INSIGHTFUL PERFORMANCE ANALYSIS OF MANY-TASK RUNTIMES THROUGH TOOL-RUNTIME INTEGRATION

by

NICHOLAS A. CHAIMOV

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2017

DISSERTATION APPROVAL PAGE

Student: Nicholas A. Chaimov

Title: Insightful Performance Analysis of Many-Task Runtimes through Tool-Runtime Integration

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Allen D. Malony	Chair
Boyana R. Norris	Core Member
Hank R. Childs	Core Member
Gregory Bothun	Institutional Representative

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2017

© 2017 Nicholas A. Chaimov

DISSERTATION ABSTRACT

Nicholas A. Chaimov

Doctor of Philosophy

Department of Computer and Information Science

June 2017

Title: Insightful Performance Analysis of Many-Task Runtimes through Tool-Runtime Integration

Future supercomputers will require application developers to expose much more parallelism than current applications expose. In order to assist application developers in structuring their applications such that this is possible, new programming models and libraries are emerging, the *many-task runtimes*, to allow for the expression of orders of magnitude more parallelism than currently existing models.

This dissertation describes the challenges that these emerging many-task runtimes will place on performance analysis, and proposes deep integration between runtimes and performance tools as a means of producing correct, insightful, and actionable performance results. I show how tool-runtime integration can be used to aid programmer understanding of performance characteristics and to provide online performance feedback to the runtime for Unified Parallel C (UPC), High Performance ParalleX (HPX), Apache Spark, the Open Community Runtime, and the OpenMP runtime.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Nicholas A. Chaimov

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Portland State University, Portland, OR
Reed College, Portland, OR

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2017, University of Oregon
Master of Science, Computer and Information Science, 2012, University of Oregon
Bachelor of Science, Computer and Information Science, 2010, University of Oregon
Bachelor of Science, Biology, 2007, University of Oregon

AREAS OF SPECIAL INTEREST:

High-Performance Computing
Scientific Computing
Performance Monitoring

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, Computer and Information Science, University of Oregon, 2010–2017

Software Engineering Intern, Intel Federal LLC, Summer 2016

Research Assistant, Lawrence Berkeley National Lab, Summer 2015

Research Assistant, Lawrence Berkeley National Lab, Summer 2014

GRANTS, AWARDS AND HONORS:

Gurdeep Pall Graduate Student Fellowship, University of Oregon, 2016

Student Travel Grant, High Performance Distributed Computing (HPDC), 2016

Member of 1st Place Graduate Team, Eugene Luks Programming Competition, University of Oregon, 2015

Member of 1st Place Graduate Team, Eugene Luks Programming Competition, University of Oregon, 2012

Member, Upsilon Pi Epsilon International Honor Society for the Computing and Information Disciplines, 2010–Present

PUBLICATIONS:

Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. “Scaling Spark on HPC Systems.” *International Symposium on High-Performance Parallel and Distributed Computing* (HPDC). 2016.

Nicholas Chaimov, Allen Malony, Costin Iancu, and Khaled Ibrahim. “Scaling Spark on Lustre.” *Workshop On Performance and Scalability of Storage Systems*. 2016.

Nicholas Chaimov, Allen Malony, Khaled Ibrahim, Costin Iancu, Shane Canon, and Jay Srinivasan. “Performance Evaluation of Apache Spark on Cray XC Systems.” *Cray Users Group*. 2016.

Md Abdullah Shahneous Bari, **Nicholas Chaimov**, Abid M Malik, Kevin A Huck, Barbara Chapman, Allen D. Malony, Osman Sarood. “ARCS: Adaptive Runtime Configuration Selection for Power-Constrained OpenMP Applications.” *IEEE International Conference on Cluster Computing* (CLUSTER). 2016.

Nicholas Chaimov, Khaled Ibrahim, Sam Williams and Costin Iancu. "Reaching Bandwidth Saturation Using Transparent Injection Parallelization." *International Journal of High Performance Computing Applications*. 2016.

