices on CPU; cuBLAS [151] is optimized for BLAS operations on GPUs; and spBLAS [84] is optimized for BLAS operations between a sparse matrix and a dense matrix. It is because the maximal performance for each BLAS operation under each hardware configuration can significantly vary according to the density and size of given matrices. Hence, it is challenging to integrate existing scale-up solutions to the Map-Reduce programming paradigm, where the same operation is to be applied over each partition (or element) in the data set, through a single run-time solution when different tasks form a directed acyclic graph.

Therefore, we propose ARION, a dynamic hardware acceleration framework atop Spark. ARION is carefully designed for scaling up BLAS operations in a MapReduce like scale-out environment where multiple map tasks are concurrently running. ARION supports multiple hardware accelerated solutions for BLAS operations on both sparse and dense matrices. In addition, ARION treats available GPUs as additional cluster slots to run tasks. Thus, we allow to run tasks on both GPUs and CPUs on the condition that the overheads to schedule tasks on each processing unit does not exceed the benefits. The ability to leverage multiple hardware resources increases the overall system utilization. ARION bridges the mismatch of the design between local accelerators and scale-out platforms by utilizing techniques such as stream processing and kernel batching. More specifically:

- We extend our previous work [206] by designing and implementing ARION, a dynamic hardware acceleration framework atop Spark, and effectively combine scale-up hardware acceleration for distributed linear algebraic operations with scale-out capabilities

of Spark.

- We design distributed matrix manipulation support for both dense and sparse matrices in Spark for scale-out matrix operations. In contrast, the current state of the art, e.g., MLlib [140], considers only sparse matrices in a distributed setting.
- We design a dynamic algorithm that chooses the best hardware accelerator at runtime by considering key factors such as matrix size, density, and system utilization. Our approach does not require changes to the Spark applications on user side, instead it leverages already-available libraries to support the target operations.
- We design a dedicated GPU component along with the BLAS libraries to support concurrent usage of both CPU and GPU to increase overall system utilization. The GPU component is optimized to maximize the GPU utilization by streaming processing and kernel batching techniques.

On top of our design and implementation, for computing Gram matrix (XX^T [214]), Arion shows more than 2× and 1.5× speed-up for end-to-end performance for dense and sparse matrices, respectively. Most notably, for dense matrices, Arion achieves 57.04× of speed-up in the computation time. Finally, Arion performs faster than or comparable to a larger scale setup with default Spark.

5.2 Background

In abstract, most of machine learning algorithms can be understood as the operations between a weight matrix that represents the training model and feature vectors that represents the data samples. Thus, linear algebra operations forms the foundation of machine learning algorithms, making MLlib, the machine learning package of Spark, the default package for distributed linear algebra operations in Spark. MLlib wraps the lower level Spark Resilient Distributed Dataset (RDD) generation and operations needed to perform matrix operations and machine learning algorithms. A common assumption across components in MLlib is that a matrix will be mostly likely to be sparse, tall, and thin [215]. Assuming each row vector of a matrix will fit into a reasonable size of machines, most of linear algebra operations in MLlib does not support fully distributed linear algebra operations, along with limited support for dense matrix operations, except *BlockMatrix*.

Such a design choice leads to challenges in accelerating linear algebra operations in Spark. Assuming sparse matrix, MLlib heavily exploits sparse matrix representation to compactly store non-zero terms for the sake of efficiency. A consensus on accelerating BLAS operations is that GPUs are typically less efficient in performing sparse matrix operations than dense matrix operations [123]. Thus, in most cases, to support sparse operations, a user inflates the sparse matrices into a dense one and offloads the processing to the accelerators. Otherwise,

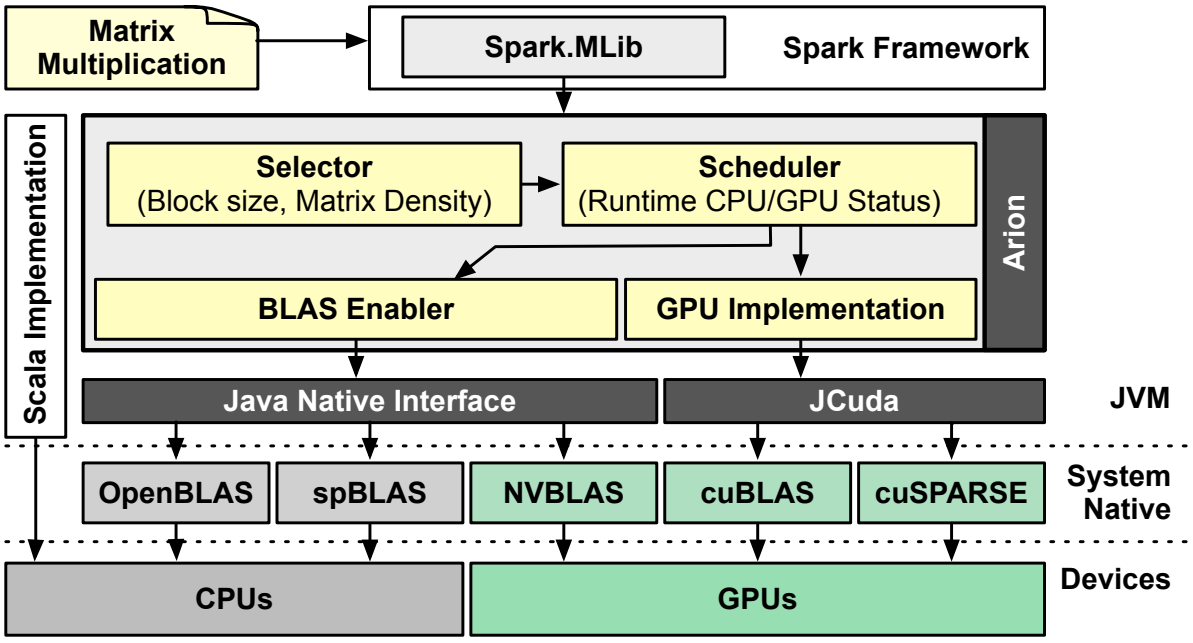


Figure 5.1: System architecture of ARION.

accelerator inflates internally. In addition to data representation, Spark partitions the matrix into concurrent multiple tasks per node, assigning in a relatively smaller chunk to each task (e.g., default partition size for *BlockMatrix* is $1K \times 1K$). The execution of each task often forms multiple waves, resulting in a limited number of concurrent tasks per node. As a result, a small number of concurrent matrix operations arrive at GPU at the unit of small chunks, leading to a low utilization of GPU. Accordingly, a straightforward drop-in solution of using BLAS libraries from Spark MLlib, including *BlockMatrix* class, cannot guarantee a full utilization of hardware capabilities.

5.3 Design

This section describes the design overview of ARION, along with detailed explanation on our design choices.

5.3.1 System Architecture

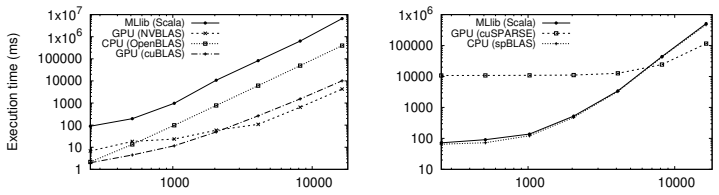
Figure 5.1 illustrates the architecture of ARION and its key components that all run in Spark executors on participating worker nodes. **Selector** is responsible for selecting the optimal

processing variant such as BLAS and GPU implementation (**GPU Impl**) based on matrix characteristics. **Selector** addresses the problem of the standard MLlib always using the Scala implementation for multiplications in *BlockMatrix*. We employ MLlib to partition the input datasets per given block size, and generate RDDs and associated tasks, which are then send to the executor. **Selector** calculates the density of assigned sub-matrices (this also catches skewed density cases that can be missed if examining a matrix as a whole), retrieves the block size information, and decides which processing variant to use to perform the multiplication. The tasks are then submitted to the **Scheduler**.

Our **Scheduler** aims for high resource utilization of both CPUs and GPUs. Thus, tasks are scheduled based on the system runtime utilization using the processing variant chosen by the **Selector** with a high priority. **Scheduler** works with node-local hardware accelerators, which can help the case of a heterogeneous cluster where each node has different hardware configurations. Moreover, to increase resource utilization, **Scheduler** negotiates with Spark task scheduling system to dynamically allocate more tasks if hardware resources become idle. This is in contrast to the current Spark process, which only considers CPU core resource.

5.3.2 Hardware Acceleration for Dense and Spare Matrices

Table 5.1 lists the processing variants supported in Arion. The ad-hoc Scala implementation is preserved in our system as one of the options. For others, we proceed as follows. Given the block size, the matrices are partitioned into block sub-matrices. Each task performs the multiplication of two block sub-matrices. Arion supports multiplications of both ddGEMM and spGEMM. As shown in Figure 5.1, Arion enables hardware acceleration via two channels: **BLAS Enabler** and **GPU Impl**. **BLAS Enabler** is able to adopt underlying hardware optimized BLAS libraries through a Java native interface (JNI). Currently we adopt OpenBLAS [224], NVBLAS [152], and spBLAS [84]. Here, OpenBLAS and NVBLAS support acceleration for ddGEMM on CPU and GPU, respectively. Both are allowed in MLlib with the integrated Netlib-Java [149] as the JNI layer. To support hardware accelerated sdGEMM, we adopt the spBLAS library under **BLAS Enabler**. For this purpose, we extend Netlib-Java to support linking to the spBLAS library so that it is transparent to MLlib. Note that our design of **BLAS Enabler** is flexible and support plugging in of any external libraries via JNI. While detailed discussion is out of the scope of this paper, we do not prevent the adoption of SpGEMM (sparse-sparse matrix multiplication) libraries without hardware acceleration. We also implement a dedicated GPU matrix multiplication component, **GPU Impl**, with GPU runtime dense matrix operation libraries such as cuBLAS [151] and spark matrix operation libraries such as cuSPARSE, which bypasses initializing BLAS libraries through JNI. Thus, Arion supports co-execution using both CPU and GPU, i.e., using **GPU Impl** without JNI BLAS library linking, while using CPU BLAS libraries through JNI. This can yield more effective scale-up by leveraging both CPU and GPU accelerator resources.



(a) Dense-dense matrix. (b) Sparse-dense matrix.

Figure 5.2: Execution time of matrix multiplication using different implementations. X-axis represents b in a block matrix $b \times b$. Both axes are log-scaled.

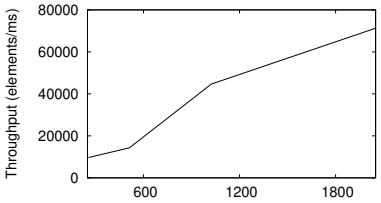


Figure 5.3: Throughput of NVBLAS. X-axis represents b in a block matrix $b \times b$.

5.3.3 Choice of Processing Variants

Once **Selector** determines the matrix type of a given multiplication task, it recommends the methods to **Scheduler**, which are then given priority for processing the matrix. Note that the **Scheduler** can also ignore the recommendation if it conflicts with resource availability and utilization. To suggest the right option for performing multiplication on a given environment, we adopt a user configurable system parameter Th_{block} (size of a block matrix) to decide the best compute variants, given environments and workloads. We choose block size as a parameter since other parameters such as density can skew in a matrix, and with a fixed block size, we can calculate the density of each block. Note that the Th_{block} is for both dense and sparse matrix.

Let us show our approach to decide Th_{block} used in our evaluation, which also can highlight the differences between the three supported methods. We first performed ddGEMM with both NVBLAS and **GPU Impl** using cuBLAS as the two methods for GPU acceleration, and OpenBLAS as CPU acceleration on a GPU node in *Rhea* [154] (Table 5.2 shows the specifications). For simplicity, we measured the end-to-end computation time for the multiplication of two matrices of the same size. In this measurement, we included the data transfer time between the host and GPU into the computation time since this time is included in the computation time of tasks in Spark. We used block sizes within the range of 256×256 to $16K \times 16K$ in the experiment, which is within the reasonable range for Spark. The default block size in Spark is $1K \times 1K$. Figure 5.2(a) shows the execution time using different options with different block sizes. The reported execution time is an average of five runs for each block size. For small block sizes up to $4K \times 4K$, we observe that **GPU Impl** performs better than the other two. As the block size increases, NVBLAS starts to outperform **GPU Impl** due to NVBLAS design (subsection 5.3.4). OpenBLAS performs reasonably within the block size of $1K \times 1K$, but the performance degrades dramatically as the size grows more than $2K \times 2K$, when OpenBLAS starts to perform worse than NVBLAS. Based on this observation, we identify Th_{block} for dense matrix in our system to be $2K \times 2K$ (the cross point of the NVBLAS and **GPU Impl**). In our system, **Selector** would either choose NVBLAS or **GPU Impl** based on the threshold as they outperform OpenBLAS in any cases.

Table 5.1: Summary of all processing variants supported in Arion. Methods on "GPU (direct)" platform can co-execute with methods running on "CPU" platform.

Methods	Matrix type	Platform
OpenBLAS	Dense-dense	CPU
NVBLAS	Dense-dense	GPU
cuBLAS (GPU Impl)	Dense-dense	GPU (direct)
spBLAS	Sparse-dense	CPU
cuSPARSE (GPU Impl)	Sparse-dense	GPU (direct)

Next, we repeat the experiment with a dense matrix and a sparse matrix with density of 0.05. We recorded the end-to-end time of multiplication with **GPU Impl** using cuSPARSE, and CPU acceleration using spBLAS. As shown in Figure 5.2(b), GPU does not accelerate the calculation until the block size reaches $8K \times 8K$. Thus, we chose the Th_{block} to be $8K \times 8K$ for sparse-dense matrix multiplications. The thresholds are configurable by the users, which they can set based on their specific environment profiling. The values remain unchanged if the environment does not change. The default in our system is set based on our experiments above. Moreover, we found in our experiment that there is an upper limit size for a single GPU bounded by the GPU memory size, $32K \times 32K$ in our case, thus Th_{block} should be below this limit. In practice, a user's environment may have multiple decision points, for which we use a configuration file. The file is used by the **Selector** after it has determined the computation type (ddGEMM or sdGEMM) to select appropriate processing variants.

Discussion: To show the impact of hardware acceleration, we also study the performance of the Scala implementation of MLlib as our baseline case for our studied scenarios. We can see that for ddGEMM, hardware acceleration significantly speeds up the execution time. However, spBLAS performs slightly better or similar to MLlib. This is because ddGEMM is more computation intensive than multiplication with highly sparse matrices, thus ddGEMM offers more opportunity for performance improvement.

5.3.4 Improving System Utilization

Our design aims to improve resource utilization with the goal to maximize performance and efficiency. We achieve this via a simple, yet effective, scheduling algorithm. If a GPU has to be employed, we also prevent node-level waves of tasks. This is because the number of tasks assigned per node can be much larger than the concurrency limit of the available GPU. We propose to batch task executions to improve the concurrency efficiency of GPUs.

Task Scheduling

The goal of node-level task scheduling is to increase resource utilization whenever possible. First, **Scheduler** over-provisions the task slots and enables tasks to run on both GPUs and CPUs simultaneously, instead of just utilizing one type of resource at a time. Note

that the data parallel design of Spark guarantees the data independency among Spark tasks so there is no data communication between CPU tasks and GPU tasks. A side effect of this approach can be that tasks are scheduled on suboptimal resources. To remedy such problems, **Scheduler** speculatively executes lagging tasks on available resources.

To over-provision the resources, we assign the task slots n_{slots} with the number of CPU cores plus the GPU capacity, i.e., the maximum number of block sub-matrices the GPUs attached to a node can hold. For example, in our experimental setup described in subsection 5.3.3, n_{slots} = 56 (CPU cores)+32 (GPU streams [151] of 2 GPUs per node). Among the n_{slots} task slots, 56 are for the CPUs and the others for the GPUs. This allows ARION to request more tasks per node from Spark DAG scheduler to execute concurrently.

The scheduling algorithm works as follows. Given the processing option, and the occupied GPU and CPU slots, **Scheduler** obtains the resource type used by the processing variant as determined by **Selector**. If the resource type is GPU and all the GPUs are currently occupied, **Scheduler** changes the current processing variant to the method for the same matrix type in CPU platform shown in Table 5.1 to use the available CPU slots, i. e., OpenBLAS for ddGEMM and spBLAS for sdGEMM. On the other hand, if the resource type is CPU and all CPU slots are occupied, then depending on the block size, **Scheduler** changes the processing option to NVBLAS or to cuBLAS for ddGEMM. **Scheduler** simply changes to cuSPARSE for spGEMM. Here, to support running tasks on both OpenBLAS and NVBLAS simultaneously, we preload and link the NVBLAS library as the BLAS library, and automatically modify NVBLAS configuration to redirect the BLAS operation to OpenBLAS. Similarly, we modify the configuration back to NVBLAS when necessary. Note that this configuration switch requires each thread to maintain a thread local configuration file to avoid conflicts. In practice, the block size is usually small due to the limitations mentioned in subsection 5.3.3. As a result, ARION opts for cuBLAS over NVBLAS, and thus is able to avoid this switching and initiation overhead. Furthermore, recall that we extended Netlib-Java to support spBLAS (subsection 5.3.2). Our extension does not interfere with the existing OpenBLAS and NVBLAS libraries, instead we internally load spBLAS library from our extended Java code without the need of preloading as long as the spBLAS library can be loaded in the system library path.

Scheduler respects the variant choice of **Selector** as long as appropriate resources are available, choosing an alternate only if needed resources are occupied. However, in cases with few number of waves of tasks, executing on the suboptimal alternate can lead to suboptimal task execution time, which in turn exacerbates the performance of entire stage execution. For instance, for an application that launches 88 tasks per node as a single wave requesting cuBLAS, **Scheduler** may change the processing option of 56 tasks to OpenBLAS and executes them on CPUs. Since GPUs run much faster for dense matrices, the other tasks executed on CPUs become stragglers. Thus, **Scheduler** speculatively re-launches the tasks on idle GPUs and terminates the stage when all the results are available. Consequently, ARION is able to mitigate any suboptimal **Selector** decisions.

GPU Optimization

Arion adopts a number of optimization techniques for **GPU Impl** to ensure that all of the GPUs within a node are fully utilized.

We first discuss the performance inefficiency of the tiling technique of NVBLAS. NVBLAS uses the cuBLAS-XT [151] API under the hood. cuBLAS-XT supports a multi-GPU enabled host interface. In cuBLAS-XT, matrices are divided into configurable square tiles of a fixed dimension, called *BlockDim*, with a default value of $2K$. Each tile is assigned to an attached GPU device in a round-robin fashion, and one CPU thread is associated with one GPU device for transferring data. This enables NVBLAS to pipeline tile transfer and computation, thus improving the overall GPU throughput.

While NVBLAS outperforms other processing variants by tile splitting when matrices are large, Spark tasks work on RDD partitions where each sub-matrix is usually not big enough to be further split into square tiles of size $2K$. Usually GPU throughput is low with small sizes of matrices. To verify this, we tested GPU throughput of matrix multiplication with different sizes. As shown in Figure 5.3, GPU throughput drops exponentially as the size of matrix decreases.

Another problem is that NVBLAS handles one matrix multiplication at a time, with multiple tiles of the matrices handled concurrently. This concurrency model is different from Spark concurrency model in which each task performs one sub-matrix multiplication, and multiple sub-matrix multiplications are performed by different tasks at the same time. It is equally important to optimize the performance of using GPU for small matrices. However, NVBLAS is inefficient in achieving high concurrency for small matrices multiplications. Thus, we propose to batch small matrices into a large one to improve the concurrency and utilization of GPUs in our **GPU Impl** design (for both cuBLAS instance and cuSPARSE instance). In particular, we adopt the CUDA Streams [151] technology to overlap computation of different tasks by batching the execution of small kernels. We associate one Spark task with one separate CUDA stream to make the GPU compute the tasks concurrently. This effectively batch multiple small matrices into a large matrix to run concurrently, consequently improving the GPUs' throughput.

While it is possible to have as many streams as needed for a single GPU, the CUDA community suggests to not have more than 16 concurrent kernels per GPU [151]. We further limit the degree of concurrency based on the GPU memory capacity, such that a GPU can hold no more than $GPU_{ram}/b^2 bytes/3$ streams concurrently, where GPU_{ram} represents the GPU memory capacity. This is to accommodate the two $b \times b$ input matrices and the result matrix. Although sparse matrices can be represented with compressed formats, we keep this limitation without loss of generality (e.g., GPU is used for a mixture of multiplication tasks including ddGEMM and sdGEMM). Using multiple streams may cause higher data transfer overhead between the CPU and GPUs. In this case, we arrange a fixed number of concurrent streams for each GPU, and once the streams in one GPU are all occupied, we

Table 5.2: System specification.

System name	Rhea GPU node	Rhea CPU node
CPU model	dual Xeon E5-2695	dual Xeon E5-2650
CPU cores	14×2 (28×2 HT)	16×2 (32×2 HT)
CPU memory	1TB	128GB
GPU model	dual NVIDIA K80	N/A
GPU (CUDA) cores	4992×2	N/A
GPU memory	24×2 GB	N/A
CUDA ver.	7.5	N/A
Network Interface	1G Ethernet	1G Ethernet

Table 5.3: Studied matrix sizes, densities, and raw file size.