Kevin Huck, Allan Porterfield, **Nicholas Chaimov**, Hartmut Kaiser, Allen D. Malony, Thomas Sterling, and Rob Fowler. "An Autonomic Performance Environment for Exascale." *Supercomputing Frontiers and Innovation* 2, no. 3 (2015): 49-66.

Robert Lim, Allen Malony, Boyana Norris, and **Nicholas Chaimov**. "Identifying Optimization Opportunities Within Kernel Execution in GPU Codes." *Euro-Par 2015: Parallel Processing Workshops*, pp. 185-196. 2015.

Nicholas Chaimov, Khaled Ibrahim, Sam Williams and Costin Iancu. "Exploiting Communication Concurrency on High Performance Computing Systems." *International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM). 2015.

Nicholas Chaimov, Boyana Norris, and Allen D. Malony. "Toward Multi-target Autotuning for Accelerators." *International Conference on Parallel and Distributed Systems* (ICPADS). 2014.

Nicholas Chaimov, Boyana Norris, and Allen D. Malony. "Integration and Synthesis for Automated Performance Tuning: the SYNAPT Project." *International Workshop on Automatic Performance Tuning* (iWAPT). 2014.

Nicholas Chaimov, Scott Biersdorff, and Allen D. Malony. "Tools for machine-learning-based empirical autotuning and specialization." *International Journal of High Performance Computing Applications* 27.4 (2013): 403-411.

ACKNOWLEDGEMENTS

I thank my advisor, Prof. Allen Malony, for his help with research, with identifying research areas, with securing interesting and useful internships, and, most of all, for convincing me to pursue a PhD. I also thank Prof. Hank Childs and Prof. Boyana Norris for their help with my research, as well as Sameer Shende and Kevin Huck of the Performance Research Lab. I thank my collaborators at Lawrence Berkeley National Lab: Costin Iancu, Khaled Ibrahim, and Sam Williams; and my collaborators at Intel: Bala Seshasayee, Romain Cledat, Bryan Pawlowski, and Nick Pepperling.

Parts of this document were supported by the DOE Office of Advanced Scientific Computing Research (contract number DE-AC02-05CH11231). Parts of this document are based upon work supported by the Department of Energy Office of Science under Award Number DE-SC0008717. Support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE-FG02-11ER26050 and DE-SC0006925. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No.

DE-AC02-05CH11231. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was partially supported by the Intel Parallel Computing Center at Lawrence Berkeley National Laboratory - Lustre.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. Thesis Statement 2

1.2. Dissertation Outline 5

1.3. Coauthored Material 11

II. BACKGROUND AND RELATED WORK 12

2.1. Current Programming Models 13

2.2. Capturing Performance Data 23

2.3. Autotuning 26

2.4. Performance Modeling 37

2.5. Performance Diagnosis 43

2.6. Exascale Computing and Future Programming Models 53

2.7. Conclusion 69

III. ONLINE COMMUNICATIONS ADAPTATION IN UPC 72

3.1. Introduction 72

3.2. Maximizing Message Concurrency 73

3.3. Communication and Concurrency 76

3.4. Runtime Design 79

3.5. Network Performance and Saturation 85

3.6. Parallelizing Injection in Applications 95

3.7. Discussion 105

3.8. Other Related Work 107

3.9. Conclusion 108

3.10. Bridge 109

IV. PERFORMANCE MEASUREMENT AND ONLINE ADAPTATION IN HPX 112

4.1. Introduction 112

4.2. APEX Design 113

4.3. Experimental Results 119

4.4. Tuning with a Global View 123

4.5. Conclusion 124

4.6. Bridge 125

V. STORAGE OPTIMIZATION AND VARIABILITY IN SPARK 127

5.1. Introduction 127

5.2. Motivation 128

5.3. Spark Architecture 132

5.4. Experimental Setup 139

5.5. Single Node Performance 142

5.6. Scaling Concerns 143

5.7. Improving Shuffle Scalability With Better Block Management 159

5.8. Spark-Perf Benchmark on Lustre 162

5.9. Localizing Metadata Operations with Shifter 165

5.10. Input Disk I/O versus Shuffle Disk I/O 166

5.11. Localizing JVM and Spark Runtime Metadata Accesses 168

5.12. Xeon Phi and the Effect of Straggler Tasks 169

5.13. Network Latency 170

5.14. Discussion 171

5.15. Related Work 174

5.16. Conclusion 175

Chapter Page

5.17. Bridge 176

VI. STRAGGLER ANALYSIS IN OCR 194

6.1. Introduction 194

6.2. APEX-OCR Integration 195

6.3. Task Eligibility 200

6.4. Tracing OCR Applications 200

6.5. Blame Analysis 203

6.6. Load Balancing 205

6.7. Conclusion 206

6.8. Bridge 208

VII. OPTIMIZING SCHEDULING IN OPENMP 209

7.1. Introduction 209

7.2. Motivation 210

7.3. Framework 214

7.4. Experiment Design 218

7.5. Related Work 236

7.6. Conclusions 238

7.7. Bridge 239

VIII. CONCLUSIONS 240

REFERENCES CITED 242

LIST OF FIGURES

Figure Page

1. Comparison of fork-join and task parallelism. 2
2. The fork-join model as used in OpenMP. 14
3. The Communicating Sequential Processes model as used in MPI. 16
4. Architecture of the NVIDIA Fermi GPU family. 19
5. Architecture of the AMD Radeon 7000 GPU family. 20
6. Architecture of the Intel Xeon Phi Knights Corner family. 22
7. Hypothetical architecture for a unified autotuning system. 27
8. Architecture of Orio. 31
9. Architecture of an autotuning system using ROSE. 33
10. Architecture of the Periscope Tuning Framework. 34
11. Architecture of Prophesy. 39
12. Architecture of the Brainy data structure selection system. 42
13. Examples of the Paradyn "why" and "when" hierarchies. 45
14. Architecture of the PERISCOPE system. 47
15. Symptom and cause hierarchies. 50
16. Methodology of DECAN. 52
17. Architecture of the Charm++ load balancer. 57
18. Decision procedure used by PICS. 58
19. DAG for an OCR Fibonacci code. 64
20. Computational model of HPX. 68
21. Runtime architecture of THOR. 80
22. Cray Aries network saturation. 85