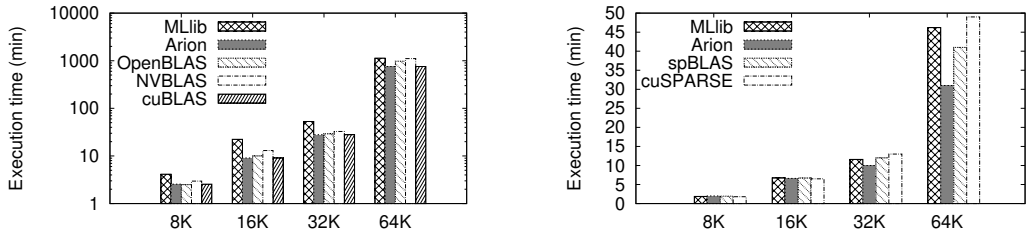
Matrix	8K	16K	32K	64K
Density=1.00	0.98 GB	3.93 GB	15.73 GB	80.82 GB
Density=0.05	0.30 GB	1.19 GB	4.77 GB	19.08 GB

dispatch the next batch tasks to the next GPU in a round robin fashion. Therefore, we can efficiently utilize multiple GPUs to overlap the data transfer and accelerate Spark tasks for small matrices with high concurrency.

5.4 Evaluation

We have integrated ARION in Spark by modifying MLlib (mainly small changes to *BlockMatrix* class). However, ARION can also be integrated with other scale-out platforms. In this section, we evaluate ARION's ability to efficiently accelerate matrix multiplication in Spark compared to vanilla MLlib implementation. To this end, we use a large scale Gramian matrix ($\mathbf{XX^T}$) computation kernel. This kernel is common in ML algorithms such as SVD and PCA [76]. Gramian matrix computation also plays a critical role in popular data analysis techniques, e.g., all-pair similarity [214].

We use Spark 1.6.1, the latest version at the time of this work. We manually compiled OpenBLAS and spBLAS library version 0.2.19 with acceleration instruction flags set including AVX2 and FMA3 [116]. We use the default NVBLAS library included in CUDA toolkit 7.5. We implemented **GPU Impl** with cuBLAS and cuSPARSE APIs from the CUDA toolkit. Our experiments are conducted on six highly-scalable GPU nodes assigned from a super computing cluster, *Rhea*, as described in Table 5.2. We configure Spark with one master and six worker nodes. One worker is co-located with the master. Each worker node runs one executor. We configure each executor to have 800 *GB* memory and 56 cores. We repeat each experiment five times and report the average results.



(a) Dense matrix. (b) Sparse matrix.

Figure 5.4: End-to-end execution time with default Spark MLlib, Arion, and the naive adoption of other processing variants. (Note log-scaled Y-axis in (a).)

5.4.1 Overall Performance

In our first experiment, we study the overall performance gain obtained from Arion. For this purpose, we generate random square matrices of different orders with uniform densities as shown in Table 5.3. We store both dense and sparse matrices in text files with same format to maintain the same data preprocessing process of Spark. We use density 1.0 for dense matrices and 0.05 for highly sparse matrices. We see that our raw input file sizes scale up to 80 GB with computation of up to 4.3 billion data points. We perform generalized matrix multiplication from an input matrix, $\mathbf{X}$, to compute Gramian Matrix, $\mathbf{XX^T}$. Due to the nature of the computation of Gramian Matrix, the pair of matrices of multiplication would be either both dense or both sparse. However, Arion may still employ acceleration of sparse-dense matrix multiplication on block sub-matrices where there may be skewness within the matrix. To study the performance of the different processing variants, we consider several scenarios: (i) default Spark MLlib, Arion for both dense and sparse matrix $\mathbf{X}$; (ii) Arion using OpenBLAS only (with **Selector** and **Scheduler** disabled), Arion using NVBLAS only, Arion using **GPU Impl** with cuBLAS only for dense matrix $\mathbf{X}$; (iii) Arion using spBLAS only and **GPU Impl** with cuSPARSE only for sparse matrix $\mathbf{X}$. We use the default block size of $1K \times 1K$ for our experiments.

We record the end-to-end execution time of the application, which includes reading input data file, creating a *BlockMatrix* matrix type from the input file through intermediate data types (from *RowMatrix* to *IndexedRowMatrix* and *CoordinateMatrix*), and performing matrix multiplication on *BlockMatrix*. Figure 7.5 shows the results. The x-axis represents execution time against the order of matrices as shown in Table 5.3.

We first compare the performance of Arion to default Spark MLlib. Here, we observe that Arion performs better than MLlib for dense matrices with an average speed-up of 1.87×, and up to 2.47×. For sparse matrices, Arion speeds up performance by 1.15× on average and up to 1.5× for $64K \times 64K$ matrix. This performance improvement mainly comes from the hardware acceleration for performing matrix multiplications (further investigated in subsection 5.4.2. The performance of Arion over MLlib is higher in the case of dense matrices (y-axis is only log scaled in the case of dense matrix) due to the fact that the computation of dense matrices is more intensive than the case of sparse matrices, and dense

matrices can benefit more from the GPU computation power.

Next, we analyze the impact of different implementation variants. From the figure, we see that for dense matrices, all hardware acceleration methods (OpenBLAS, NVBLAS, cuBLAS) finish earlier than MLlib. For sparse matrices, hardware acceleration methods (spBLAS, cuSPARSE) perform similar to, or only slightly better than MLlib. This is because for highly sparse matrices with 5% of non-zero elements, the overhead of random memory access outweights the benefit of GPU acceleration where the overhead is incurred by storing non zero entries non-sequentially in memory due to the sparsity of the matrices. This observation is consistent with the results reported in [215]. Overall, ARION outperforms other implementation variants for both dense and sparse matrices. Among all matrices sizes, ARION increases the performance by 10.75% and 20.28%, compared to the naive use of OpenBLAS and NVBLAS, respectively. ARION performs similar to cuBLAS case because ARION selects cuBLAS for dense matrices in our system, and most tasks are scheduled to GPU over CPU due to the short task execution times. The performance improvement reaches up to 31.89% for dense matrices. Similar trend is observed in sparse matrix case where ARION further improves performance by 10.59% compared with cuSPARSE, and 10.13% compared with spBLAS. The main reason for this is that ARION dynamically chooses the optimal variant per block based on density and order of block sub-matrices and schedules tasks to otherwise idled compute resources, consequently increasing resource utilization significantly.

5.4.2 Performance Breakdown

The end-to-end performance includes data preprocessing (reading input file and converting to *BlockMatrix*), matrix multiplication, and persisting results to storage. Figure 7.7 shows the breakdown of the end-to-end time for these three stages under ARION. The time fraction for the "Multiply" phase suggests how much time is spent on computation, which is the component where hardware acceleration can speed up. As seen in the figure, in general, dense matrices spend more time on multiplication than sparse matrices, thus dense matrices benefit more from hardware acceleration. Moreover, larger matrices spend more time on computation while smaller matrices spend more time in data preprocessing. We thus conclude that hardware acceleration favors bigger and denser matrices. The benefit of hardware acceleration on computation tends to be discounted by large data preprocessing time for smaller and more sparse matrices.

In the "Multiply" stage, Spark tasks shuffle the block matrices to get the right sub-matrices, and then perform the computation. To further investigate the acceleration of computation, we eliminate the data shuffle phase, and record only the computation time. Note that we use aggregated compute time as a simplified metric due to the complexity of Spark task scheduling for this set of experiments. Figure 5.6 shows the speed-ups of aggregated compute time of ARION and other variants compared to MLlib. The observed speed-up for dense matrices increases with matrix size, and peaks at 57.04× with the matrix of $64K \times 64K$.

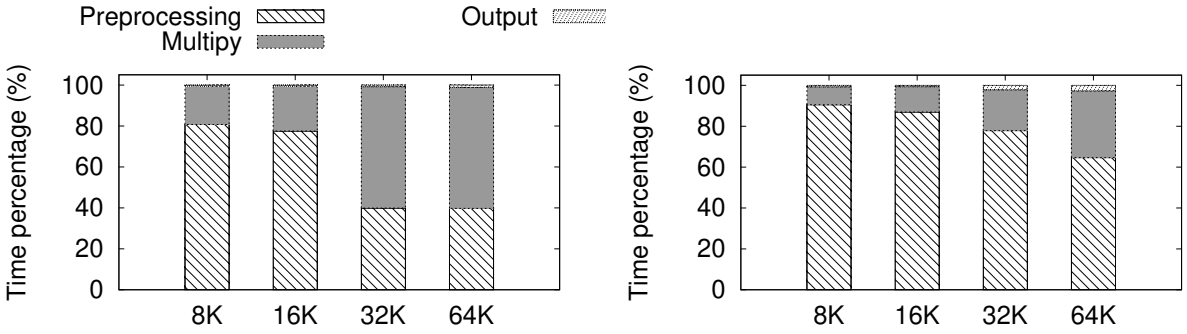
This is because Arion fully utilizes both CPUs and hardware accelerators. As the matrix grows, hardware accelerators demonstrate greater speed-ups. Using hardware accelerators improves the computation performance by up to 57 times. However, the overall performance decreases due to the data preprocessing and other framework overheads. For sparse matrices, we still see a slight speed-up (up to 1.5×) under Arion over the Scala implementation of MLlib, which is consistent with the overall results in Figure 5.2(a). Again, we observe that Arion performs similar to or better than other implementations for both dense and sparse matrices, similar to the overall performance. As the matrix size grows, the performance improvement of Arion and **GPU Impl** increases, up to 88.2% and 84.9% for dense and spare matrices, respectively, compared to BLAS libraries. We expect the performance improvement of Arion to increase with growing matrix sizes against just vanilla hardware acceleration.

Next, we compare the computation speed-ups of different implementation variants. Among the three hardware solutions for dense matrices (OpenBLAS, NVBLAS, and cuBLAS), cuBLAS outperforms the other two variants, which is consistent with results in Figure 5.2(a). Moreover, OpenBLAS slightly surpasses NVBLAS with an average improvement of 9.07%, which is different from the pure measurement of Figure 5.2(a). This is because the default block size of Spark falls out of the sweet spot of NVBLAS. As described in subsection 5.3.4, NVBLAS is designed to opt for large matrices. However, Spark divides matrices to $1K \times 1K$ block sub-matrices, which hinders NVBLAS to benefit from its optimizations such as matrix tiling and pipelining of data transfers. Furthermore, NVBLAS does not support stream processing and kernel batching as in **GPU Impl**, falling short in meeting the demand of high concurrency of Spark and resulting in low GPU utilization. Finally, the overhead of NVBLAS in initializing the NVBLAS library, reading a configuration file, and selecting GPUs is not negligible. In contrast, **GPU Impl** avoids this overhead by directly operating through GPU function calls. As a result, we see that in Figure 7.5, for dense matrices, NVBLAS always performs the worst among the hardware acceleration variants. For the libraries for sparse matrices (spBLAS and cuSPARSE), spBLAS performs similar to the Scala implementation of MLlib, and cuSPARSE degrades computation heavily. This is also in line with our hardware profiling results in section 7.3 (Figure 5.2(b)). However, we do not see this impact on the overall performance in Figure 7.5. This is because the computation portion is not dominant for sparse matrices (Figure 7.7), thus the impact is again negated by the other overheads. Nevertheless, Arion's selection of the best processing variants and utilizing multiple resources still results in overall performance improvement.

5.4.3 System Utilization

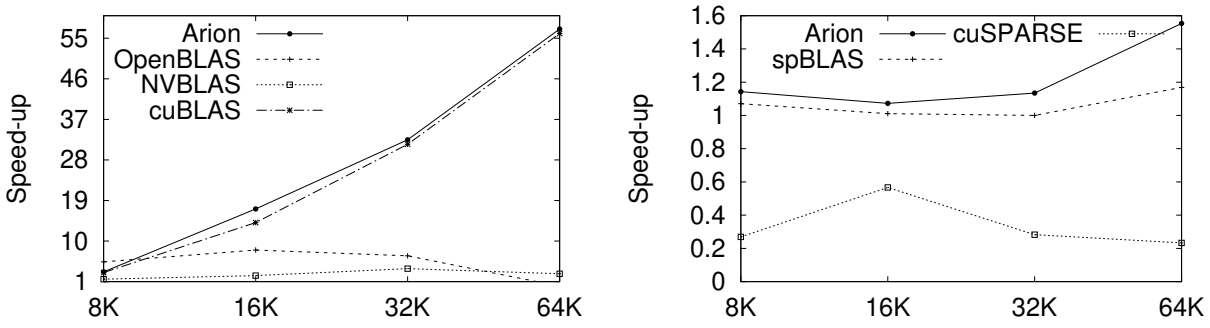
Next, we repeat the experiments with $16K \times 16K$ dense matrix and $64K \times 64K$ sparse matrix using MLlib, Arion, OpenBLAS, NVBLAS, and **GPU Impl**. We observe differences in resource utilization across variants.

First, we compare the CPU utilization of MLlib, Arion, and OpenBLAS as shown in Fig-



(a) Dense matrix. (b) Sparse matrix.

Figure 5.5: Performance breakdown of studied matrices under ARION.



(a) Dense matrix. (b) Sparse matrix.

Figure 5.6: Speed-up of aggregated compute time with ARION, and other processing variants with respect to the Scala implementation in MLlib.

ure 5.7. Here, x-axis represents the time sequences, and the y-axis depicts the average CPU utilization of the testbed nodes. Since the computation using MLlib takes longer than 250 seconds, we omit the tail results after $250sec$, where the average utilization is less than 1.3%. We observe that OpenBLAS has the highest CPU utilization, reaching 100%, since this variant heavily exploits CPUs to accelerate matrix multiplication. MLlib also exhibits medium CPU utilization, which reaches 37.24% due to the usage of Scala based implementation executing on CPUs. However, this Scala based implementation has much lower CPU utilization compared to OpenBLAS. In contrast, ARION shows the lowest CPU utilization with an average of 2.49%. For default $1K \times 1K$ dense blocks, ARION favors GPU accelerations for computation, and **GPU Impl** calculates fairly fast for each task and finishes earlier than OpenBLAS. Moreover ARION selects **GPU Impl** for all tasks as there is only one wave of tasks for $16K \times 16K$ matrix.

Figure 5.8 shows the aggregate GPU utilization of all nodes for ARION, ARION using **GPU Impl** only, and NVBLAS. Note that maximum GPU utilization reaches $6 \times 100\%$. Here we observe that NVBLAS demonstrates the lowest GPU utilization among the three. The tiling of NVBLAS is effective on block sub-matrices with size greater than $2K$ instead of small ones. Compared to NVBLAS, **GPU Impl** increases GPU utilization by associating

each small task to a CUDA stream and batching the task execution. Arion shows a similar GPU usage with **GPU Impl** due to the main adoption of **GPU Impl** for the $16K \times 16K$ matrix case. The aggregated GPU utilization of both Arion and **GPU Impl** fails to reach the maximum capacity of the cluster. This is because in Spark, matrix multiplication tasks first shuffle data to retrieve needed sub-matrices. The actual computation happens after all the data has been obtained. In this case, the actual computation of different tasks may start at different times based on how long the shuffle takes for a task. In the case of multiple waves, the discrepancy in computation start time increases across tasks. This feature of MapReduce based scale-out platforms like Spark limits the chances of fully extracting GPU utilization with the batching task design of **GPU Impl** inside Arion, and also prevents the **Scheduler** of Arion to detect GPU resource contention. However, such shuffle behavior is nondeterministic so we can benefit from the flexibility of Arion to fully utilize idle resources when available. Moreover, when applied to a multi-tenant environment or multiple parallel phases of a job—where there may be resource contention, e.g., when a GPU intensive job runs alongside a matrix multiplication job—, **Scheduler** is able to avoid resource contention by scheduling tasks to resources with higher availability. For highly synchronized scale-out platforms such as Bulk Synchronous Parallel (BSP) based frameworks [138, 169, 195], Arion is expected to be able to fully utilize the hardware resources and schedule tasks to avoid resource contention.

To evaluate how Arion effectively utilizes both CPU and GPUs resources, we run an experiment with sparse matrix multiplication of size $64K \times 64K$. Figure 5.9 shows aggregate CPU and GPU utilization. Here, while CPUs are mostly fully utilized, GPUs exhibit a spike pattern throughout the execution. While **Selector** initially selects spBLAS, **Scheduler** notices idle GPU resources and fully occupied CPU slots. **Scheduler** then reassigns the tasks to GPU based variants (**GPU Impl** with cuSPARSE). The fluctuation in GPU utilization comes from Arion's decision to respect choices of **Selector**. That means tasks are assigned with spBLAS whenever a CPU slot becomes available. Thus, we see that Arion improves the overall resource utilization.

5.4.4 Impact of Block Size

In our next experiment, we perform Gramian matrix calculation with $16K \times 16K$ dense matrix using Arion with different block sizes. We record the end-to-end runtime of all cases and break it down into the three stages as illustrated in Figure 5.10. Here we can see that Spark spends most of time on data preprocessing and performing multiply computation, and the time on writing outputs is thus trivial. Therefore, we focus on preprocessing and multiplication times. The data preprocessing is minimum with block size of 512, but the matrix multiplication is minimum with block size of $1K$, the default case. The overall performance reaches minimum at 9.08 minutes with the block size of $1K$. Although suboptimal block sizes (such as 256) cause huge data preprocessing overhead from data partitioning and shuffling, in this paper we fix block size with default value and focus on optimizing computation time.

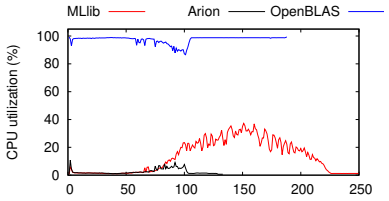


Figure 5.7: CPU utilization of $16K \times 16K$ dense matrix using Spark MLlib, ARION, and OpenBLAS.

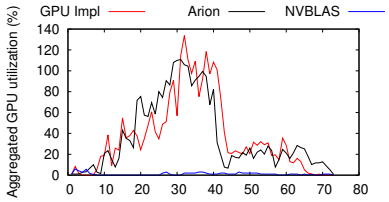


Figure 5.8: Aggregated GPU utilization of $16K \times 16K$ dense matrix using ARION, **GPU Impl**, and NVBLAS.

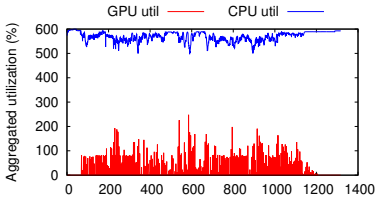


Figure 5.9: Aggregated utilization of $64K \times 64K$ sparse matrix using ARION.

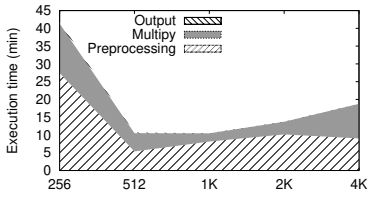


Figure 5.10: Breakdown of end-to-end execution time with different block sizes.

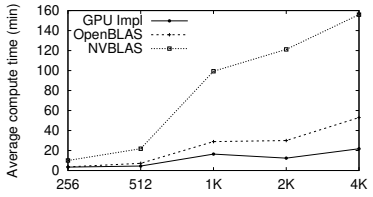


Figure 5.11: Average computation time of *BlockMatrix* multiplication using different block sizes with **GPU Impl**, OpenBLAS, and NVBLAS.

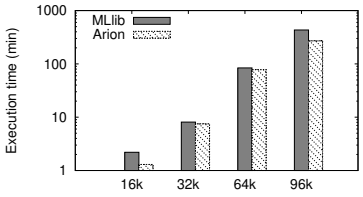


Figure 5.12: Performance of default Spark MLlib and ARION in a 128-node scale. Y-axis is log-scaled.

Next, we study the relationship between computation and block sizes, by repeating the experiment with hardware acceleration implementations (**GPU Impl**, OpenBLAS, and NVBLAS). We record the average compute time of each task. Figure 5.11 shows the results. For all the implementations, computation time increases as the block size increases. Larger block sizes result in larger matrices and longer computation time, which is consistent with our earlier observation (Figure 5.2(a)). However, in contrast to Figure 5.2(a), where NVBLAS performs better when the matrix size grows and finally outperforms other two implementations, here we see that NVBLAS performs worse as the block size grows and the gap between the other two variants also increases. This is because under the highly concurrent environment of Spark, NVBLAS cannot handle multiple task requests, and causes a delay to execute subsequent tasks. With smaller sizes of matrices, although the number of tasks increases, the computation finishes fast enough to overlap the shuffle of other tasks. However, when the matrix size grows, even if there are fewer tasks, the computation takes longer and tasks are prone to a higher chance and duration of waiting. On the contrary, the Spark-aware design of **GPU Impl** yields a consistent best performance among the processing variants in any block size.