Figure Page

23. Cray Gemini network saturation. 86

24. InfiniBand network saturation. 86

25. Speedup with parallel injection for an all-to-all microbenchmark. 94

26. The exchange boundary communication phase. 97

27. Fraction of time spent communicating in miniGMG. 97

28. Performance of UPC miniGMG with parallelization relative to without parallelization. 99

29. Performance of UPC miniGMG with selective parallelization relative to without parallelization. 99

30. Performance of UPC miniGMG with communication servers relative to without parallelization. 100

31. Performance of UPC+CUDA miniGMG with communication servers. 102

32. Aggregate bandwidth achieved with one-sided Put and Get operations. 110

33. Performance of the UPC-FT benchmark. 110

34. Overall trends comparing hybrid setup. 111

35. Design of the APEX introspection system. 113

36. Performance behavior of HPX 1D Stencil under different throttling policies. . 121

37. HPX miniGhost strong scaling. 122

38. Energy usage of HPX miniGhost under different throttling policies. 123

39. Local and global tuning sessions of HPX 1D stencil. 124

40. Decomposition of a Spark job into stages and tasks. 133

41. Data movement in Spark. 133

42. Architecture of the Lustre filesystem. 134

43. Node architecture of the Cori Burst Buffer. 135

44. Network topology of the Cori Burst Buffer. 135

45. Big Data Benchmark single node performance. 142

Figure Page

46. Distribution of file I/O on Lustre vs. ext4 local disk. 144
47. Time for map and reduce phases of GroupBy. 148
48. Average time for open, read, and write operations. 149
49. Performance improvements from amortizing file opens. 151
50. GroupBy and PageRank performance on Cori. 153
51. GroupBy weak scaling on Cori. 154
52. PageRank and Big Data Benchmark scaling on Cori. 155
53. GroupBy at large scale on Cori on Lustre. 158
54. GroupBy at large scale on Cori on Lustremount. 158
55. PageRank perfomrance on Edison. 160
56. I/O behavior of PageRank on Edison. 160
57. Number of partitions read during shuffe for PageRank. 177
58. GroupBy performance on Lustre or RAMdisk. 178
59. GroupBy performance on Lustre or local SSD. 178
60. Slowdown of Spark Core benchmarks on Comet. 179
61. Slowdown from open-per-read rather than single-open. 179
62. Slowdown of MLLib on Comet. 180
63. Architecture of Shifter. 181
64. GroupBy benchmark on Lustre, RAMdisk, or per-node loopback filesystem. . . 181
65. Slowdown of Spark Core on Cori with Lustre instead of loopback filesystem. . 182
66. Slowdown of MLLib on Cori with Lustre instead of loopback filesystem. . . . 183
67. Weak scaling for MLLib on loopback filesystems and local SSDs. 184
68. Weak Scaling results for Mini TeraSort. 185
69. Weak Scaling results for Big Data Benchmark. 185
70. Weak Scaling results for Power Iteration Clustering. 186

Figure Page

71. Weak Scaling results for Spearman Correlation. 186

72. Weak Scaling results for Pearson Correlation. 187

73. Weak Scaling results for GroupBy on Xeon Phi. 187

74. Weak Scaling results for GroupBy on Xeon Phi in Cache vs Flat MCDRAM mode. 188

75. Time spent in garbage collection for GroupBy on Xeon Phi. 188

76. Execution traces of GroupBy on Haswell and Xeon Phi. 189

77. Hypothetical execution time of GroupBy after removing stragglers. 189

78. Weak scaling for the first block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori. 190

79. Weak scaling for the second block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori. 191

80. Weak scaling for the third block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori. 192

81. Bandwidth by message size for UDP and native RDMA on Comet and Cori. 192

82. Injection CPU overhead by message size for UDP and native RDMA on Comet and Cori. 193

83. End-to-end latency by message size for UDP and native RDMA on Comet and Cori. 193

84. Design of the OCR-APEX interface. 196

85. Task creation graph of the HPCG miniapp. 198

86. Concurrency visualization of the Stencil2D benchmark. 199

87. Concurrency visualization of the HPCG miniapp. 199

88. Task eligibility visualization of the HPCG miniapp. 201

89. User interface of the APEX Trace Viewer. 202

90. Process for assigning blame for idle regions. 204

91. OCR MiniAMR with and without load balancing policy. 207

Figure Page

92. Execution time for compute_rhs using different OpenMP runtime configurations. 212

93. Feature comparison between the default and *ARCS-Offline* strategy. 226

94. Execution time and package energy comparison among the default and ARCS strategies in SP 227

95. Execution time and energy consumption comparison of ARCS strategies and the default strategy in data set C of SP. 228

96. Feature comparison between the default and *ARCS-Offline* strategy at TDP power level for compute_rhs region of BT. 230

97. Application level execution time and package energy comparison among the default and ARCS strategies in BT with data set B. 232

98. Application level execution time and package energy comparison among the default and ARCS strategies in LULESH. 234

99. OpenMP events data for top 5 time consuming regions from LULESH. 235

100. Feature comparison among default and ARCS strategies on CalcFBHourglassForceForElems_1 region. 236

LIST OF TABLES

Table Page

1. Sizes of messages sent by UPC+CUDA miniGMG. 102
2. Messages sent in MLLib *pic* benchmark. 167
3. Per-parallel-region parameter settings with best performance for NAS OpenMP benchmarks. 211
4. Set of ARCS search parameters for OpenMP parallel regions. 217
5. Optimal configuration chosen by ARCS-Offline strategy for SP regions. 224

CHAPTER I

INTRODUCTION

Current supercomputers are equipped with $O(10,000)$ nodes, each with within-node concurrency of $O(100)$, providing performance of tens to one hundred petaflops [216]. Reaching exascale performance will require increases in both within-node concurrency to $O(1,000)$ and in the number of nodes to $O(100,000)$, with the effect that in order to make use of the available resources, programs will need to expose multi-billion-way concurrency [6]. The requirement to make available such a large volume of concurrent work is driving the development of new programming models and runtimes for those models: the many-task runtime [56].