5.4.5 Scalability of Arion

In our final experiment, we test the scalability of Arion. In this experiment, we scale Arion in a 128-node "Rhea CPU" cluster (4096 cores in total). Table 5.2 shows the specifications for both setups. Here, we configure Spark executor with 120 GB memory and 64 cores for the CPU nodes, and repeat the experiments for dense matrices using default Spark MLlib. We record the end-to-end execution time of both default Spark MLlib and Arion. Arion is able to apply CPU hardware acceleration for dense matrix multiplications in this set-up. We are able to scale our input matrix up to $96K \times 96K$ with a raw file size of 177 GB for Arion. However, default Spark MLlib fails to finish the $96K \times 96K$ matrix case due to *OutOfMemory* error after 7.2 hours running as plotted in the figure. This is because the computation with Scala implementation is executed inside JVM and thus requires on-heap memory space, while hardware acceleration uses out-of-heap memory (sometimes GPU memory) for calculation. Figure 5.12 shows the results. Note the log-scaled y axis. We see that Arion is also able to improve the performance by 13.35% (up to 40.9%) in a large scale system. This improvement is however limited by the CPU only infrastructure of the "Rhea CPU" environment. We expect Arion to have more performance improvement on emerging systems that have multiple hardware accelerators.

We also compared Arion on a 6-node "Rhea GPU" with default Spark on a scale-out 16-node CPU only cluster. Our results show that Arion on a smaller setup performs similar to default Spark MLlib on a larger setup (the detailed results are omitted due to page limit). Moreover, Arion improves performance by 17.45% for $16K \times 16K$ matrix, and 39.13% for $32K \times 32K$ matrix compared with default Spark MLlib on a larger setup. Since Arion effectively utilizes both CPUs and hardware accelerators in the cluster, we can achieve the same or better performance with a smaller cluster of fat machines than a larger cluster of less-endowed machines.

5.5 Chapter Summary

We have presented Arion, a dynamic hardware acceleration framework atop Spark. Arion supports scale-up accelerations for linear algebraic operations in scale-out data processing platforms. The approach adopts both drop-in solution of BLAS libraries and customized GPU acceleration in Spark for the best-effort processing of both dense and sparse matrices. Moreover, we design a scheduler that maps tasks to different resources at runtime based on both matrix characteristics and system utilization. Our evaluation of Arion shows a 2× end-to-end speed-up compared to extant solutions in Spark. In addition, Arion is able to exploit hardware acceleration to enable the use of a smaller-sized cluster to achieve performance comparable to that of a larger Spark cluster. Although this work focuses on the usecases of matrix operations to provide a proof-of-concept, the result from this work can have a broad impact on various algorithms in machine learning and graph analysis, since

the most of machine learning and graph algorithms are implemented in a layered approach on top of matrix operations.

Chapter 6

Resource Mangement for Heterogeneous Applications

6.1 Introduction

MapReduce [17] has emerged as the *de facto* distributed processing model for big data analytics due to its ease-of-use and ability to scale. Although a wide-range of applications, such as web crawling and text processing [12] and log querying [10], benefit from the MapReduce model, there exists numerous rich data analytics workflows that can not be fully or efficiently captured using MapReduce. Consider Metagenomics [122], a life science workflow that consists of clustering and multi-dimensional scaling operations that involve parallel linear algebra computations [163]. Such compute- and communication-intensive jobs entail intricate communication patterns that cannot be captured by only the shuffle supported in MapReduce. For such applications, the well established Message Passing Interface (MPI) [15] is more suitable due to its ability to support any communication pattern. Moreover, MPI has proved its mettle in large-scale high-performance scientific computing (HPC). However, simply porting the I/O-intensive data analytics workflow to MPI is impractical, given the demonstrated ability of MapReduce to deliver high-throughput for data-intensive workloads. Moreover, a MapReduce application has the added advantage of being able to run on public cloud services, e.g., Amazon AWS [1], which removes the need for dedicated in-house clusters and lower the barrier-to-entry for users. Consequently, a new two-cluster paradigm is emerging, wherein the compute-intensive tasks of a workflow are run using MPI on traditional HPC clusters, and then the data is moved to a MapReduce cluster for data-intensive processing [141]. This is undesirable due to unnecessary dual maintenance of two kinds of clusters. Moreover, such settings entail performance degrading manual copying of data from one model to another, e.g., from the cluster file systems of HPC to HDFS [141] for Hadoop/MapReduce.

An alternative approach is to port MPI applications to MapReduce, either via auxiliary libraries [3, 11, 67, 115, 217] or by re-designing parallel algorithms using MapReduce APIs [117, 157, 199, 213]. Though promising, this approach is not always possible without completely redesigning well-established applications [199], which undermines the effort done in validating and developing the original applications. A promising development here is the evolution of the standard monolithic Hadoop design [31] into YARN [187] via decoupling of resource management from task management. YARN lays the foundation for supporting diverse programming paradigms and not just MapReduce on the managed resources. However, while YARN is programming model agnostic in concept, it is based on refactored code from Hadoop 1.0 [187], and inherits many of the design decisions and implementation aimed at supporting only the MapReduce model.

In this paper, we address the above issues by designing GERBIL, a YARN-based framework for *transparently* co-hosting unmodified MPI applications alongside MapReduce applications on the same set of resources. Enhancing YARN to host MPI applications allows realization of rich data analytics workflows as well as efficient data sharing between the MPI and MapReduce models within a single cluster. GERBIL enables fine-grained application-to-model matching and has the promise to significantly improve performance and ease-of-use. Users can leverage say an MPI matrix multiplier with a MapReduce log analysis, both within a single workflow running on an integrated cluster without manual hacking. GERBIL hides the low-level details such as resource negotiations and management of dual models from the users, while supporting user-specified task allocation strategies, e.g., optimizing for minimum job wait time, inter-process locality, or desired cluster utilization, in a multi-tenant environment. Moreover, there is a great interest from the open source Hadoop community to embrace MPI as a first class citizen on YARN [143]. GERBIL aims to deliver on this promise.

We faced several challenges when designing GERBIL. YARN provides for customized application management component (application master, AM) that can be exploited to support multiple programming models. First, we need to design an AM that manages the life cycle of allocated resources and the assignment of MPI processes to the allocated resources in an efficient way, requiring in-depth understanding of YARN and its interactions. Second, we found that there is a significant difference in the design and resource management principles of MapReduce and MPI, precluding a straightforward extension of YARN to support MPI. For example, YARN allocates resources based on heart beats from node managers, which results in long resource allocation time and can lead to significantly long MPI application launching compared to a native MPI cluster. Third, we have to address cases where available resources are not enough to assign all tasks of an MPI application. A naive solution of waiting for more resources to become available, and not using the currently available resources, can lower cluster utilization and increase application turn-around time. We propose resource over-subscription so as to allow an MPI application to start albeit in a degraded mode due to sharing/over-subscription of the underlying resources.

Specifically, this paper makes the following contributions:

- We identify a fundamental mismatch between YARN resource negotiation mechanism and the MPI programming model (Section 6.2). We found that the MapReduce-tailored, container-based resource negotiation protocol in YARN does not work well for MPI applications that in essence require gang resource allocation.

- We present the design and implementation of Gerbil (Section 7.3 and Section 6.4), which supports traditional MPI job submission semantics as well as allocation of cluster resources to MPI jobs via negotiating with YARN and transparently launching the applications.

- We present five resource provisioning strategies for Gerbil for multi-tenant environments (Section 6.3.3). Such flexible resource provisioning helps in executing the MPI applications based on various cluster utilization goals such as minimizing application launching time, reducing network traffic and increasing cluster resource utilization.

- We conduct an in-depth performance study of Gerbil on an 19-node cluster [25] using representative applications (Section 7.3.3). Our experiments show that Gerbil can run MPI applications with performance comparable to the native MPI setup. The runtime overhead of Gerbil, mostly incurred by the resource negotiation with YARN, is observed to be constant and would become negligible when amortized across long-running applications.

6.2 Enabling Technologies

In this section, we describe MPI and YARN, which serve as the enabling technologies for Gerbil.

6.2.1 The MPI Framework

MPI (Message-Passing Interface) [15] uses explicit messages to coordinate between distributed processes. In our work, we use the publicly available MPICH [16]. MPICH provides several internal process managers [61] such as Gforker, Remshell, SMPD, Hydra [7], and MultiPurpose Daemon (MPD) [69] that we use in our prototype. A typical MPI job is submitted through the `mpiexec` command, which also accepts arguments such as a machine file or number of processes to use. The process manager then spawns connected daemon processes on participating nodes, which prepare the communication environment, handle process binding, and spawn the application MPI processes. The process manager also handles tasks such as signal forwarding and I/O forwarding, and cleanup upon task completion. The MPI model does not provide resource management mechanisms, and relies on systems such as Portable Batch System (PBS), SLURM [109], Torque [178], Moab [14] and Cobalt [83].

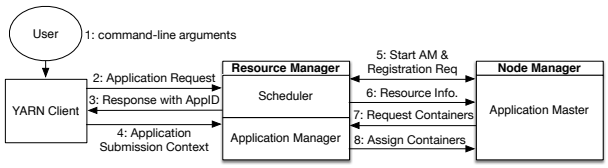


Figure 6.1: Steps of launching an application in YARN.

6.2.2 YARN

YARN [187], the second generation Hadoop release, has two key components: a global *ResourceManager* (RM) that provides resource management across all available resources and allocates resources for applications, and a per-application *ApplicationMaster* (AM) that manages the allocated resources and handles job scheduling and monitoring. The RM leaves the control over the applications to their associated AMs. YARN uses a *container* abstraction to allocate and manage resources. The containers provide some degree of performance isolation and guarantees for application execution. Figure 6.1 shows the steps involved in running an application on YARN. A user submits jobs through a client to the RM by providing: (a) application submission context containing ApplicationID, user, queue, and other information needed to start the AM; and (b) container launching context (CLC) that specifies resource requirements (Memory/CPU), job files, security tokens, and other information needed by the AM to run. The RM then allocates the requested container, starts the AM in the container, and passes control to the AM that can then request more containers from the RM as needed.

Although YARN is designed as a flexible and scalable resource manager, it is not well-suited for MPI for two reasons. First, YARN does not provide support for gang scheduling needed by MPI jobs, which can lead to long waiting time before an MPI job can start. Second, fine-grained resource container allocation in YARN can lead to multiple containers allocated to a single physical node. This creates challenges for the MPI process manager design, as traditional MPI setups support only one manager process per node.

6.3 System Design

In this section, we first discuss the objectives of GERBIL design. Then, we give an overview of our architecture and describe the major components of GERBIL, followed by a discussion on how I/O and fault tolerance can be supported.

6.3.1 Design Objectives

The key design objectives of GERBIL are as follows.

Transparent System Interface: We want to ensure that GERBIL allows unmodified MPI

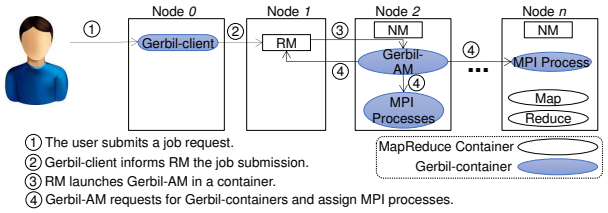


Figure 6.2: Gerbil architecture for running MPI on YARN.

and MapReduce applications. This is crucial for increasing the efficacy of our approach and easier adoption by the users, as any solution requiring modifications either to YARN or to MPI will induce additional maintenance difficulties or burden the application developers.

Ease of Use: MPI is the extant parallel programming framework for scientific computing, and users are already familiar with its process model and job submission semantics. We aim to preserve this ease-of-use and familiar user experience by enabling MPI-like interactions in Gerbil.

MPI-Aware Container Allocation: As explained earlier (Section 6.2), the *fine-grained* container allocation in YARN does not match well with the *gang allocation* requirement of MPI applications. It is crucial for Gerbil to provide MPI-aware resource allocation atop the fine-grained resource allocation of YARN to mitigate this problem.

Tunable Allocation Strategy: Gerbil is aimed at multi-tenant environments, which entails adapting the resource allocation strategy based on dynamic cluser resource utilization and user demands. For example, the user may opt for minimizing job wait time, increasing overall resource utilization, or achieving better inter-process locality to reduce data movement across nodes. To facilitate this, Gerbil must support a flexible and extensible interface that would allow users to specify the allocation strategy best suited to their applications.

6.3.2 Architecture Overview

Figure 6.2 shows the architecture of Gerbil. We leverage YARN to manage the cluster, and allocate and launch containers on which MPI jobs would be run. We use the allocation to first provision a temporary MPI "cluster" (tMc) for each submitted MPI job. A tMc is composed of a set of containers that RM and Gerbil allocate, including a special container for the associated AM, and exists only for the duration of the associated job. Gerbil then instantiates tMc with an execution environment similar to that of a traditional MPI cluster. This ensures that no modifications or re-compilation is needed for running MPI jobs on the allocated resources comprising tMc. Moreover, multiple tMc's can co-exist with each other as they host different applications.

Gerbil consists of three main components. Gerbil-Client is responsible for accepting user command-line arguments and providing the parameters and environments needed by YARN to launch Gerbil-AM. Gerbil-Client provides an MPICH compatible application launch-

Options	Arguments
`-a, --mpi`	MPI executable file
`-l, --list`	MPI application list file
`-n, --processes`	number of processes
`-p, --priority`	proirity of the MPI application
`-f, --hostfile`	nodes/processes description file
`-r, --relax`	allocation algorithm to be used
`-o, --oversubscribe`	oversubscribe resources
`-i, --arguments`	arguments for the MPI application
`-k, --kill`	YARN application ID to terminate

Table 6.1: Command-line arguments supported by GERBIL-Client.

ing interface shown in Table 6.1. This allows MPI applications to transparently utilize YARN managed resources. After a user submits an MPI job, GERBIL-Client sets up the necessary environment for executing the application. The client then constructs the application submission context, including command line information, for the RM that schedules an available container and launches GERBIL-AM. At this point, control is handed over to GERBIL-AM.

GERBIL-AM is the customized application manager that handles execution of MPI jobs on YARN. This component requests needed containers from the YARN RM, manages the life cycles of allocated containers, assigns processes to containers based on user-specified allocation policies, and launches a GERBIL-Container on each allocated container. GERBIL-Containers launch associated MPI process manager, set up the MPI execution environment, and monitor the resource usage of MPI processes running within the containers.

6.3.3 User-Specified Container Allocation Modes

Table 6.2 shows the different modes supported by GERBIL, which allows the users to request cluster resources with specific characteristics. In the `alloc_relax_none` mode, a user submits an MPI host file describing the list of preferred hosts and the number of processes to run on each host. In this case, GERBIL strictly respects the specified requirements, requests one container per listed host and assigns exactly the number of processes per node as specified in the host file. This mode gives users fine-grained control over process allocation. The flip side is that when any one of the specified hosts does not have sufficient capacity to support the requested containers, GERBIL-Client has to wait until the resources become available for starting the GERBIL-AM. Here, users opt for control at the cost of potentially increased allocation latency. Alternatively, users can only specify the number of processes needed for their MPI jobs in the `alloc_relax_all` mode, where GERBIL freely allocates the nodes based on resource availability throughout the cluster. However, such flexibility comes at the cost of potential loss in data locality as containers are now assigned to arbitrary nodes.

Algorithm 5: Container allocation – common functions

```
Input: hostfile, relax, total_nproc, cluster_list
begin
    {m, v} ← minimum MEM, vCPU from YARN config;
    sort_list_descend(cluster_list);
    if hostfile exists then
        np ← total_nproc;
        local_list ← new localList;
        for each (node, nproc) in hostfile do
            node_size ← node.get_size();
            size ← {m × nproc, v × nproc};
            switch relax do
                case "none" do
                    ALGO_relax_none();
                end
                case "loc" do
                    ALGO_relax_loc();
                end
                case "dist" do
                    ALGO_relax_dist();
                end
                case "loc + dist" do
                    ALGO_relax_loc+dist();
                end
            end
        end
        if relax == "dist"||relax == "loc + dist" then
            round_robin(local_list, np);
        end
        combine_containers(local_list);
    end
    else
        /* relaxed allocation */
        ALGO_relax_all();
        combine_containers(cluster_list);
    end
end
Function combine_containers(list)
    for each node in list do
        if node.get_num_containers() > 1 then
            combine containers into one;
        end
    end
Function round_robin(list, np)
    while np > 0 do
        node, node_size ← get first node from list;
        if node_size < {m, v} then
            degraded ← true ;
        end
        request_container(node, {m, v});
        sort_list_descend(list);
        np − −;
    end
```

The aforementioned allocation modes represent the two extreme cases of container allocation in Gerbil. To support a wider range of application needs as well as to mitigate the allocation latency due

Algorithm 6: Container allocation – mode implementation

```
Function ALGO_relax_none()
    if node_size ≥ size then
        request_container(node, size);
    end
    else if node_size > {m, v} then
        request_container(node, node_size);
        degraded ← true ;
    end
    else
        request_container(node, {m, v});
        degraded ← true ;
    end
    local_list.add(node);
Function ALGO_relax_loc()
    top_node ← get the top node from cluster_list;
    top_size ← cluster_list.get_size(top_node);
    if top_size ≥ size then
        request_container(top_node, size);
    end
    else if top_size > {m, v} then
        request_container(top_node, top_size);
        degraded ← true ;
    end
    else
        request_container(top_node, {m, v});
        degraded ← true ;
    end
    local_list.add(top_node);
    cluster_list.remove(top_node);
Function ALGO_relax_dist()
    local_list.add(node);
Function ALGO_relax_loc+dist()
    top_node ← get the top node from cluster_list;
    local_list.add(top_node);
    cluster_list.remove(top_node);
Function ALGO_relax_all()
    round_robin(cluster_list, total_nproc);
```

to the lack of gang scheduling support in YARN under multi-tenancy, GERBIL also supports three more container allocation modes with varying degree of flexibility as shown in Table 6.2.

Relaxation Mode: (alloc_relax_)	Location	Distribution	Node Count
none	·	·	·
loc	×	·	·
dist	·	×	·
loc+dist	×	×	·
all	×	×	×

Table 6.2: Container request modes in Gerbil. For each mode, '×' and '·' denote whether a factor is relaxed or not, respectively. Here, location, distribution, and node count means the location (i.e., nodes on which processes are launched), the distribution (number) of processes per node, and the total number of nodes used to service a request, respectively.

These modes allow relaxing the requirement for the location of processes (`alloc_relax_loc`), distribution of processes per node (`alloc_relax_dist`), or both the location and distribution but not the number of requested nodes (`alloc_relax_loc+dist`). The different allocation modes provide users with flexibility to run their MPI applications based on different application characteristics by providing the appropriate host file and allocation mode arguments. For example, for communication-intensive MPI applications, it is preferred to allocate the containers and nodes close to each other so that the communication between processes on different machines does not involve multiple network hops. This can be achieved by using the `alloc_relax_dist` mode. Similarly, for CPU-intensive MPI applications where the location of the processes is not crucial, the `alloc_relax_all` mode provides for more flexibility and faster allocation. However, if the user wants performance guarantee, she can ensure a minimum number of compute nodes is allocated to the job by using the `alloc_relax_loc+dist` mode. Thus, the main purpose of supporting multiple allocation modes is to reduce application launch time without compromising their needs.

6.3.4 Container and MPI Process Management

Gerbil-AM is responsible for deciding the number and size of containers to request, as well as employing strategies for assigning processes to containers based on user-specified allocation mode and currently available cluster resources. The actual resource negotiation with the RM is done by Gerbil-AM using container requests and leases, communication for which are piggybacked on AM-RM heartbeat messages. To minimize the container launching time, we pack all the needed information, e.g., the number, size and location of the containers, etc., within a single heartbeat message. This information is used to assign processes to hosts depending on the allocated resource capabilities and the user-specified container allocation mode. Gerbil-AM then manages the application life cycle and affects policies to automatically recover from any failures.

Container Allocation and Process Assignment

MPI process management mechanisms force GERBIL to have only one container per node per MPI job, as only one MPI process manager per node is allowed per job. If multiple MPI processes are to be spawned on a node, they are all run inside the same container running on the node. Thus, while a node can have multiple containers each for a different MPI application, each application can have only one container per node. Moreover, keeping all the processes of an application in one container per node makes it easy for the GERBIL-Container to monitor and track the whole process tree, and determine the resource utilization of all the processes within the container. Thus, GERBIL-AM only requests one container per node for each MPI application, and assigns one or more MPI processes within each container. This is also beneficial in limiting the impact of YARN container allocation overhead on GERBIL as we show in our evaluation (Section 6.5.6). Since GERBIL may host multiple MPI processes in a single container, the optimal size, e.g., the amount of memory and the number of vCPUs, of different containers can vary depending on the number of processes assigned to the container. For example, GERBIL attempts to allocate twice as much capacity, e.g., memory and vCPUs, to a container with two processes compared to a container that hosts just one process.

Resource Oversubscription

GERBIL-AM compares container allocation requests against available resources to detect oversubscription, which occurs if the number of requested resources exceeds the available resources. Such oversubscription may occur under any of the resource allocation modes described earlier in Section 6.3.3. GERBIL handles such scenarios depending on the user command line option *oversubscription*. If the option is set to $false$,—and the requested resources do not exceed the max capacity of the setup, which results in an error—GERBIL simply waits until sufficient resources become available before allocating the container. If the option is set to *true* (default), GERBIL-AM runs the MPI runtime in an oversubscription mode, i.e., it allows different containers/processes to compete for the same resources. The goal is to avoid the unpredictable wait times for resources to become available.

Different implementations of MPI handle such oversubscription related performance degradation differently. In OpenMPI [21], the degraded mode can be controlled by Modular Component Architecture (MCA) parameter `mpi_yield_when_idle`. In MPICH $v1.1$ or later, users can configure the system to control whether to perform in a degraded mode or to keep the performance but sacrifice the intra-node communication. Oversubscription can sometimes be beneficial, especially in competitive environments, e.g., modest levels of oversubscription has been shown to improve system throughput by 27% [107].

Container Allocation Algorithms

We design five container allocation algorithms in Gerbil-AM based on the five allocation modes described in Section 6.3.3, namely `ALGO_relax_none`, `ALGO_relax_loc`, `ALGO_relax_dist`, `ALGO_relax_loc+dist`, and `ALGO_relax_all`. Algorithm 5 and Algorithm 6 show the steps taken during the container allocation process when supporting the five allocation modes. We set the *oversubscription* option to a default of *true* in our algorithms.

The input variables of Algorithm 5 are: (i) the host file path, $hostfile$, that contains a list of $\{node, nproc\}$ pairs where *node* is a host and *nproc* is the number of processes to run on the node; (ii) the relaxation mode *relax*; (iii) the total number of processes to use *total_nproc*; and (iv) the cluster list *cluster_list* that specifies the available nodes and is retrieved by Gerbil-AM from the YARM RM. This algorithm retrieves the value of the minimum allocation of memory m and virtual cores v from the YARN configuration file and uses this information as the basic resource allocation unit. However, users can change the value of $\{m, v\}$. By default, a single process is assigned a unit of $\{m, v\}$ resources, while *nproc* processes are assigned with $nproc \times \{m, v\}$ resources. We also support a user option that, if set to 'true,' implies that the system should sort the nodes from the cluster list in a descending order based on their available resources. We give higher priority to memory than CPU, as over-committing memory would cause extra disk I/Os and worse performance degradation than over-committing CPU resources. Thus, a node is ranked higher if it has more amount of free memory. If two nodes have the same amount of memory, then the node with more free virtual cores is ranked higher.

***A*LGO_relax_none:** When the user specifies a host file with hosts and number of processes per host with `alloc_relax_none` mode, containers are requested strictly based on the user's request. For each node *node* in the host file, Gerbil-AM adds it to a list of nodes to use, *local_list*, and calculates the needed resources *size* as $\{m, v\} \times nproc$. If the available resources *node_size* of *node* is greater than *size*, Gerbil-AM requests a container of size *size*. If *node_size* is smaller than *size* but greater than $\{m, v\}$, Gerbil requests a container of size *node_size*. Otherwise, it requests a container of size $\{m, v\}$. Note that the last two cases result in over-committing of the container resources.

***A*LGO_relax_loc:** If the user selects mode `alloc_relax_loc`, the process distribution per node is respected but the specified hosts may be different depending on availability. Gerbil-AM retrieves the needed nodes from the top of the *cluster_list* to ensure a higher probability of getting the desired resources. The remaining process is the same as before. Finally, every time a container is requested on a node, the node is removed from the *cluster_list* and added to *local_list*.

***A*LGO_relax_dist** and ***A*LGO_relax_loc+dist:** Under the modes `alloc_relax_dist` and `alloc_relax_loc+dist`, Gerbil-AM first builds the *local_list* from the $hostfile$ or the specified number of nodes from the top of the *cluster_list* for *A*LGO_relax_dist and *A*LGO_relax_loc+dist, respectively. Next, processes are assigned to the set of nodes in a round-robin fashion, where

a process is assigned one $\{m, v\}$ resource at a time, and the process repeats until all of the *total_nproc* processes have been allocated.

***A*LGO_relax_all:** If the user only specifies the number of processes, GERBIL-AM employs round-robin process assignment using all the nodes in the *cluster_list*.

Under all algorithms, if any of the nodes have resources less than the minimum $\{m, v\}$, the algorithm is blocked until more resources become available. Whenever over-subscription is detected, *degraded* is marked as *true*. Finally, GERBIL-AM checks the *local_list* or *cluster_list* to combine multiple containers on the same node into one. This is done by adjusting the container size to make sure that only one aggregated container is requested per node per MPI job.

Since the algorithms are based on a snapshot of the available cluster resources, it is possible that the availability changes after a request has been issued but not serviced. The request can stall due to unavailability of actual resources. In order to reduce such wait time, we do not change the containers that are already allocated. Rather, we issue new requests for the remaining containers and wait until all new containers are allocated before launching the MPI job.

6.3.5 Discussion

Supporting HDFS I/O GERBIL supports any distributed file system that is compatible with MPI. For clusters that are not equipped with a distributed file system, GERBIL supports reading and writing of data from direct attached storage by transparently transferring data between local storage and the default YARN distributed file system, HDFS. Moreover, alternative shared file systems such as fuse-dfs [5], HDFS NFS Gateway [20], GPFS [167], and GFS [91] can be also employed. Users can also leverage the HDFS native library to handle I/O in their programs directly. Nevertheless, it has been shown [77] that maintaining a conventional distributed shared file system on a commodity PC cluster, i.e., resources that typically support YARN, is operationally cheap compared with the performance cost of HDFS.

Handling Failures Failures are the norm and not an exception in large-scale clusters. Thus, it is crucial to understand the failure behavior of our approach. In case of GERBIL-AM failure, YARN automatically relaunches the application master. Upon relaunch, GERBIL-AM requests the needed containers again and re-starts the MPI application automatically. This would be a heavyweight process and akin to cluster failure, e.g., due to power or networking loss, in traditional MPI setup. However, a more common failure is that of a container.

Container failure can lead to the failure of the hosted MPI job. GERBIL leverages built-in MPI fault tolerance to handle such job failures. There are a number of efforts in the MPI community to provide fault tolerance in case of process failure or communication failure,

and help MPI programmers achieve resilience both at the application level and at the MPI framework level. To this end, techniques such as group communication can be used to write a robust MPI application [99]. MPI/FT [62] considers task redundancy to provide fault tolerance, while FT-MPI [97] adopts a dynamic process management approach. Moreover, different MPI implementations deliver different fault tolerance mechanisms. Users can register various built-in error handlers and customized handlers in different MPI implementations. For communication failures, techniques such as message re-transmission, message logging and automatic message rerouting have been developed [58], and can be used alongside Gerbil as needed.

Finally, checkpointing is widely supported for MPI job abort/recover in different MPI implementations [16, 21]. Gerbil supports checkpointing with different MPI implementations. For example in MPICH, Gerbil configures MPICH to checkpoint with BLCR [2]. Both manual and automated checkpointing with an interval can be configured in Gerbil. Although failures are handled by MPI, Gerbil-AM provides automated job restart without user intervention. Gerbil-AM first communicates with RM and checks whether sufficient number of containers are available based on the container allocation mode. Unless under heavy cluster load, the AM requests containers with the same size to replace the failed containers, after which the jobs are restarted.

6.4 Implementation

We have implemented Gerbil with YARN Hadoop-2.2.0 and MPICH-$v1.1rc1$, though Gerbil can be easily extended to use other MPI implementations. For job submission, we extended the YARN standard client object, YarnClient, to Gerbil-Client that passes user specified arguments to YARN RM for launching Gerbil-AM.

Gerbil-Container

The Gerbil-Container encapsulates our MPI-specific handling mechanism and is responsible for starting the MPI MPD. The MPD in turn retrieves the appropriate CLC, and starts the MPI processes. We monitor the MPD through YARN NM and inform Gerbil-AM when the job is complete and the results are available.

Gerbil-AM

Gerbil-AM is responsible for requesting, allocating, and managing the containers, initiating the MPI runtime environment and launching MPI jobs. Figure 6.3 shows the different components of Gerbil-AM and their interactions. Gerbil-AM communicates with all of

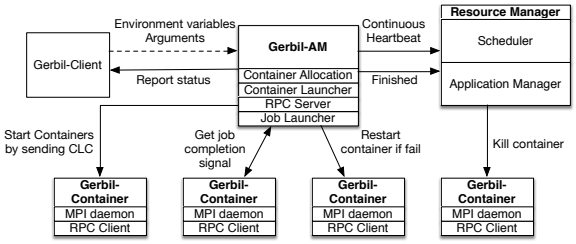


Figure 6.3: GERBIL components and their interactions.

the associated containers via RPC. To this end, an **RPC Server** is run within the GERBIL-AM and each of the launched container runs an RPC client. Communication such as when containers are ready for execution, when MPD has been launched/completed, and when a job is completed, is exchanged via the RPC Server. The **Container Allocator** implements the container allocation algorithms described in Section 6.3.4 to request needed containers from the YARN RM. The **Container Launcher** initiates the containers with the requisite execution environment, process/executable dependencies such as classpath, the application launching command, and arguments included in the CLC. The container launcher then sends the CLC to an appropriate NM and starts the associated GERBIL-Container. The **Job Launcher** starts the MPI job after GERBIL-AM has received the allocated containers and process assignments, using the following command:

mpiexec -hosts <number of host nodes> <host n ><number of processes> ... <mpi application> <-other options>

Upon job completion, GERBIL-AM collects the results (and writes them to HDFS), and returns the control back to GERBIL-Client, thus completing the process.

6.5 Evaluation

6.5.1 Experimental Setup

We evaluate GERBIL using a 19 node (1 Master, 18 slaves) test cluster [25], where each node contains two Intel Quad-core Xeon E5462 2.8 GHz processors, 12 MB L2 cache, 8 GB memory and one 320 GB Seagate ST3320820AS_P SATA disk. Nodes are connected using 1 *Gbps* Ethernet. The GERBIL-AM component is configured to use 3072 MB of memory and 2 cores. Each of the slave nodes has 6144 MB memory available for containers, with a minimum allocation of 1536 MB and 1 core. Thus, each node can have 4 containers, for a total of 70 containers available in our setup. We use MPICH version 1.1rc1 with NFS for native MPI setup, and Hadoop-2.2.0 for GERBIL with NFS to support MPI-IO where needed in addition to HDFS. In the following, we conduct each experiment four times and report the average values.

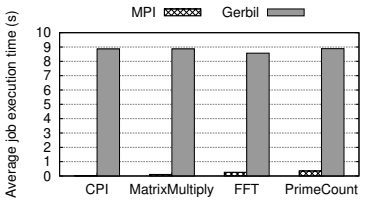


Figure 6.4: Job execution time for small MPI applications.

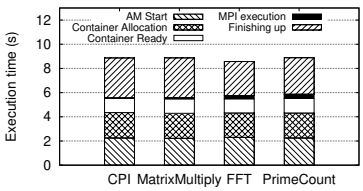


Figure 6.5: Execution time breakdown for small MPI applications.

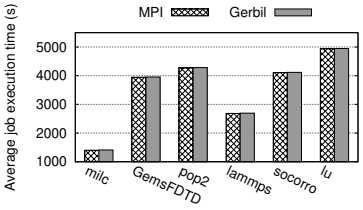


Figure 6.6: Job execution time for large MPI applications.

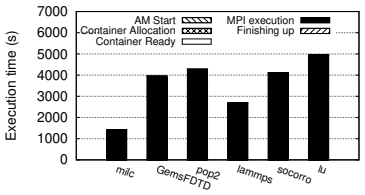


Figure 6.7: Execution time breakdown for large MPI applications.

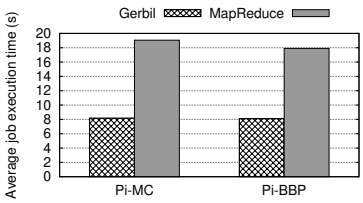


Figure 6.8: Performance of Gerbil MPI vs. MapReduce application versions.

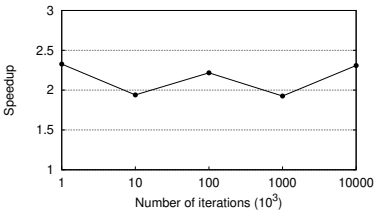


Figure 6.9: Speedup of Gerbil MPI over MapReduce Pi-MC versions with increasing number of iterations.

6.5.2 Performance Comparison of Gerbil and Native MPI

In our first experiment, we compare the performance of Gerbil against native MPI. Here, we run only one job at a time to isolate the performance differences. For each application, we launch 18 containers using `alloc_relax_none` mode in Gerbil with one container per node and one process per container. Correspondingly for native MPI, we run 18 processes with one process per node.

First, we use MPI example programs CPI, MatrixMultiply, 2D-FFT and PrimeCount, which take small input and have relatively small execution time. For MatrixMultiply and 2D-FFT we use randomly generated input matrices, while CPI and PrimeCount need no input. Figure 6.4 shows the comparison of running these *small* applications using Gerbil versus native MPI cluster. We observe that Gerbil imposes a significant overhead (about 9 seconds) for all of the applications. To further understand this, Figure 6.5 shows the detailed execution time breakdown of the observed execution time overhead.

The ***AM Start*** phase involves job admission control, scheduling and container allocation for Gerbil-AM. Since we have only one job running at a time, this metric measures the RM overhead of YARN. The ***Container Allocation*** phase involves container negotiations between Gerbil-AM and the RM (dictated by 1 Hz heartbeat messages of YARN) as well as the container allocation time. The ***Container Ready*** phase involves container lease

verification with the NM, configuring and instantiation of GERBIL-Container, loading the appropriate CLC, and starting the MPI daemon process. ***MPI Execution*** is the actual MPI job execution time. Finally, ***Finishing up*** shows the time used for cleaning up the containers and completion of GERBIL-AM and the job. We observe that while the MPI execution time is similar under GERBIL and native MPI, it is the YARN functions that incur the overhead.

Next, we repeat the experiment using six *large* MPI applications from the SPEC MPI 2007 benchmark [23]. These applications conduct large simulations from different domains, and have small input size (less than 50 KB) but employ complicated simulation algorithms. Figure 6.6 shows the results, which for this case shows only 0.3% average overhead across the applications under GERBIL compared to the native MPI. We also studied the breakdown for these applications as before, and as seen in Figure 6.7, compared to the MPI execution time the overhead is negligible.

These experiments show that the overhead of GERBIL is mainly due to YARN and would depend on the number of containers requested and not on the duration of the job. As discussed in Section 7.3, GERBIL only allocates one container per node for hosting the MPI tasks, and thus the overhead is fixed (about $9s$ for our setup). Consequently, for the typical long-running MPI jobs, the overhead of GERBIL versus native MPI is amortized across job duration and becomes negligible.

6.5.3 Performance Comparison of Gerbil and MapReduce

In our next test, we compare GERBIL performance to MapReduce by running two compute-intensive applications: Pi calculation using Monte-Carlo (Pi-MC) and Bailey–Borwein–Plouffe (Pi-BBP) algorithms with the same input parameters. we implemented the algorithms using an MPI version for GERBIL and a MapReduce version. We use 18 processes or mappers for each job in out test. Figure 6.8 shows that compared to MapReduce, GERBIL improves the average job execution time by 133% and 121% for Pi-MC and Pi-BBP, respectively.

Next, we selected the Pi-MC application and repeated the experiment with increased computational load by increasing the number of iterations. Figure 7.6 shows the job execution time under the MapReduce version normalized to that of MPI on GERBIL. We observe that GERBIL achieves a speedup of up to 2.3×. These tests illustrate that, as expected, GERBIL offers a better alternative than MapReduce for compute-intensive jobs.

6.5.4 Performance Impact of Gerbil Container Allocations

In our next set of experiments, we study the container allocation time and application execution time under the five algorithms of Section 6.3.4 in a multi-tenant environment. We carefully devise this test to investigate the numerous factors that impact these times, e.g.,

	Group A		Group B		Group C	
Nodes	n1	n2	n3	n4	n5	n6
BG1	4	4	0	0	0	0
BG2	0	0	4	4	0	0
Pi-MC	3	3	3	3	2	2

Table 6.3: Container allocation distribution with backgroud jobs BG1, BG2 and Pi-MC MPI job.

current cluster utilization, input data size, job arrival, etc. We first partition the cluster nodes into three groups Group A, B and C with 8, 6, and 4 nodes, respectively. We then launch a background job, BG1, with an approximate running time of 6 minutes on Group A, and BG2 (2 minute) on Group B. The background jobs can be MapReduce jobs, MPI jobs, or other jobs that run alongside the target MPI application. The background jobs consume all the containers in Groups A and B. Next, we wait for 5 seconds after the submission to ensure that BG1 and BG2 have started running within the containers. Then we submit one MPI application, Pi-MC (Section 6.5.3) to the cluster, using one of the five different allocation modes. Each instance requests 16 total containers and two nodes per Group, with 3, 3 and 2 containers per node from Groups A, B and C, respectively. Table 6.3 shows the allocated container distribution on 6 of the 18 nodes for Pi-MC. We repeat the experiment for five times running Pi-MC with different allocation modes.

Figure 6.10 shows the container allocation and job execution times. Pi-MC has to wait for BG1 to finish before running under both `alloc_relax_none` and `alloc_relax_dist` modes, as no resources are available in Group A. Similarly, even though 4 nodes can be allocated in Group C under `alloc_relax_loc`, the job has to wait until the remaining 2 become available in Groups A or B. Thus, the waiting time is non-deterministic depending on other jobs in the system. A similar behavior is observed under `alloc_relax_loc+dist` as well. Finally, as expected, the job is able to use all of the available resources in Group C under `alloc_relax_all` and exhibit the fastest performance. Note that the execution time of Pi-MC is the same under all cases. This is because this application is not sensitive to locality and there is no oversubscription. For applications that are sensitive to such factors, the five algorithms provide useful trade-offs to minimize the end-to-end application execution time, e.g., increasing allocation time to reduce execution time via better locality.

6.5.5 Impact of Oversubscription

In the next set of experiments, we investigate the performance penalty of oversubscription. For this purpose, we request an increasing number of processes under `alloc_relax_none` and force oversubscription by asking for all of the processes on a single node. We compare the results with the same requests under `alloc_relax_all`, where the requests are distributed to different nodes and no oversubscription occurs. Figure 6.11 shows the impact on Pi-MC

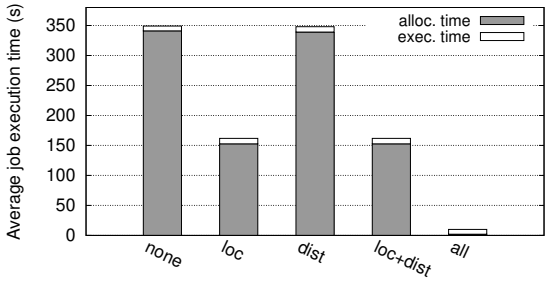


Figure 6.10: Total job execution time under studied allocation algorithms.

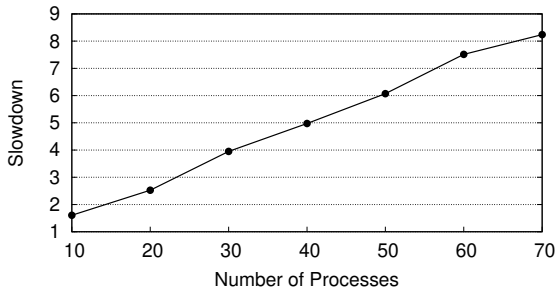


Figure 6.11: Slowdown due to oversubscription with increasing number of processes, normalized to `alloc_relax_all`.

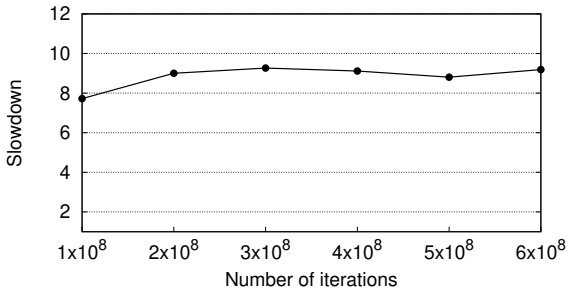


Figure 6.12: Slowdown due to oversubscription with increasing number of iterations, normalized to `alloc_relax_all`.

execution time as the degree of oversubscription is increased, and shows a linear relation as more and more processes content for the same resources under `alloc_relax_none`.

Next, we repeated the experiment with the max of 70 processes as the computation is increased via increased number of iterations. In Figure 6.12, we observe that the slowdown of oversubscription remains unaffected (approximately 8.8×) by the computation load, and depends only on the number of competing processes.

YARN does not implement oversubscription for MapReduce. In case a MapReduce job has more tasks than the available capacity, it is queued until all the needed resources become available, even though such gang scheduling is not required for MapReduce. GERBIL mitigates this by employing oversubscription. In our next test, we study the impact of oversubscription on reducing allocation waiting times by comparing GERBIL with MapReduce. We run both the MPI GERBIL and MapReduce versions of Pi-MC with 10 K iterations with 70 containers, i.e., the max in our setup. Figure 6.13 shows the results with increasing number of requested containers/processes. We see that while GERBIL immediately launches the required processes by oversubscribing them, MapReduce is unable to run all the mappers, instead running them in waves. Consequently, the execution time increases linearly under MapReduce but is effectively constant under GERBIL.

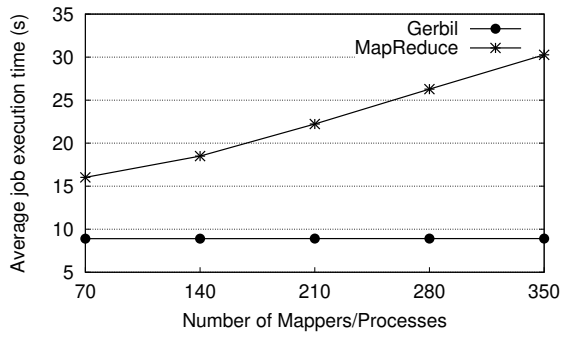


Figure 6.13: Gerbil vs. MapReduce execution time under oversubsciption.

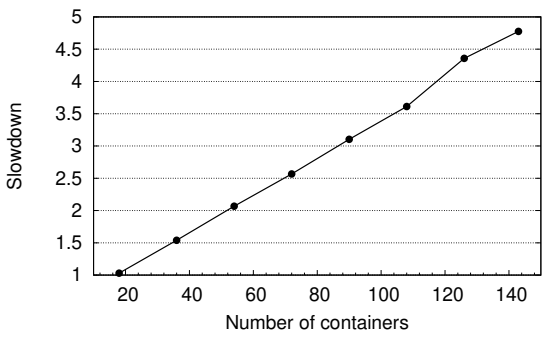


Figure 6.14: Container allocation slowdown of YARN RM normalized to `alloc_relax_all`.

6.5.6 Impact on Application Launch Time

In our next experiment, we investigate the container launch overhead of the default YARN RM container allocator compared to Gerbil using `alloc_relax_all`. For this purpose, we request an increasing number of containers each with 100 MB memory and 1 CPU core to a max request of $143(8\times18-1)$ containers possible on our setup. For each request, we measure the time it takes for the allocation to complete under the two cases. Figure 6.14 shows the slowdown in the YARN RM allocation normalized to Gerbil under `alloc_relax_all` as the number of requested containers is increased. Since the default container allocator works at the rate of heartbeat messages and can only allocate 18 containers at one time,—i.e., one container request per heartbeat to each of the 18 nodes in our setup—the allocation time increases linearly as more containers are requested. In contrast, Gerbil needs only one container per node to run multiple MPI processes; it can always allocate the needed containers in one heartbeat regardless of the number of processes. Thus, our design provides for mitigating long launching times as long as the user does not request additional constraints, i.e., under `alloc_relax_all`.

6.6 Chapter Summary

Co-hosting multiple programming models on the same resources helps avoid expensive data movement between clusters and greatly reduces workflow processing time. In this paper, we present Gerbil, a framework that supports the MPI programming model on YARN and enables execution of unmodified MPI applications on YARN managed resources alongside MapReduce applications. Our evaluation shows that the overhead of Gerbil is mainly due to the YARN framework and is observed to be constant regardless of the size of the applications. This is promising, as the overhead will become negligible when amortized over long-running applications. In our future work, we plan to extend Gerbil's static container management to consider dynamic cluster usage and runtime application profiling.

We are also exploring techniques to reduce container allocation times of YARN to help mitigate the observed MPI instantiation latency. In our future work, we aim to enhance GERBIL to automatically select the best allocation strategy for applications based on their characteristics.

Chapter 7

A Heterogeneity-Aware Task Scheduler for Spark

7.1 Introduction

Modern computer clusters that support big data platforms such as Spark [218] and Hadoop [31] are increasingly heterogeneous. The heterogeneity can arise from nodes comprising out-of-core accelerators, e.g., FPGAs, or staged upgrades resulting in nodes with varying performance and capabilities. Similarly, the resource needs of today's applications are also heterogeneous and dynamic. Modern applications in machine learning, data analysis, graph analysis, etc., can have varying resource requirements during the application lifetime. For instance, a machine learning job may be I/O-bound and need I/O resources for input data processing in the initial stage, and then be memory- and compute-bound during the later processing stages. However, big data platforms, e.g., Spark, are typically oblivious of the dynamic resource demands of applications and the underlying heterogeneity, as the unit of per-task resource allocation is a homogeneous container (i.e., the abstraction does not capture heterogeneity of resources such as CPU cores, RAM, and disk I/O). This fundamental mismatch between the high level software platforms and the underlying hardware results in degraded performance, and wastage of resources due to inability to efficiently utilize different resources.

We focus on the Spark scheduler in this paper, as Spark has become the de facto standard for big data processing platforms. Similar to most task schedulers in the MapReduce framework, the Spark scheduler mainly considers data locality for scheduling, and does not differentiate between the various resource capabilities, e.g., CPU power between the nodes, assuming them to be uniform, and is not aware of other resources that a node may have such as GPUs and SSDs. Varying resource demands within an application are also not captured. While, some external resource managers such as YARN [186] and Mesos [103] have begun to support

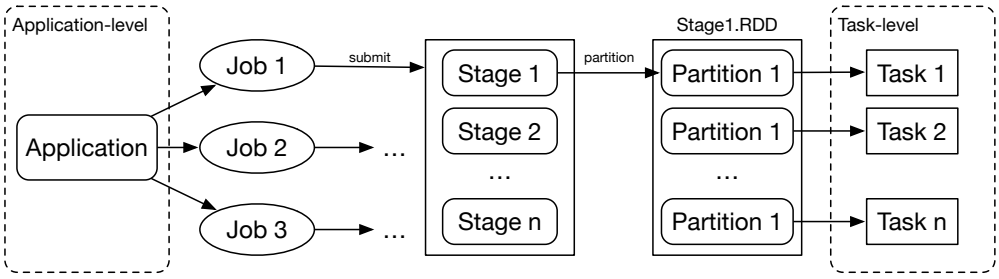
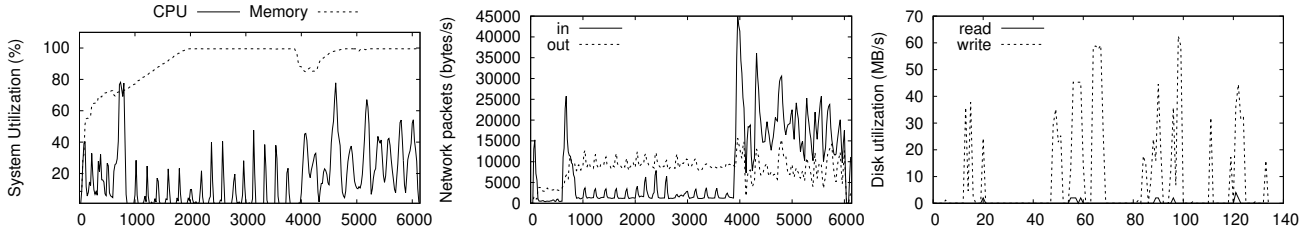


Figure 7.1: Spark application architecture.



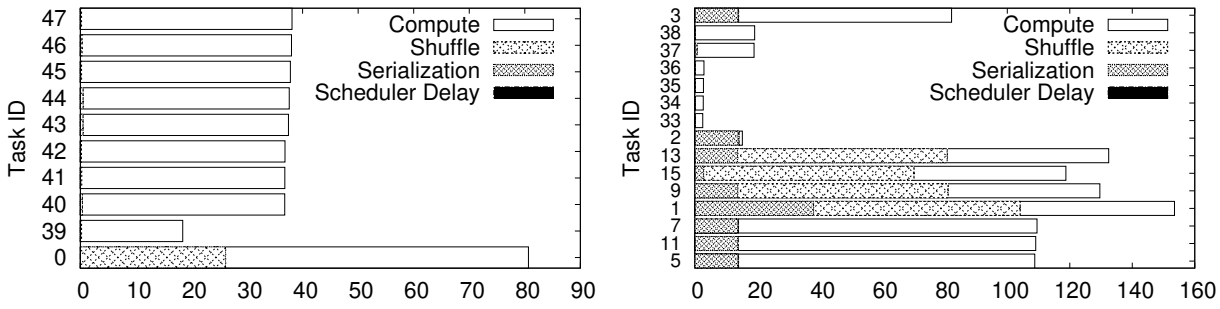
(a) CPU and Memory (b) Network (c) Disk

Figure 7.2: System resource utilization under $4K \times 4K$ matrix multiplication. X-axis represents relative timestamp (trimmed for Disk performance).

heterogeneous clusters, these external managers are not aware of the internal heterogeneity of tasks within applications, and rely entirely on applications to request the right resources for further second-level (i.e., task-level) scheduling. This needlessly burdens the application developer. Other approaches either require historical data for periodical jobs [112] or require job profiling [82, 88, 189]. Moreover, they do not count the emerging accelerators and storage devices. Note that this paper focuses on internal task-level scheduling, which is orthogonal to aforementioned job-level resource managers.

Extant approaches [36, 98, 192] often make the assumption that tasks perform generic computations, and tasks in the same Map/Reduce stage would have same resource consumption patterns. As a result, such approaches focus on general purpose CPU architectures and often optimize for a dominant resource bottleneck for tasks in a Map/Reduce stage, regardless of the differences among tasks in a single stage, as well as the tasks that may benefit from special devices such as GPUs [207]. A recent study [156] shows that a task in Spark requires multiple resources when executing, and the completion time of a stage depends on the time spent on each resource. Consequently, a scheduler should consider the heterogeneity of the resource consumption characteristics of tasks in an environment with heterogeneous hardware, as disregarding such factors leads to suboptimal scheduling and overall inefficient use of resources.

In this paper, we address the above problems and propose RUPAM, a heterogeneity-aware task



(a) node-1 (b) node-2

Figure 7.3: Task distribution and execution breakdown of PageRank.

scheduler for distributed data processing frameworks. Rupam considers both heterogeneity in the underlying resources as well as the various resource usage characteristics of each task in each stage of an application. Rupam manages the life cycle of all tasks inside a job. To this end, Rupam uses a self-adaptable heuristic that does not sacrifice data locality, to efficiently match a machine with the right task that can benefit most from the machine's resources in a dynamic fashion. The heuristic coupled with real-time resource utilization, enables Rupam to avoid resource contention and overlapping tasks with different resource demands, and thus yields high overall performance.

Specifically, this paper makes the following contributions.

1. We present a motivational study to demonstrate the diverse resource utilization and dynamic characteristics of tasks in Spark applications and show how the current Spark task scheduler is unable to properly handle heterogeneity in resources and application demands.

2. We design a self-adaptable heuristic based algorithm to automatically schedule tasks based on task characterization, cluster node capabilities, runtime resource utilization, and data locality.

3. We implement Rupam atop Spark and evaluate Rupam using SparkBench [130]. Results show that compared to the current Spark task scheduler, Rupam reduces workload execution time by up to 2.5×. The performance increases (up to 3.4× in our tests) with more application iterations (common in emerging deep learning applications).

7.2 Background and motivation

In this section, we first discuss the standard scheduling process of Spark. Next, we present an experimental study to motivate the need for Rupam.

7.2.1 Spark scheduling

As shown in Figure 7.1, a typical Spark application comprises multiple "jobs" triggered by various actions on Spark Resilient Distributed Dataset (RDDs). A job is further divided into multiple "stages" differentiated by shuffle dependencies, which perform the data transformation operations for an RDD. The "tasks" within a stage perform the same operation on different data partitions. Here, two levels of scheduling are performed for a Spark application, namely, application level scheduling that provisions nodes in a cluster for an application, and task level scheduling.

Application scheduling is done by cluster managers, either the internal Spark standalone cluster manager or external managers such as Mesos and YARN. The cluster manager provisions a subset of cluster nodes for an application, which Spark then uses to run jobs and tasks. Spark and major resource managers provision resources into abstract units of number of CPU cores and sizes of memory. However, other resources of a node, e.g., storage setup, network configuration, accelerators, etc. are not considered in the scheduling decisions. Mesos and YARN do provide support for accelerators, e.g., GPUs, but that too is limited and requires user configuration and labeling. Moreover, such configuration-based approaches are static throughout the application life cycle and unable to capture the dynamic application requirements. The main reason for this limitation is that the resource managers are not aware of the application task characteristics, and programmatic approaches need to be employed to negotiate and renegotiate between application and resource managers to dynamically require resources only when needed.

Once the resources are acquired from the resource manager, task-level scheduling is performed to assign the tasks to each executor running on a node. Current Spark task schedulers assign one task per a CPU core. Given a node, as long as it has cores available, the scheduler finds a task that has data residing on that node, and schedules the task to the node. Thus, only data locality is considered, while other factors are ignored. For example, a node with available cores may not be suitable for running more tasks say because it does not have enough memory left, and a new task would fail with an out of memory error. Existing heterogeneity-aware schedulers only focus on the heterogeneity of the underlying nodes, and often assume that all tasks in the same Map/Reduce stage have the same characteristics, which as we show does not always hold true.

7.2.2 Motivational study

To show that the assumptions in current Spark scheduler are not always true and can lead to inefficiencies, we performed several illustrative experiments with Spark. We use a simple 2-node setup, each node having 16 CPU cores and 48 GB memory. However, we configured the nodes with different CPU frequencies and network throughputs to emulate a heterogeneous environment. We explore more heterogeneous resources including memory, storage, and

accelerators in Section 7.3.3. We configured node-1 to have 1.6 GHz CPU frequency and 10 GbE network speed, and node-2 to have 2.4 GHz CPU frequency and 1 GbE network speed. We set up the latest version of Spark 2.2 with one master and two workers.

Resource utilization of different stages in a single application

In our first test, we study the dynamic demands for resources during an application's execution. Here we employ a crucial kernel used in numerous machine learning applications [207, 221, 222, 223], matrix multiplication, to make our case. We multiply two $4K \times 4K$ matrices as input and monitor the resource utilization during the execution. Figure 7.2 shows the CPU and memory utilization, network utilization for both inbound and outbound traffics, and disk utilization for both read and write. We observe that memory utilization remains high with an initial slope and a slight reduction in the final stages. In contrast, CPU usage is high for only the last stages where the actual multiplication happens, and shows a spike in the beginning data processing stage. Network utilization shows spikes in both beginning and final stages due to the reduce operations. The application exhibits relatively low disk reads but high disk writes during different shuffle stages.

We see from the graph that the matrix multiplication application requires multiple resources including CPU, memory, network and storage to execute, and the demand for different resources changes with the different execution stages. For example, it is CPU dominant in the beginning stages, memory dominant in the middle stages, and finally network dominant. Statically allocating a subset of cores and memory to a given application, as is the case in current schedulers, does not consider this diversity and inconsistency in the needed resources.

Task skewness in a single stage

Existing heterogeneity-aware schedulers assume tasks in the same stage to have similar characteristics as they perform the same computation. However, this assumption does not hold true due to data skewness, shuffle operations, etc. To demonstrate the diverse task characteristics in the same stage, we perform a PageRank calculation with 20 GB input data on the 2-node cluster. Figure 7.3 shows the task assignments on the two nodes. We further break down the execution time into four categories: compute, shuffle, data serialization, and scheduler delay. Here y-axis represents task ID, and x-axis represents the execution time in seconds. Note that node-1 has a higher CPU processing capacity and lower network throughput than node-2. We can see that tasks in the same stage have different execution times, with the difference being as much as 31× between the two nodes. Also we can see some tasks, such as task 47, are CPU intensive where they spend most of time on compute, while other tasks, such as task 13, are shuffle intensive. However, Spark task scheduler does not consider the characteristics of the tasks and assigns most CPU intensive tasks to node-1, and shuffle intensive tasks to node-2. It also does not consider overlapping tasks with different resource

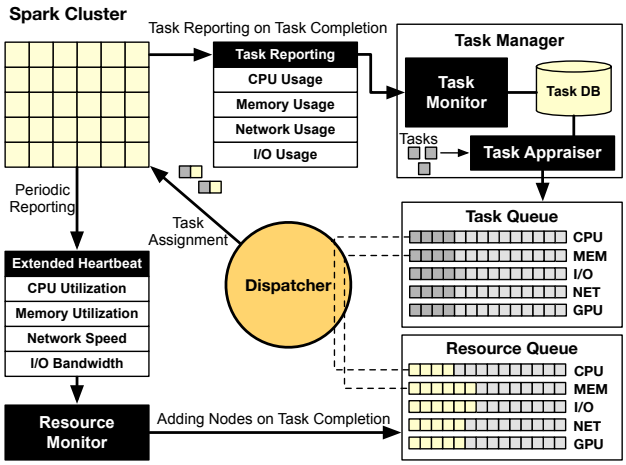


Figure 7.4: RUPAM system architecture.

demands on the same node. For example, most tasks in node-1 are compute intensive, and compete for CPU resource. Moreover, node-1 has 10 tasks assigned while node-2 has 15 tasks. The uneven task scheduling can also cause unbalanced load among the cluster.

These experiments show that, a Spark application may require varying resources even within the application life cycle. The tasks therein also have varying characteristics. Moreover, the resource heterogeneity amplifies the challenge of matching resources to appropriate tasks. RUPAM aims to manage this heterogeneity in an efficient manner.

7.3 Design

In this section, we first describe the architecture of RUPAM. Then, we detail the heterogeneity-aware scheduling of RUPAM.

7.3.1 System architecture

Figure 7.4 shows an overview of the RUPAM architecture, and highlights key components in addition to the original Spark cluster and the interactions therein. The goal of RUPAM is to match the best resource with a Spark task, given a heterogeneous resource pool allocated to the application. Note that RUPAM is a task-level scheduler that works with any application-level and job-level schedulers such as Mesos, YARN, or Spark standalone scheduler. In this work, we showcase RUPAM atop Spark standalone scheduler.

RUPAM has three major components: **Resource Monitor (RM)**, **Task Manager (TM)**, and **Dispatcher**. **RM** is in charge of real-time resource monitoring of the system. It has a

Node metrics	Description
cpuFreq	CPU clock frequency.
gpu	The number of idle GPUs in the machine.
ssd	The disk that Spark uses for intermediate data storage is an SSD device or not.
netBandwith	The bandwidth of the network.
freeMemory	Size of free memory in the node.
cpuUtil	CPU load of the node.
diskUtil	The I/O load of the node.
netUtil	The network load of the node.

Table 7.1: The node metrics that **RM** monitors on each node.

Task metrics	Description
computeTime	Time the task spent on computation (including serialization and deserialization).
gpu	Whether the task uses GPUs.
peakMemory	The maximum memory size used by the task during execution.
shuffleRead	Time the task spent on reading shuffle data.
shuffleWrite	Time the task spent on writing shuffle data to the disk.
optExecutor	The executor where the task has the lowest runtime by far.
historyResource	The history resource bottlenecks that **TM** has determined for this task.

Table 7.2: The task metrics that **TM** monitors on each node.

central *Monitor* running in Spark master and a distributed *Collector* running on each Spark worker node. The *Collector* reports resource usage such as CPU, memory, network, I/O, and GPU on each node, and the *Monitor* collects and records the information from the nodes. **RM** can be extended to collect more information based on the resource capabilities, e.g., NVM devices. **TM** tracks the tasks resource usage to determine any resource bottlenecks for a task. **Dispatcher** component implements the main logic flow of Rupam such as determining the size of executor to launch per node, number of tasks to launch on a specific node, matching a task to a suitable node, and scheduling tasks based on multiple factors.

RM starts when Spark is initiated on a cluster. When an application is submitted, Spark instantiates a centralized application-level resource scheduler that can be *CoarseGrained-Scheduler* for Spark standalone mode, and *ExternalClusterManager* for external resource managers. In the meantime, the Spark task scheduler (*TaskScheduler*) is also launched to take control of task life cycles. When tasks are submitted to *TaskScheduler*, it queues the tasks and waits for the resource scheduler to release a node. Then *TaskScheduler* schedules a task for a given resource based on locality. Here Rupam launches **TM** and **Dispatcher** instead of *TaskScheduler*. Instead of scheduling tasks based on locality alone, **TM** keeps track of resource utilization of each task and decides its crucial characteristics. The information is then used by **Dispatcher** to schedule task to suitable resources based on both resource

characteristics of tasks and nodes.

7.3.2 Real-time resource and task monitoring

The dynamic scheduling mechanism of RUPAM relies on real-time resource monitoring and task characterization. Specifically, when a node becomes available for running a task, RUPAM should be able to assign a task that has matching performance characteristics to the resource characteristics of the node (Section 7.2.2 showed tasks in the same stage with different resource bottlenecks). For instance, if the node is equipped with large amount of memory, a task requiring large memory capacity should be assigned to the node. However, the resource utilization of a node does not stay the same. Available resources can change as tasks are launching and finishing in the node. It is crucial to keep the most updated resource utilization for each node. Moreover, the resource bottleneck of the same task may change along with the status of the node it is executed on. For example, a task may have a bottleneck on CPU when executed on a node with poor CPU clock speed, but the same task may spend a large amount of time shuffling data over the network when it is executed on a powerful node (to remedy the CPU bottleneck) in the next iteration. The resource contention among tasks in a node can also affect such characteristics. Hence, it is also important to keep track of task characteristic in the entire life cycle to decide the best node for a given task with trade-offs. To this end, **RM** monitors resource utilization of each node real-time to provide updated utilization metrics, and **TM** keeps track of each task as the application executes and collects the statistics when the task finishes.

Resource monitoring

Spark adopts a master-slave deployment architecture. When a Spark worker launches, it registers itself with Spark master using a message including its ID, IP address, CPU cores, and memory. With this information, Spark master can later on launch executors with certain CPU cores and memory on each worker. To tolerate node failure, Spark master requires a simple heartbeat acknowledgement message to each worker periodically to decide whether a node is healthy. RUPAM takes advantage of this mechanism, and piggy-backs real-time resource monitoring data on the heart beats, thus providing scalable monitoring without introducing extra communication overhead. To consider heterogeneous hardware configurations in the cluster, RUPAM supports multi-dimensional resource characterization and availability reporting. Table 7.1 shows a list of supported monitoring metrics. For static properties such as SSD and maximum network bandwidth, *collector* only sends the information once when registering to the master. For real-time properties, RUPAM collects the information periodically via our extended heartbeat messages. We consider CPU frequency as a dynamic value due to the workload-aware energy saving features in modern CPUs. In addition, RUPAM can be easily extended to support other resource types.

RM records all of the collected information from the nodes and later on pass to the **Dispatcher**. Since every node has different types of resources as listed in Table 7.1, the key challenge is how to organize the metrics for different nodes as the system scales. To this end, RUPAM again leverages an existing object of *executorDataMap* inside Spark to reduce memory overhead. After receiving the heartbeat messages from the *collector*s, **RM** first stores all of the information in the *executorDataMap* object. However, just keeping the information does not help decide appropriate nodes for given tasks. Multi-dimensional resource availability complicates this process. To help **Dispatcher** figure out the best node for each resource category, RUPAM uses one priority queue for each resource type ("Resource Queue" in Figure 7.4), i.e., CPU, GPU, network, storage, and memory. Each queue is sorted with capacity in descending order (most powerful/capable/capacity first) and associated utilization in ascending order (least used first). In order not to overwhelm the memory usage by keeping these queues, we only insert a record whenever a node is ready to execute a task. Based on our observation, instead of making bulk scheduling when all nodes become available, Spark detects whether a node is ready for tasks with heartbeat messages and immediately schedules a task whenever a subset of nodes are available. It then blocks incoming messages until tasks are scheduled, before going to the next round of making offers. As a result, RUPAM only needs to sort out the small subset of nodes in a single round, and all of the queues can be emptied before the next round of offers. In this way, we keep the size of our resource queues in check and associated sorting time complexity low. This minimizes the overhead of RUPAM.

Task monitoring

To select an appropriate task for a given node with certain capabilities, RUPAM also needs to determine the task characteristics. To this end, **TM** monitors the resource usage of tasks for every application and records the information. RUPAM uses a task characteristics database (DB_{task_char}) to store the task metrics based on the observation that data centers usually run the same application on input data with similar patterns periodically [181, 188]. Table 7.2 shows the task metrics that RUPAM maintains. Once tasks are submitted to **TM**, RUPAM first searches for the task in DB_{task_char} and retrieves its characteristics. To separate tasks with different resource needs in a stage, **TM** also keeps a queue for each resource type ("Task Queue" in Figure 7.4). As stated earlier, these queues will be reset when current tasks finish and before a new task wave arrives. Algorithm 7 gives a detailed view of the steps **TM** takes to determine resource bottlenecks. Here, Res_factor is a parameter that decides the sensitivity to resource bottlenecks. For example, a task is considered compute-bound if it spends 2× more time than shuffle operation. Users can adjust the sensitivity, and RUPAM will modify the frequency of task re-characterization and re-scheduling accordingly.

If there is no record for a task in DB_{task_char}, i.e., this is the first time the task has been submitted, RUPAM first checks the current stage of the task. If it is in map stage (*ShuffleMapTask*), RUPAM considers it to be bounded by all types of resources and thus enqueue

it in all queues. A task in a reduce stage (*ResultTask*) is considered to be network bound, since a reduce task typically first reads data from shuffling and then sends the results back to the Spark driver program; activities that are all network intensive (this assumption can be relaxed by **TM** for later iterations.). When a task finishes, RUPAM combines the task metric information from Spark and records the information in DB_{task_char} for future use, i.e., future task iterations and job runs.

For tasks that use special accelerators such as GPUs, **TM** checks with **RM** to see if any GPU is used during the task execution period, and marks all the tasks in the same stage to be GPU tasks. This is because the tasks in the same stage usually perform the same computation. The **TM** updates the task metrics in DB_{task_char} whenever a task finishes. This is because the same task may show different resource usage when executing in a different environment as discussed earlier. This ensures that RUPAM has the most updated information for its decision making. However, this also creates the challenge of how to manage the overhead of frequent DB_{task_char} accesses. To address this, RUPAM creates a helper thread for accessing DB_{task_char}. All DB_{task_char} write requests are queued and served by the helper thread. For read requests, the helper thread first checks the queue to see if the task has written to the database yet, and if it has, the request is served from the enqueue requests if any before accessing the database.

7.3.3 Task scheduling

We model the problem as the scheduling of n tasks onto m parallel machines. Our objective is to minimize the total processing time of all tasks for all machines, $T_{max} = \max(T_i = \sum_{\forall j} p_j)$, where p_j denotes the processing time of a task j. T_{max} represents the maximal makespan among machines, which is equivalent to the makespan in parallel machines or the total processing time of all tasks for all machines. In order to capture the different hardware capability, capacity of nodes, and data locality, we assume that the processing time of the task j on each machine i, $p_{i,j}$, varies across machines. Our goal is to minimize $T_{max} = \max(T_i = \sum_{\forall j} p_{i,j})$ under the following constraints:

$$\forall i, \sum_{j} x_{i,j}^{r} \leq C_i^r$$
$$\forall i, j, x_{i,j} \in 0, 1,$$

where C_i^r denotes the capacity of resource r on machine i, and $x_{i,j} = 1$ if and only if a task j is scheduled on the machine i. If the resource r is not available on the machine i, $C_i^r = 0$, which prevents task j from mapping to machine i.

The relevant underlying theoretical problem that applies to our conditions is is unrelated parallel machine scheduling [175], for which obtaining optimal solution has been shown to be

Algorithm 7: Task characterization procedure

Input: $taskSet$, $computeTime$, gpu, $shuffleRead$, $shuffleWrite$, Res_factor

```
begin
    for each task in taskSet do
        if gpu then
            pendingGpuTasks.enqueue(task);
        end
        else if computeTime > Res_factors × max(shuffleRead, shuffleWrite) then
            pendingCpuTasks.enqueue(task);
        end
        else if shuffleRead > Res_factor × shuffleWrite then
            pendingNetTasks.enqueue(task);
        end
        else
            pendingDiskTasks.enqueue(task);
        end
    end
end
```

NP-hard [37, 73, 98]. The most popular solution to this problem is list scheduling algorithm, a greedy algorithm that maps tasks to available machines without introducing idle times, if it is not needed (e.g., dependencies between tasks.) Since list scheduling provides a practical solution with a reasonably good theoretical bound [175], RUPAM adopts a heuristics based on the greedy algorithm.

Specifically, the **TM** determines resource bottleneck for each task, and passes it to **Dispatcher** for scheduling. The **Dispatcher** waits for underlying node(s) to be available. Combined with the task metrics from **TM** and node information from **RM**, **Dispatcher** is able to match a task to a node. There are several factors that RUPAM needs to consider for heterogeneous resource aware scheduling:

- hardware capability/capacity of nodes (e.g. w/wo SSDs, w/wo GPUs);
- resource consumption for each task;
- resource contention for each resource in a node;
- number of tasks running in the same node; and
- data locality.

Algorithm 8: Task scheduling algorithm

```
Input: taskSet, speculativeTaskSet
begin
    {res, node} ←dequeue_node_rr;
    task ←schedule_task(taskSet, res, node);
    if task is null then
        /*check stragglers*/
        task ←schedule_task(speculativeTaskiSet, res, node);
    end
end
Function schedule_task(pendingTasks, res, node)
    taskList ←get_tasks_with_res(res, pendingTasks); taskWithBestLoc ← null;
    for each task in taskList do
        if task.peakMemory > node.freeMemory then
            /*the task has been identified to bottlenecked by all of the 5 resources
              and the history shows running on node yields best performance.*/
            if task.historyResource.size = 5 and task.optExecutor = node then
                return task;
            end
            else if task.get_locality(node) = PROCESS_LOCAL then
                return task;
            end
            else if task.get_locality(node)> taskWithBestLoc.get_locality(node) then
                taskWithBestLoc ← task;
            end
        end
    end
    return taskWithBestLoc;
```

Scheduling policy

Data locality based scheduling mitigates performance degradation due to network transfers, but is unable to capture resource and task heterogeneity as discussed earlier. In contrast, RUPAM uses multi-dimensional characteristics for scheduling. The **Dispatcher** matches the task with the right resources with the help of "Task Queue" and "Resource Queue". Algorithm 8 describes the steps **Dispatcher** takes to schedule tasks. After **RM** populates the "Resource Queue", RUPAM dequeues one node from each resource queue at a time in a round-robin fashion to make sure no task with a single resource type is starved. Since the first node dequeued from a specific priority queue will have the highest capacity/capability and the least utilization of the available resource type, **Dispatcher** goes over the tasks in the queue of that resource type, makes sure that the node has enough free memory to launch the

task, and finally find the task with the best locality to that node in the order of `PROCESS_LOCAL` (data is inside the java process), `NODE_LOCAL` (data is on the node), `RACK_LOCAL` (data is on the node in the same rack) and `ANY` (data on a node in a different rack). This heuristic greedily finds a node N with the best capability and lowest contention for a resource R, and then schedules to N a task T, that had R as the bottleneck in its previous run, and now N offers the best locality for T. RUPAM does not try to find the optimal scheduling strategy for each task, as that may cause a huge scheduling delay and will be counterproductive. Instead, RUPAM tries different node assignments for a task, e.g., with well-endowed CPUs or better I/O throughput, and records the node where the task has achieved the best performance. This node is used to schedule the task, even if the task has some bottleneck for say CPU on that node. This "locking" of a task to the node on which it gives the best observed performance, also prevents moving tasks back and forth between nodes due to temporary fluctuations in the task characteristics.

Resource allocation

Spark launches executors with a fixed number of cores and memory, and considers a node to be available if there are free cores in the node. This static configuration is inefficient. First, in a heterogeneous environment, nodes have different memory size and number of cores. RUPAM schedules tasks beyond this size limitation, i.e., based on the resource availability for each node. For example, RUPAM changes the executor size when necessary so that different nodes will have executors with different memory sizes. Second, determining the number of task slots based on the number of CPU cores in a node is not accurate. For example, a node with no free CPU cores may only have 10% CPU utilization if all tasks assigned the node are I/O bound. On the other hand, a node that only has one task may be using 100% of the CPU. In this case, scheduling more CPU intensive tasks will cause resource contention and slowdown the application progress. To this end, RUPAM treats a node to be available as long as it has enough resources to execute a task. The usage of priority queue makes sure that the node that is dequeued has the least utilization for such a resource. By over-committing a node that has some idle resources and matching the node with the right task, RUPAM can overlap tasks with different resource demands. For instance, a node that has all cores that are occupied by CPU intensive tasks, may have idle GPUs. It will be the first node in the GPU priority queue, and RUPAM can use it to run a GPU-friendly task. Thus making more efficient use of available resources. Such resource overlapping is possible because Spark often launches tasks from different stages at the same time whenever possible (as long as there are no data dependencies between the stages).

Straggler and task relocation

To remedy the possible suboptimal decisions **Dispatcher** may make, RUPAM also works with recently introduced Spark speculative execution system to launch copies of stragglers

Name	CPU (GHz)	Memory (GB)	Network (GbE)	SSD	GPU	#
thor	3.2	16	1	Y	N	6
hulk	2.5	64	1	N	N	4
stack	2.4	48	1	N	Y	2

Table 7.3: Specifications of Hydra cluster nodes.

in available nodes. The Spark speculative execution system monitors the current runtime of the tasks. When the number of tasks completed reaches a threshold (default 75% of total tasks), the system searches the tasks that take more time than a factor (1.5× by default) of mean execution time of finished tasks and mark them as stragglers. A copy of the stragglers will be executed in the next available node to compete with the original copy. Besides the standard stragglers detected by Spark, RUPAM also detects resource stragglers due to the heterogeneous environment. For this purpose, we change the `checkSpeculatableTasks()` function in Spark to consider resource usage when marking tasks as stragglers. For example, BLAS application is designed atop underlying libraries that either use CPU (OpenBLAS) or GPU (NVBLAS) for acceleration [207]. Although such task would be marked as GPU tasks in RUPAM, RUPAM does not wait until GPU nodes are available to execute the tasks, instead it will also schedule such tasks to available nodes with powerful and idle CPU. Whichever version finishes first will continue, while the unfinished version is aborted and the resources freed. On the other hand, memory is a crucial resource because *OutOfMemory* error can cause the application to abort. In the case when the OS rejects the JVM memory allocation request, the whole JVM can be killed by the OS, which is then followed by a catastrophic failure of the Spark worker. To prevent such a scenario, RUPAM also takes an aggressive approach to detect memory stragglers. First, whenever **RM** detects a node that has low free memory, it sends a message to **TM** and by examining the currently running tasks, **TM** marks the task that has the highest memory consumption as straggler, and terminates the task in the node. After marking a task as a straggler, a copy of the task is sent to **TM**, and it analyzes the task metrics to determine the bottleneck and enqueues it to the "Task Queue" again. The **Dispatcher** can then again assign the task to a node that has appropriate idle resources for the task.

7.4 Evaluation

We evaluate RUPAM using a local heterogeneous cluster, Hydra, consisting of 12 nodes with three types of resources as Table 7.3, namely thor, hulk, and stack. Each thor node has an 8-core AMD FX-8320E processor, and a 512 *GB* Crucial SSD. However, these nodes have the lowest memory capacity of 16 *GB* each. The hulk machines have 32-core AMD Opteron Processor 6380. The hulk machines have the highest memory capacity of 64 *GB*

Workload	Input size (GB)
Logistic Regression (LR)	60
TeraSort	40
SQL	35
PageRank (PR)	0.95 (500K vertices)
Triangle Count (TC)	0.95 (500K vertices)
Gramian Matrix (GM)	0.96 (8K*8K matrix)
KMeans	3.7

Table 7.4: Studied workloads and input sizes.

each and a high network bandwith of 10 *GbE*. Finally, stack machines have 16-core Intel Xeon E5620 CPUs and a moderate memory size of 48 *GB*. Each stack nodes is equipped with an NVIDIA Tesla C2050 GPU. All nodes besides thor have a 1 *TB* Seagate HDD device as storage. We have 6 thor nodes, 2 stack nodes, and 4 hulk nodes in our cluster. We set up Spark 2.2 with one master node and 12 worker nodes with the master running on a node that is also a worker. We set the executor memory size to 14 *GB* to accommodate the thor machines in our cluster for default Spark setup. We evaluate Rupam using a variety of applications (Table 7.4) covering graph, machine learning applications, SQL, and TeraSort from SparkBench [130]. For our evaluation, we also utilize a widely-used GPU-intensive application employed in machine learning algorithms, Gramian Matrix calculation [207]. Note that Gramian Matrix and KMeans utilize GPUs for acceleration.

7.4.1 Hardware capabilities

To study the hardware capabilities of each machine group in our heterogeneous cluster, we first use SysBench [118] to benchmark the CPU and I/O performance of a node from each group. We also use Iperf [8] to determine the real network bandwidth between the workers and the master nodes. Table 7.5 shows the results. For CPU tests, we use the default CPU test workload from SysBench that calculates 20 *K* prime numbers using all available cores. We record the time in seconds and latency in milliseconds. We can see that thor machines perform the best and are 5× faster than stack and hulk machines. Thor also has the lowest latency. Hulk machines performs slightly better than stack machines. We also run the default I/O tests with a 10 *GB* file, using direct I/O to avoid memory cache effect. We observe that the thor machines have the best read and write performance, given it has the attached SSDs. Finally, we use Iperf with master node (stack1) set as the server, and test the UDP (protocol used in Spark) performance from different set of machines. Since all machines are connected through a 1 *GbE* network, the results are similar for all the machines. In a large-scale environment, more complicated network topology would result in a more disparate network bandwidth availability among node in different subnets.

SysBench	stack	hulk	thor
CPU (sec)/latency (ms)	10.14/4.63	10.06/2.23	2.16/1.73
I/O read (MB/s)	63.87	148.92	236.69
I/O write (MB/s)	6.21	52.58	134.18
Network (Mbits/s)	809	934	813

Table 7.5: Hardware characteristics benchmarks.

7.4.2 Overall performance

In our first set of tests, we study the overall performance impact of RUPAM. For a fair comparison, we also enable speculative task execution (via `spark.speculation`) for default Spark in these tests. We run all workloads five times and clear DB_{task_char} after each run, and record the average execution time and 95% confidence interval under both default Spark and RUPAM. The workloads include both compute intensive (KMeans, GM, etc.) and shuffle intensive (SQL, TeraSort, etc.) applications. Some workloads, such as PR, are memory intensive such that default Spark fails with memory error in some runs. Figure 7.5 shows the results. We observe that all workloads experience performance improvement under RUPAM, with an average of 37.7% over default Spark. This is because RUPAM selects a node that offers the best resource for a task of each type of workload, while Spark only considers data locality, PR yields the highest speedup of 2.65×. This is due to the memory error failure and recovery during the execution (which also causes the large error bar for PR with default Spark). In contrast, RUPAM finishes without memory errors due to the dynamic executor memory configuration based on each node's memory capacity as well as the memory usage aware task scheduling policy discussed in Section 7.3.3. KMeans also achieves a 2.49× speedup, but GM only shows a negligible 1.4% performance improvement. This is because GM only has one iteration of computation, which makes it difficult for RUPAM to test and determine an appropriate resource for the workload, while KMeans' five iterations enable RUPAM to better match tasks with suitable resources. The behavior is also observed for other workloads with only one iteration such as SQL (per query) and TeraSort, which only have moderate speedups of 1.19× and 1.32×, respectively. Workloads with multiple iterations (PR, LR, TC, KMeans) have an average speedup of 2.31×. This shows that RUPAM performs better when there are multiple iterations in a workload due to the log based task characterization of RUPAM. The more iterations an application has, the better matching RUPAM can achieve for the application.

In order to have a clearer view of the relationship between the performance of RUPAM and the number of iterations of a workload, we experiment further with LR. Here, we use the same input size but alter the times of regression to vary the number of iterations of the workload. We record the speedup of RUPAM compared to default Spark in Figure 7.6. We can see that as the number of iteration increases, the speedup achieved by RUPAM also increases, up to 3.4×. Note that regardless of iterations, RUPAM is able to match or outperform the default

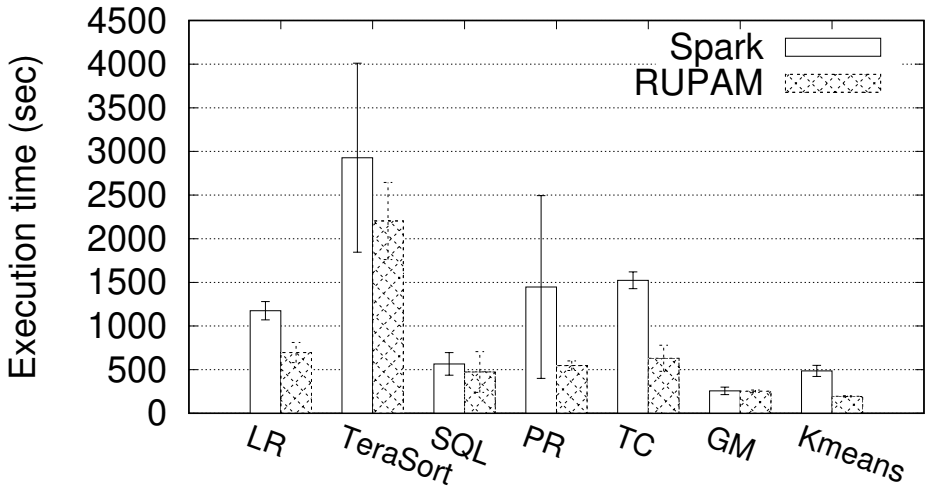


Figure 7.5: Overall performance of studied workloads under default Spark and Rupam.

Spark scheduler, thus making is a desirable approach for all types of applications.

7.4.3 Impact on locality

Rupam attempts to find the task whose requirements best match a node's resource, as well as will achieve the best data locality on the node. However, it is possible that a node N that offers the best locality (`NODE_LOCAL` and `ANY`) will affect a task T's performance significantly when considering other resources beside locality, and thus, unlike default Spark N will not be picked by Rupam for running T. In our next test, we record the number of tasks that are scheduled by Rupam on a node with different locality than a node chosen by default Spark. We use this number as a measure of Rupam's impact on preserving data locality similar to default Spark. Table 7.6 shows the results. Note that all workloads have zero `RACK_LOCAL` tasks. We can see that for all workloads, default Spark has more `PROCESS_LOCAL` tasks than Rupam. This is expected, as Spark aims to schedule task with the best data locality available for a node. For some workloads, such as TeraSort and TC, Spark has more total number of tasks than Rupam. This is due to the fail and retry of some tasks with the out-of-memory error under Spark when they are scheduled to a node with less memory capacity and high memory contention. In such cases, Spark still strives to allocate tasks with the best data locality, so we still observe a lower number of `NODE_LOCAL` and `ANY` tasks under such scenario. However, Rupam has more tasks with poorer data locality for such workloads because Rupam relocates tasks to a node with higher memory capacity and lower contention. Thus, Rupam trade-offs locality for better matching resources in such cases, with the goal to achieve higher end-to-end performance (in this even task completion and error avoidance). As the goal of any big data platform is faster time to solution, and not

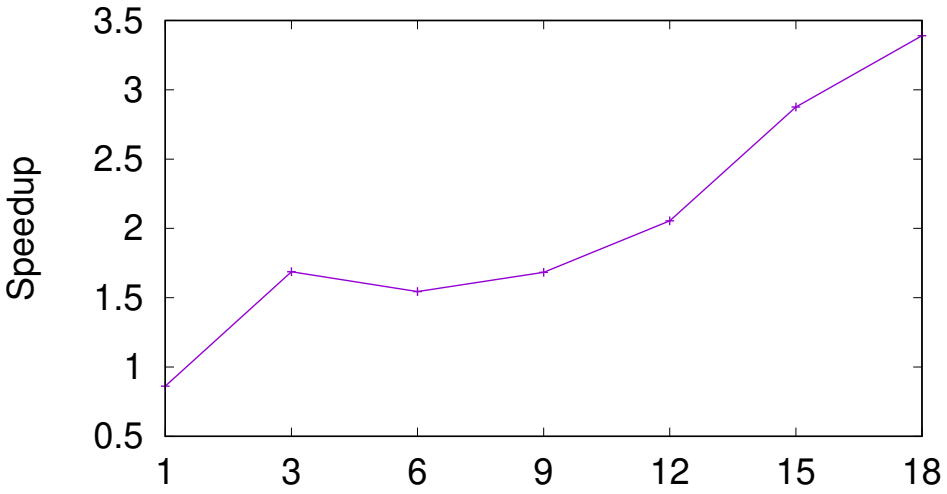


Figure 7.6: Speedup of LR under RUPAM wrt. default Spark with increasing number of workload iterations.

preserving locality for the sake of it, such trade-offs are justified and necessary.

7.4.4 Performance breakdown

In our next experiment, we select three representative workloads for each category—LR (machine learning), SQL (database), and PR (graph)—and study the breakdown of performance into five categories: compute, garbage collection (GC), network shuffle, disk shuffle (read and write), and scheduler delay. Figure 7.7 shows the results. We find that all selected workloads have improved compute times, which underscores RUPAM's ability to schedule compute-intensive tasks to nodes with better computational power and less CPU utilization to reduce contention (Section 7.3.3). For the LR workload, we observe less GC overhead for RUPAM, but similar or higher GC overhead for PR and SQL, respectively. Combined with Figure 7.8(b), we see that SQL consumes the largest amount of memory among the three studied workloads. Furthermore, SQL has only one iteration per SQL query with no data that needs to be preserved across queries, but involves a lot of shuffle operations for data join, so GC is triggered often to free space for shuffle. Moreover, RUPAM increases the memory usage up to the node capacity compared to a conservative configuration of Spark, thus resulting in JAVA spending more time to search the whole JVM memory space for GC, resulting in a big GC overhead compared to default Spark. On the other hand, LR has moderate memory usage and intermediate data needs to be kept across iterations. In this case, larger memory capacity provided by RUPAM is able to cache more data compared with the static memory configuration of default Spark where more GC operations are triggered for LRU cache management. Thus, RUPAM entails less GC operations and experiences a lower GC overhead.

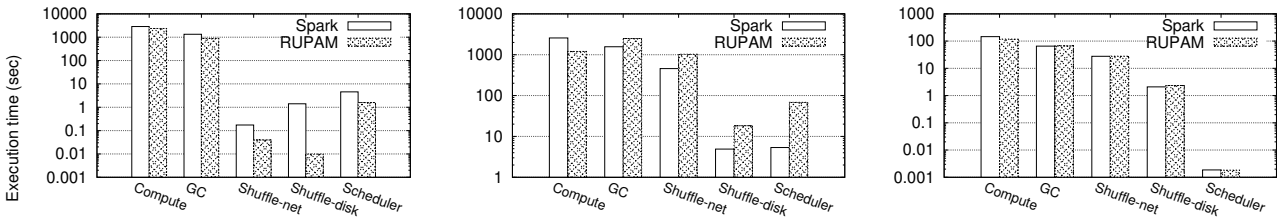
Workload	PROCESS		NODE		ANY	
	Spark	RUPAM	Spark	RUPAM	Spark	RUPAM
LR	295	292	12	15	0	0
TeraSort	2490	1803	339	188	30	12
PR	13800	13785	16	33	134	132
TC	2052	3924	612	1605	556	949
SQL	492	480	0	0	1	13
GM	512	256	64	320	0	0
Kmeans	2004	2259	95	101	10	0

Table 7.6: Number of tasks with different locality level under default Spark and Rupam.

For SQL workload, Rupam yields a high shuffle overhead than Spark. This is because SQL only has one iteration and Rupam simply treats the tasks to be general without specific resource bottleneck as described in Section 7.3.2. Moreover, Rupam achieves worse data locality compared to Spark as shown in Section 7.4.3, which result in worse shuffle performance. For other workloads, such overhead is mitigated by the performance improvement due to correct characterization of tasks. From Table 7.6, we see that for LR, Rupam has 15 `NODE_LOCAL` tasks and 0 `ANY` tasks, while Spark has 12 `NODE_LOCAL` tasks and 0 `ANY` tasks. Thus, we do not observe much shuffle overhead over network, but we observe a higher shuffle overhead from disk for Spark than for Rupam. This is because although Rupam has similar number of `NODE_LOCAL` tasks to Spark, Rupam schedules the I/O intensive tasks to nodes with SSDs. On the other hand, we see that Rupam has 13 `ANY` tasks but Spark only has 1. This creates the larger shuffle network overhead of Rupam. PR has a similar number of `ANY` tasks, but Rupam has twice the number of `NODE_LOCAL` tasks than Spark. However, here again we observe similar shuffle disk overhead of Rupam due to the I/O task scheduling of Rupam.

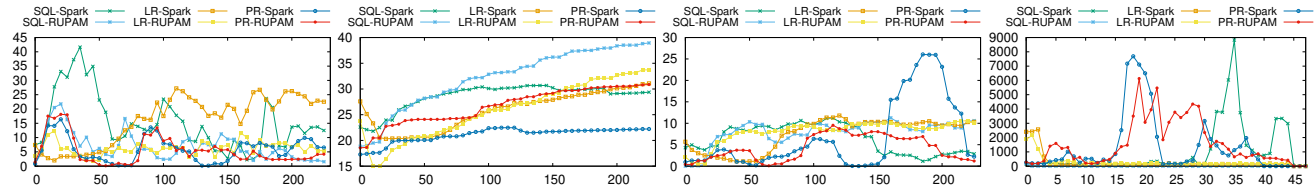
Finally, we observe that although Rupam takes more steps for task scheduling, by tracking resource and task monitoring, and using the simple heuristic, the resulting scheduler delay under Rupam is moderate compared to default Spark.

Discussion: We carefully select the input data size to fully saturate the capacity of the scale of our setup, such that the task would execute without crashes under default Spark. All task slots are filled during the experiments. We expect that bigger data size would generate more tasks to be scheduled in the queue. With the same hardware configuration, Rupam continues to strategically assign tasks based on real-time resource consumption of both tasks and the underlying nodes, while Spark's static task slot-based scheduling policy can cause suboptimal scheduling decision and resource contention. Though speculative execution can ease the problem, more frequent launching and relaunching of tasks also increase the scheduling overhead.



(a) LR (b) SQL (c) PR

Figure 7.7: Performance breakdown of selected workloads under default Spark and RUPAM. Note that y-axis is log-scaled.



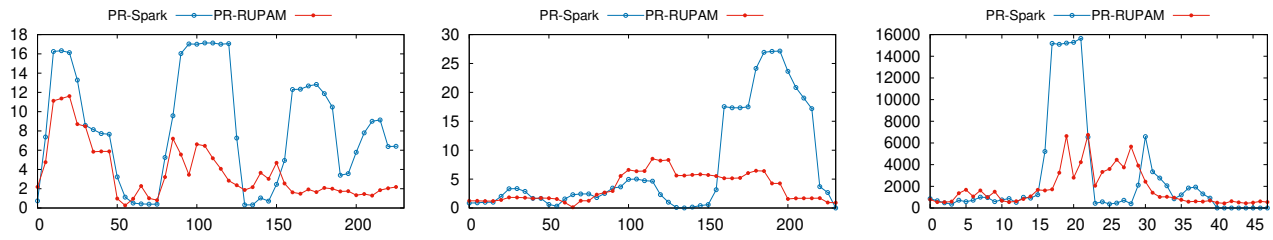
(a) CPU User (%) (b) Memory Used (GB) (c) Network Packets (MB/s) (d) Disk Utilization (KB/s)

Figure 7.8: Average system utilization of the nodes with selected workloads under default Spark and RUPAM.

7.4.5 Impact on system utilization

In our final test, we investigate how RUPAM impacts resource utilization. We repeat the previous experiment (with LR, SQL, and PR) and measure the average utilization of CPU, memory, network, and disk I/O of the 12 nodes in our cluster. Figure 7.8 shows the results. We see that for CPU utilization, RUPAM shows a lower average CPU user percentage compared with default Spark. This is because RUPAM takes the real-time CPU utilization into consideration when assigning tasks, which can help balance CPU load and reduce CPU contention. The same trend can also be observed for network and disk I/O utilization. However, for memory, RUPAM shows a higher usage than default Spark for all workloads. This is because default Spark takes a static global configurable memory size for launching the executors on each worker node. In our setup, we have to accommodate the node with the smallest memory size in order to launch the executors without memory errors. However, RUPAM is able to launch executors with different memory sizes on different nodes with different memory capacity. Thus, RUPAM yields a higher overall/average memory usage.

Next, to test RUPAM's impact on the resource load balance, we also calculate the standard deviation of the resource utilization among the nodes in the cluster during the execution of workloads. To get a clear view, Figure 7.9 only shows the result for PR. However, the other workloads show similar patterns. Here we omit the results for memory usage due to



(a) CPU User (%) (b) Network Packets (MB/s) (c) Disk Utilization (KB/s)

Figure 7.9: Standard deviation of system utilization among the nodes with selected workloads under the default Spark and Rupam.

the Rupam's design of using all available memory of the node. We observe from the figure that, in general, Rupam shows a lower standard deviation of CPU utilization than default Spark. For network and disk I/O, Spark shows spikes, while Rupam keeps a low and stable standard deviation. This is because PR performs heavy shuffle operations in the late stages, and stresses the use of network and disk. The low standard deviation among the nodes of Rupam shows that Rupam scheduling is able to dispatch tasks to nodes with less contention on the resources and balance the resource utilization among the nodes in a cluster, while Spark scheduler only considers data locality and may cause an unbalanced workload and contention on individual nodes, which results in a higher standard deviation in utilization among the nodes.

7.5 Chapter Summary

In this chapter, we present Rupam, a heterogeneity-aware task scheduling system for big data platforms. Rupam goes beyond just using data locality for task scheduling, and also factors in both task-level resource characteristics and underlying hardware characteristics including network, storage, and out-of-core accelerators in addition to the extant CPU and memory. Rupam adopts a self-adaptable heuristic for scheduling tasks based on the collected metrics without loss of data locality. Experiments with an implementation of Rupam atop Spark shows an overall performance improvement by up to 62.3% compared to the extant Spark task scheduler. In our future work, we plan to explore machine learning techniques to further fine-tune, enhance, and adapt Rupam to dynamic workloads and heterogeneous hardware.

Chapter 8

Conclusion and Future Work

As systems grow more complicated and heterogeneous, it is becoming more difficult to efficiently orchestrate all resources in a large system. The rapid increase of the need for diverse workloads makes it important to manage resources to best serve the dynamic demands of these workloads.

This dissertation reveals the problems of existing resource management for big data analytic platforms. It demonstrates inefficiencies in resource allocation from memory resources to processing resources, as well as insufficient task scheduling for workloads with intricate resource usage patterns. Deep empirical analyses [159, 204, 205] show a fundamental problem in the workload agnostic design of popular resource managers and task schedulers. This thesis also points out the challenges of designing such resource management and task scheduling systems that can satisfy both application needs and optimize overall resource utilization [203, 207, 208].

Additionally, this dissertation proposes a workload-aware resource management and task scheduling system. This system bridges the gap between big data platforms and underlying heterogeneous resources. We adopted machine learning approaches to capture the workload characteristics, and simple yet effective heuristics for resource allocation and task scheduling. We also applied adaptive algorithms with real-time feedbacks in our system for fast adjustment of suboptimal decisions. This thesis demonstrates our approach in memory resource management and processing resource management; finally, combining these efforts, the thesis proposes a general task scheduler that can host different kinds of applications and schedule tasks while considering resource consumption including memory, processors, networking, and storage.

8.1 Summary

Modern applications need to process larger amounts of data, with petabyte-scale data being generated every day. This trend is observed in both cloud and scientific applications. Driven by this need, we observed the increasing complexity and diversity of workloads, as well as the heterogeneity and scale of underlying systems. However, there is no efficient resource management and task scheduling system that is aware of the diversity on both sides. *This dissertation presents the problem of extant workload agnostic resource management systems in modern systems and proposes a workload-aware resource management and task scheduling system to provide the best scalability and performance within cost.*

The thesis first examines memory management for in-memory big data analytic platforms. We found that the static-partitioning-based approach in modern memory intensive platforms leads to memory resource contention that causes failure due to memory shortage. By dynamically allocating memory resources with workload information, we were able to avoid this error and improve the end-to-end performance with the same capacity of memory. We also propose an optimal data partitioning scheme that carefully selects the best parallelism and memory consumption at the task level that further increases the efficiency of memory usage, resulting in more performance improvement.

We then explored high-throughput accelerators such as GPUs in the HPC environment for scientific discoveries on big data. We enabled the adoption of big data analytic platforms in an HPC environment by providing transparent support to use GPUs to accelerate linear algebra operations that are frequently used in scientific discoveries. We then performed extensive evaluations of different workloads. Our experiments showed that, counter-intuitively, high-speed GPUs do not always deliver better performance, but this is heavily dependent on the workload. Based on our findings, we designed Arion to dynamically allocate and schedule tasks to both CPUs and GPUs based on task characteristics, input data, and resource availability.

Finally, this dissertation re-affirms that no one resource management scheme and task scheduling algorithm can fit all situations, especially when facing the diversity of modern big data applications and complicated workflows with different workloads. The thesis emphasizes the need for workload-aware resource management and task scheduling systems, and proposes Gerbil, which can co-host fundamentally different applications such as MPI and MapReduce. We also demonstrated that a workload-aware task scheduler that leverages workload information to schedule tasks in a heterogeneous setup can reduce failure-prone resource contention, improve workload balance, and improve workload performance.

8.2 Future Work

Distributed systems are quickly developing and changing. This dissertation's key idea of workload-aware resource management might be applied in future systems and workloads.

However, with more specialized hardware resources and larger-scale software, we will face new challenges for resource management. Here, we briefly outline future work in realizing workload-aware resource management in future environments, and the algorithms and methods that might improve.

8.2.1 Scalable Resource Monitor

We see a trend of future distributed systems growing to be larger scale and more heterogeneous, along with costlier and energy-efficient specialized hardware devices and more affordable traditional devices. Improved network speeds also enable devices to connect through the internet, i.e., IoT (Internet of Things). Hence, we foresee a future distributed system with much larger-scale and heterogeneous devices. There is an extreme need for a scalable resource manager.

Scale-out: Existing resource managers either adopt a centralized design, which limits deployment in large-scale systems, or a decentralized design, which sacrifices global optimization for scalability. Applying a workload-aware resource management and task scheduling system in scale requires a hybrid solution that can achieve both scalability and overall optimization for multi-tenant diverse workloads.

Scale-up: Machines in a fully distributed environment might be heterogeneous in terms of machine type (e.g., IoT), capacity, and I/O devices with which the machine is equipped. With more devices becoming available in commodity systems, we expect an increased diversity of machines/devices. Adapting to this trend will require tools that can automatically detect different hardware settings in a single machine/device and can profile such settings with the right methods to later help make decisions for different optimization goals.

8.2.2 Accurate Workload Profiler

The accurate characterization of workloads is key to workload-aware task scheduling. We have adopted methods such as offline profiling [159] that require extra overhead for a test run of an application and heuristic-based online metric monitoring and feedback [208] that introduce overhead such as online metric collecting. Moreover, online-based methods cannot capture the whole picture of an application, often lead to locally optimal solutions instead of global optimum, and introduce overhead for decision correction with feedback. As workloads become more complicated, we need innovative methods to quickly understand the characteristics and resource demands of a workload to better apply workload-aware task scheduling.

8.2.3 Performance Prediction

The performance of a task running in a certain machine/device is affected by several factors: resource availability, contention with other tasks, data locality, task characteristics, etc. It is not guaranteed to achieve the best performance even when identifying a machine that satisfies a task's resource requirements. A machine's inconsistent run-time status can also cause variance in the performance of a task. It would be interesting to create a performance model that can capture all relevant factors. Combined with modern deep learning methods, real-time performance prediction for a task with a confidence interval would greatly boost the efficacy of a workload-aware task scheduling system.

Bibliography

[1] Amazon Web Services (AWS) - Clound Computing Services. http://aws.amazon.com/.

[2] Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR.

[3] Crossbow: Genotyping from short reads using cloud computing. http://bowtie-bio.sourceforge.net/crossbow/index.shtml.

[4] Scaling memcached at Facebook. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919?comment_id=26807469&offset=0&total_comments=159&comment_tracking=%7B%22tn%22%3A%22R9%22%7D.

[5] MountableHDFS. https://wiki.apache.org/hadoop/MountableHDFS.

[6] Hamster: Hadoop And Mpi on the same cluSTER. https://issues.apache.org/jira/browse/MAPREDUCE-2911.

[7] Hydra process management framework. http://wiki.mcs.anl.gov/mpich2/index.php/Hydra.

[8] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr. [Online; accessed 12-Dec-2017].

[9] The Kepler Project. https://kepler-project.org/.

[10] AWS Case Study: Yelp. http://aws.amazon.com/solutions/case-studies/yelp/.

[11] Apache Mahout: Scalable machine learning and data mining. http://mahout.apache.org/.

[12] Wiki: Apache Hadoop. http://en.wikipedia.org/wiki/Apache_Hadoop.

[13] Memcached. http://memcached.org.

[14] MOAB Workload Manager. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.

[15] Message Passing Interface Forum. http://www.mpi-forum.org, .

[16] MPICH. www.mpich.org, .

[17] Hadoop MapReduce. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[18] MapReduce-MPI Library. http://mapreduce.sandia.gov/.

[19] MR+. http://www.open-mpi.org/video/?category=mrplus#Greenplum_RalphCastain.

[20] HDFS NFS Gateway. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/HdfsNfsGateway.html.

[21] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[22] Redis. http://redis.io.

[23] Standard Performance Evaluation Corporation. http://www.spec.org/mpi2007/.

[24] Hadoop Streaming. http://hadoop.apache.org/docs/r1.2.1/streaming.html#Hadoop+Streaming.

[25] System G. http://www.cs.vt.edu/facilities/systemg.

[26] Project Tungsten. https://issues.apache.org/jira/browse/SPARK-7075.

[27] Twemcache. https://github.com/twitter/twemcache.

[28] Consolidate storage and execution memory management. https://issues.apache.org/jira/browse/SPARK-10000.

[29] MPICH-yarn. https://github.com/alibaba/mpich2-yarn.

[30] Hadoop performance tuning. https://hadoop-toolkit.googlecode.com/files/Whitepaper-HadoopPerformanceTuning.pdf, 2012.

[31] Apache Hadoop. http://hadoop.apache.org, 2017. [Online; accessed 12-Dec-2017].

[32] PyTorch. https://pytorch.org, 2017. [Online; accessed 12-Oct-2018].

[33] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[34] E Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. *ACM SIGOPS*, 23(5), 1989.

[35] Sandeep R Agrawal, Christopher M Dee, and Alvin R Lebeck. Exploiting accelerators for efficient high dimensional similarity search. In *Proc. of the 21st ACM SIGPLAN*, page 3, 2016.

[36] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, 2012.

[37] M. Kamrul Ahsan and De-bi Tsao. Solving resource-constrained project scheduling problems with bi-criteria heuristic search techniques. *Journal of Systems Science and Systems Engineering*, 12(2):190–203, Jun 2003. ISSN 1861-9576. doi: 10.1007/s11518-006-0129-3. URL https://doi.org/10.1007/s11518-006-0129-3.

[38] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016. URL http://arxiv.org/abs/1605.02688.

[39] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: a

portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proc. 1st ACM SIGMOD SWEET Workshop*, 2012.

[40] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. Plapack: Parallel linear algebra package design overview. In *Proc. of SC*, pages 1–16, 1997. ISBN 0-89791-985-8. doi: 10.1145/509593.509622.

[41] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[42] Ali Anwar. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. PhD thesis, Virginia Tech, 2018.

[43] Ali Anwar, KR Krish, and Ali R Butt. On the use of microservers in supporting hadoop applications. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 66–74. IEEE, 2014.

[44] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 7–12. ACM, 2015.

[45] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[46] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 285–292. IEEE, 2015.

[47] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Scalable metering for an affordable it cloud service management. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 207–212. IEEE, 2015.

[48] Ali Anwar, Salman A Baset, Andrzej P Kochut, Hui Lei, Anca Sailer, and Alla Segal. Scalable metering for cloud service management based on cost-awareness, March 31 2016. US Patent App. 14/871,443.

[49] Ali Anwar, Yue Cheng, and Ali R Butt. Towards managing variability in the cloud. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1081–1084. IEEE, 2016.

[50] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 177–188. ACM, 2016.

[51] Ali Anwar, Yue Cheng, Hai Huang, and Ali Raza Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *HotStorage*, 2016.

[52] Ali Anwar, Andrzej Kochut, Anca Sailer, Charles O Schulz, and Alla Segal. Dynamic metering adjustment for service management of computing platform, March 31 2016. US Patent App. 14/926,384.

[53] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. bespo kv: application tailored scale-out key-value stores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 2. IEEE Press, 2018.

[54] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies*, page 265, 2018.

[55] Apache Spark. Support off-loading computations to a gpu. https://issues.apache.org/jira/browse/SPARK-3785, 2014.

[56] Apache Spark. Explore gpu-accelerated linear algebra libraries. https://issues.apache.org/jira/browse/SPARK-5705, 2015.

[57] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proc. of 2015 ACM SIGMOD*, 2015.

[58] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, Mark A. Taylor, T.S. Woodall, and M.W. Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *Proc. IPDPS*, 2004. doi: 10.1109/IPDPS.2004.1302920.

[59] Shuju Bai. *A Hybrid Framework of Iterative MapReduce and MPI for Molecular Dynamics Applications*. PhD thesis, Southern University, 2013.

[60] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, 2011.

[61] Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Antonio J Pena, Ken Raffenetti, Rajeev Thakur, and Junchao Zhang. Mpich user's guide. 2014.

[62] Rajanikanth Batchu, Yoginder S Dandass, Anthony Skjellum, and Murali Beddhu. Mpi/ft: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[63] Alex Belianinov, Rama Vasudevan, Evgheni Strelcov, Chad Steed, Sang Mo Yang, Alexander Tselev, Stephen Jesse, Michael Biegalski, Galen Shipman, Christopher Symons, et al. Big data and deep data in scanning and electron microscopies: deriving functionality from multidimensional data sets. *Advanced Structural and Chemical Imaging*, 1(1):1–25, 2015.

[64] L Susan Blackford, Jaeyoung Choi, A Cleary, James Demmel, I Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, et al. Scalapack: a portable linear algebra library for distributed memory computers-design issues and performance. In *Proc. of ACM/IEEE SC*, 1996.

[65] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[66] Norman Bobroff, Peter Westerink, and Liana Fong. Active control of memory for java virtual machines and applications. In *Proc. of 11th ICAC 14*, Philadelphia, PA, June 2014. URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/bobroff.

[67] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *VLDB Endow.*, 3(1-2):285–296, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920881. URL http://dx.doi.org/10.14778/1920841.1920881.

[68] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *HPCA*, 25(4):496–509, 2011.

[69] Ralph M. Butler, W. D. Gropp, and Ewing L. Lusk. A scalable process-management environment for parallel programs. In *7th EuroPVM/MPI Users Group Meeting*, 2000. URL http://www.springerlink.com/content/myr2r8at0183gja6/.

[70] Kirk W Cameron, Ali Anwar, Yue Cheng, Li Xu, Bo Li, Uday Ananth, Thomas Lux, Yili Hong, Layne T Watson, and Ali R Butt. Moana: Modeling and analyzing i/o variability in parallel system experimental design. 2018.

[71] John Canny and Huasha Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proc. of the 19th ACM SIGKDD*, pages 95–103, 2013.

[72] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. ACM EuroSys*, 2015.

[73] Weijia Chen, Yuedong Xu, and Xiaofeng Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*, 2017.

[74] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An alibaba case study. *arXiv preprint arXiv:1808.02919*, 2018.

[75] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[76] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 2007.

[77] Garth Gibson Chuck Cranor, Milo Polte. Hpc computation on hadoop storage with plfs. Technical Report CMU-PDL-12-115, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, November 2012. URL http://www.pdl.cmu.edu/PDL-FTP/HECStorage/CMU-PDL-12-115.pdf.

[78] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *Proc. ACM Data Management on New Hardware*, 2008.

[79] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proc. of the 30th ICML*, pages 1337–1345, 2013.

[80] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

[81] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the 6th Conference on Symposium on OSDI - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[82] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14', pages 127–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL http://doi.acm.org/10.1145/2541940.2541941.

[83] N. Desai. Cobalt: An open source platform for hpc system software research. *Edinburgh BG/L System Software Workshop*, 2005.

[84] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. A set of level 3 basic linear algebra subprograms for sparse matrices. *ACM Trans. Math. Softw*, 23: 379–401, 1995.

[85] Tarek Elgamal, Maysam Yabandeh, Ashraf Aboulnaga, Waleed Mustafa, and Mohamed Hefeeda. spca: Scalable principal component analysis for big data on distributed platforms. In *Proc. of ACM SIGMOD ICMD*, pages 79–91, 2015.

[86] Zacharia Fadika, Elif Dede, Jessica Hartog, and Madhusudhan Govindaraju. Marla: Mapreduce for heterogeneous clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 49–56. IEEE, 2012.

[87] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proc. of the 25th SOSP*, pages 394–409. ACM, 2015.

[88] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168847. URL http://doi.acm.org/10.1145/2168836.2168847.

[89] Rohan Gandhi, Di Xie, and Y. Charlie Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. USENIX ATC*, 2013.

[90] Rohan Gandhi, Di Xie, and Y Charlie Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *USENIX Annual Technical Conference*, pages 61–66, 2013.

[91] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proc. of the 19th ACM SOSP*, 2003. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

[92] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[93] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378. ACM, 2013.

[94] Alex Gittens, Jey Kottalam, Jiyan Yang, Michael F Ringenburg, Jatin Chhugani, Evan Racah, Mohitdeep Singh, Yushu Yao, Curt Fischer, Oliver Ruebel, Benjamin Bowen, Norman Lewis, Michael W Mahoney, Venkat Krishnamurthy, and Prabhat. A multiplatform evaluation of the randomized cx low-rank matrix factorization in spark. In *Proc. of the 5th ParLearning*, 2016.

[95] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. USENIX OSDI*, 2014.

[96] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[97] Jack Dongarra Graham E. Fagg. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proc. 7th European Users' Group Meeting on Recent Advances in PVM and MPI*, 2000.

[98] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14', page 455, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626334. URL http://doi.acm.org/10.1145/2619239.2626334.

[99] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *Int. J. High Perform. Comput. Appl.*, 18(3):363–372, August 2004. ISSN 1094-3420. doi: 10.1177/1094342004046045. URL http://dx.doi.org/10.1177/1094342004046045.

[100] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proc. of OSDI*, 2010.

[101] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28 (1):100–108, 1979.

[102] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB Endowment*, 2011.

[103] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of NSDI*, 2011.

[104] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi-and manycore chips. In *Proc. of WPMVP*, pages 57–64. ACM, 2014.

[105] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[106] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *Proc. USENIX ATC*, 2015.

[107] Costin Iancu, Steven Hofmeyr, Filip Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *Proc. IPDPS*, 2010.

[108] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. ACM Eurosys*, 2007.

[109] M. Jette and M. Grondona. Slurm: Simple linux utility for resource management. In *Proc. of ClusterWorld Conference and Expo*, 2003.

[110] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proc. of the 22nd ACM MM*, pages 675–678, 2014.

[111] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[112] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[113] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. Raising the bar for using gpus in software packet processing. In *Proc. of the 12th NSDI 15*, pages 409–423, 2015.

[114] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. of ICDM'09*, pages 229–238.

[115] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. 9th IEEE ICDM*, 2009. ISBN 978-0-7695-3895-2. doi: 10.1109/ICDM.2009.14. URL http://dx.doi.org/10.1109/ICDM.2009.14.

[116] David Kanter. Intel's haswell cpu microarchitecture. *Real World Technologies, November*, 2012.

[117] Qifa Ke, Vijayan Prabhakaran, Yinglian Xie, Yuan Yu, Jingyue Wu, and Junfeng Yang. Optimizing data partitioning for data-parallel computing. In *Proc. 13th USENIX HotOS*, 2011. URL http://dl.acm.org/citation.cfm?id=1991596.1991614.

[118] Alexey Kopytov. *SysBench*. http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[119] KR Krish, Ali Anwar, and Ali R Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2014.

[120] KR Krish, Ali Anwar, and Ali R Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 255–264. IEEE, 2014.

[121] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *Proc. ACM ASPLOS*, 2008.

[122] Victor Kunin, Alex Copeland, Alla Lapidus, Konstantinos Mavromatis, and Philip Hugenholtz. A bioinformatician's guide to metagenomics. *Microbiology and Molecular Biology Reviews*, 72(4):557–578, 2008.

[123] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning gemm kernels for the fermi gpu. *TPDS*, 23(11):2045–2057, 2012.

[124] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proc. ACM SIGMOD*, 2012.

[125] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. ACM ICAC*, 2012.

[126] Gunho Lee. *Resource allocation and scheduling in heterogeneous cloud environments*. University of California, Berkeley, 2012.

[127] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM SoCC*, 2014.

[128] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proc. of the 23rd HPDC*, 2014.

[129] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. of the 12th ACM ICCF*, 2015.

[130] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15', pages 53:1–53:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3358-0. doi: 10.1145/2742854.2747283. URL http://doi.acm.org/10.1145/2742854.2747283.

[131] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *Proc. of IEEE NAS*, pages 347–348, 2015.

[132] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based autotuning of mapreduce. In *Proc. Springer Euro-Par*, 2013.

[133] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *Proc. VLDB*, 2012.

[134] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M Shipman, and Chita R Das. D-factor: a quantitative model of application slow-down in multi-resource shared systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):271–282, 2012.

[135] J. Liu, Y. Liang, and N. Ansari. Spark-based large-scale matrix inversion for big data processing. *IEEE Access*, 4:2166, 2016. ISSN 2169-3536. doi: 10.1109/ACCESS.2016.2546544.

[136] Noel Lopes and Bernardete Ribeiro. Gpumlib: An efficient open-source gpu machine learning library. *IJCISIM*, 3:355–362, 2011.

[137] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha, and Zhiwei Xu. Datampi: Extending mpi to hadoop-like big data computing. In *Proc. IEEE 28th IPDPS*, 2014. doi: 10.1109/IPDPS.2014.90.

[138] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of ACM SIGMOD ICMD*, pages 135–146, 2010.

[139] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint*, 2015.

[140] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[141] Henry M Monti, Ali Raza Butt, and Sudharshan S Vazhkudai. Catch: A cloud-based adaptive data transfer service for hpc. In *Proc. IPDPS*, 2011.

[142] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[143] Arun Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Jeffrey Markham, and Joseph Niemiec. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[144] Naohito Nakasato. A fast gemm implementation on the cypress gpu. *ACM SIGMETRICS*, 38(4):50–55, 2011.

[145] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *HPCA*, 24(4):511, 2010.

[146] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proc. ACM SIGMOD*, 2011.

[147] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proc. of USENIX ATC 15*, July 2015.

[148] Netlib. Blas (basic linear algebra subprograms), 2017. http://www.netlib.org/blas/.

[149] netlib java. High performance linear algebra, 2013. https://github.com/fommil/netlib-java.

[150] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, et al. Rethinking data-intensive science using scalable analytics systems. In *Proc. of ACM SIGMOD ICMD*, pages 631–646, 2015.

[151] NVIDIA. Nvidia cuda blas library, 2007. http://docs.nvidia.com/cuda/cublas/#axzz4See8FUGO.

[152] NVIDIA. Nvidia blas library, 2007. http://docs.nvidia.com/cuda/nvblas/#abstract.

[153] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proc. of 93 ACM SIGMOD*.

[154] ORNL. Rhea - oak ridge leadership computing facility, 2017. https://www.olcf.ornl.gov/computing-resources/rhea/.

[155] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[156] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. SOSP*, 2017.

[157] Indranil Palit and Chandan K. Reddy. Scalable and parallel boosting with mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1904–1916, Oct 2012. ISSN 1041-4347. doi: 10.1109/TKDE.2011.208.

[158] Arnab Kumar Paul, Sourav Kanti Addya, Bibhudatta Sahoo, and Ashok Kumar Turuk. Application of greedy algorithms to virtual machine distribution across data centers. In *Annual IEEE India Conference (INDICON)*, 2014.

[159] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *CLUSTER, 2016 IEEE Conference on*. IEEE, 2016.

[160] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. ACM SIGMOD*, 2012.

[161] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. *Resource-Aware Adaptive Scheduling for MapReduce Clusters*, pages 187–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-25821-3. doi: 10.1007/978-3-642-25821-3_10. URL https://doi.org/10.1007/978-3-642-25821-3_10.

[162] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM TOMS*, 39(2):13, 2013.

[163] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Choi, Seung-Hee Bae, Hui Li, Bingjing Zhang, Tak-Lon Wu, Yang Ruan, Saliya Ekanayake, Adam Hughes, and Geoffrey Fox. Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics*, 11(Suppl 12):S3, 2010. ISSN 1471-2105. doi: 10.1186/1471-2105-11-S12-S3. URL http://www.biomedcentral.com/1471-2105/11/S12/S3.

[164] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proc. ACM International Conference on Extending Database Technology*, 2013.

[165] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *arXiv preprint arXiv:1602.08124*, 2016.

[166] Haakon Ringberg, Augustin Soule, Jennifer Rexford, and Christophe Diot. Sensitivity of pca for traffic anomaly detection. In *ACM SIGMETRICS*, 2007.

[167] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. 1st USENIX FAST*, 2002. URL http://dl.acm.org/citation.cfm?id=1083323.1083349.

[168] C. Selvakumar, G. J. Rathanam, and M. R. Sumalatha. Pdds - improving cloud data storage security using data partitioning technique. In *Proc. IEEE International Advance Computing Conference*, 2013.

[169] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proc. of the 2nd IEEE CloudCom*, pages 721–726, 2010.

[170] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proc. ACM SIGMOD*, 2013.

[171] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[172] Bikash Sharma, Timothy Wood, and Chita R Das. Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[173] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *VLDB Endowment*, 2014.

[174] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, August 2012.

[175] David B Shmoys, Joel Wein, and David P Williamson. Scheduling parallel machines on-line. *SIAM journal on computing*, 24(6):1313–1331, 1995.

[176] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[177] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[178] Garrick Staples. Torque resource manager. In *Proc. ACM/IEEE SC*, 2006. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188464. URL http://doi.acm.org/10.1145/1188455.1188464.

[179] Zilong Tan and Shivnath Babu. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. *Proceedings of the VLDB Endowment*, 9(10):720–731, 2016.

[180] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *The 37th IEEE International Conference on Distributed Computing Systems*, 2017.

[181] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10', pages 1013–1020, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807278. URL http://doi.acm.org/10.1145/1807167.1807278.

[182] C. Tian, H. Zhou, Y. He, and L. Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *2009 Eighth International Conference on Grid and Cooperative Computing*, pages 218–224, Aug 2009. doi: 10.1109/GCC.2009.19.

[183] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proc. ACM International Conference on Web Search and Data Mining*, 2014.

[184] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve. Automated data partitioning for highly scalable and strongly consistent transactions. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):106–118, 2016. ISSN 1045-9219.

[185] Alexander Ulanov. Nvblas:gpu usage with nvblas, 2013. https://github.com/fommil/netlib-java/wiki/NVBLAS.

[186] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[187] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. 4th SOCC*, 2013.

[188] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11', pages 235–244, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998637. URL http://doi.acm.org/10.1145/1998582.1998637.

[189] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Profiling and evaluating hardware choices for mapreduce environments: An application-aware approach. *Performance Evaluation*, 79:328 – 344, 2014. ISSN 0166-5316. doi: https://doi.org/10.

1016/j.peva.2014.07.020. URL http://www.sciencedirect.com/science/article/pii/S0166531614000832. Special Issue: Performance 2014.

[190] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. In *Proc. ACM VLDB*, 2012.

[191] Michael E Wall, Andreas Rechtsteiner, and Luis M Rocha. Singular value decomposition and principal component analysis. In *A practical approach to microarray data analysis*, pages 91–109. Springer, 2003.

[192] Bo Wang, Jinlei Jiang, and Guangwen Yang. Actcap: Accelerating mapreduce on heterogeneous clusters with capability-aware data placement. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 1328–1336. IEEE, 2015.

[193] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. A framework for distributed data-parallel execution in the kepler scientific workflow system. *Computer Science*, 9(0):1620 – 1629, 2012. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2012.04.178. URL http://www.sciencedirect.com/science/article/pii/S1877050912002992.

[194] Lin Wang. Directed acyclic graph. In *Encyclopedia of Systems Biology*. Springer, 2013.

[195] Zhigang Wang, Yubin Bao, Yu Gu, Fangling Leng, Ge Yu, Chao Deng, and Leitao Guo. A bsp-based parallel iterative processing system with multiple partition strategies for big graphs. In *Proc. of IEEE Big Data*, pages 173–180, 2013.

[196] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[197] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proc. ACM ISCA*, 2013.

[198] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proc. ACM SoCC*, 2011.

[199] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. Scalable matrix inversion using mapreduce. In *Proc. 23rd HPDC*. ACM, 2014.

[200] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. Scalable matrix inversion using mapreduce. In *Proc. of the 23rd ACM HPDC*, pages 177–190, 2014.

[201] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proc. 2013 ACM SIGMOD*.

[202] Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint*, 2014.

[203] L. Xu, M. Li, and A. R. Butt. Gerbil: Mpi+yarn. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)(CCGRID)*, volume 00, pages 627–636, May 2015. doi: 10.1109/CCGrid.2015.137. URL doi.ieeecomputersociety.org/10.1109/CCGrid.2015.137.

[204] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, volume 00, pages 383–392, May 2016. doi: 10.1109/IPDPS.2016.105. URL doi.ieeecomputersociety.org/10.1109/IPDPS.2016.105.

[205] L. Xu, S. Lim, A. R. Butt, S. R. Sukumar, and R. Kannan. Fatman vs. littleboy: Scaling up linear algebraic operations in scale-out data platforms. In *2016 1st Joint International Workshop on Parallel Data Storage and Data-Intensive Scalable Computing Systems (PDSW-DISCS)*, volume 00, pages 25–30, Nov. 2017. doi: 10.1109/PDSW-DISCS.2016.009. URL doi.ieeecomputersociety.org/10.1109/PDSW-DISCS.2016.009.

[206] Luna Xu, Seung-Hwan Lim, Ali R Butt, Sreenivas R Sukumar, and Ramakrishnan Kannan. Fatman vs. littleboy: scaling up linear algebraic operations in scale-out data platforms. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 25–30. IEEE Press, 2016.

[207] Luna Xu, Seung-Hwan Lim, Ali R Butt, and Ramakrishnan Kannan. Scaling up data-parallel analytics platforms: Linear algebraic operation cases. In *Proceedings of the 2017 IEEE International Conference on Big Data*, 2017.

[208] Luna Xu, Ali R. Butt, Seung-Hwan Lim, and R.R. Kannan. A heterogeneity-aware task scheduler for spark. In *Cluster Computing (CLUSTER), 2018 IEEE International Conference on*. IEEE, 2018.

[209] H. Yang, X. Liu, S. Chen, Z. Lei, H. Du, and C. Zhu. Improving spark performance with mpte in heterogeneous environments. In *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, pages 28–33, July 2016. doi: 10.1109/ICALIP.2016.7846627.

[210] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, 2013.

[211] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, 2013. ISSN 0163-5999.

[212] Ting Yang, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proc. of the 7th OSDI*, 2006.

[213] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proc. 18th ACM CIKM*, 2009. ISBN 978-1-60558-512-3. doi: 10.1145/1645953.1646301. URL http://doi.acm.org/10.1145/1645953.1646301.

[214] Reza Bosagh Zadeh and Ashish Goel. Dimension independent similarity computation. *The Journal of Machine Learning Research*, 14(1):1605–1626, 2013.

[215] Reza Bosagh Zadeh, Xiangrui Meng, Aaron Staple, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Alexander Ulanov, and Matei Zaharia. Matrix computations and optimization in apache spark. In *Proc. of the 22nd ACM SIGKDD*. ACM, 2016.

[216] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[217] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX HotCloud*, 2010. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[218] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[219] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*, 2012.

[220] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the 24th ACM SOSP*, 2013.

[221] X. Zhang, Z. Chen, L. Zhao, A. P. Boedihardjo, and C. T. Lu. Traces: Generating twitter stories via shared subspace and temporal smoothness. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1688–1693, Dec 2017. doi: 10.1109/BigData.2017.8258107.

[222] X. Zhang, L. Zhao, Z. Chen, A. P. Boedihardjo, J. Dai, and C. T. Lu. Trendi: Tracking stories in news and microblogs via emerging, evolving and fading topics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1590–1599, Dec 2017. doi: 10.1109/BigData.2017.8258093.

[223] Xuchao Zhang, Liang Zhao, Arnold P. Boedihardjo, Chang-Tien Lu, and Naren Ramakrishnan. Spatiotemporal event forecasting from incomplete hyper-local price data. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 507–516, 2017. doi: 10.1145/3132847.3132996. URL http://doi.acm.org/10.1145/3132847.3132996.

[224] Werner Saar Zhang Xianyi, Wang Qian. Openblas : An optimized blas library, 2011. http://www.openblas.net.

[225] Nannan Zhao, Ali Anware, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali Butt. Chameleon: An adaptive wear balancer for flash clusters. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1163–1172. IEEE, 2018.

[226] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on multicore and multi-gpu platforms using functional performance models. *IEEE Transactions on Computers*, 64(9):2506–2518, 2015.