The central idea behind task parallelism is that work is divided into discrete chunks which carry dependency information. The runtime's primary responsibility is to schedule work whose dependencies have been satisfied, and to switch between tasks with as little overhead as possible. Traditional runtimes such as MPI and OpenMP rely on synchronization primitives such as locks and barriers to enforce correct ordering of operations, resulting in the potential for load imbalance to severely limit utilization of available computational resources. Task-based runtimes replace locks and barriers with runtime awareness of dependencies, reducing idleness by allowing cores to begin processing work as soon as it is available, rather than waiting at barriers until an entire phase of the application has completed. They also allow for adaptation to system variability by allowing work to migrate across nodes to address load imbalance caused by node variability; to do this, units of work are virtualized relative to hardware. Data is often also virtualized, so that data can be moved to work, or work can be moved to data, depending upon whichever is cheaper [199]. Figure 1 demonstrates the advantage of a task-based approach over a traditional fork-join approach.

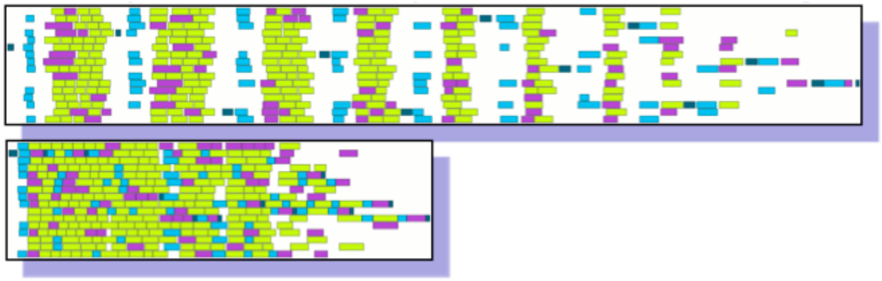


Figure 1. Comparison of fork-join and task parallelism. Execution trace of the same algorithm implemented using fork-join parallelism (top) and task-based parallelism (bottom). The bottom version executes in less time because worker threads can continue executing tasks as soon as the tasks' dependencies have been satisfied. From [244].

Application developers need performance-monitoring tools in order to understand the performance of their application so that they can determine what optimizations are needed to increase performance. Many such tools exist for programs written using traditional programming models. Changing from a fork-join or bulk synchronous programming model to a much more dynamic task-based model, in which different runs of the same application on the same data can result in different task schedules, and in which the assignment of both work and data to computational and storage resources are under the control of the runtime rather than the application, will require new tools. This document proposes new tools to aid in producing correct, insightful, and actionable performance measurements in emerging task-based runtimes.

1.1 Thesis Statement

The central premise of the dissertation is that **correct, insightful, and actionable performance monitoring requires integration between tools and runtimes.**

Applying a performance monitoring tool designed for traditional programming models to applications using a task-based runtime may yield results which are *not* *correct*: the tool may prevent the application from running to completion, such as by making assumptions about a maximum number of threads which are violated by the runtime, or by introducing overheads which are acceptable for the lower levels of concurrency exposed by traditional runtimes but which unacceptably distort timings when a large number of short-running tasks are timed.

They may produce results which are *not insightful*: by being unaware of the greater levels of abstraction found in task-based runtimes, a traditional performance monitoring tool will provide data at the level of the runtime and not of the application. For example, sampling the processes being executed and providing a report of the amount of time spent in each function is unlikely to provide insight, as it is not useful for an application developer to learn that during execution of the application, time was spent in various scheduling and network functions internal to the runtime. The tool must instead map runtime operations to their associated application-level task, providing the developer with performance data *at the level of the application*.

Finally, they may produce results which are *not actionable*: the tool must make clear, for a given performance problem, what changes could remedy the problem. Consider an application, measurement of which reveals regions of low concurrency, during which many workers do not have tasks assigned to them. Knowing this is not sufficient for the developer to know what to do about it. The application developer needs to know *why* no tasks are running: are there tasks whose dependencies have been satisfied, but which are not yet executing due to scheduler overhead? If so, a different scheduler policy may help. Are all existing tasks not yet eligible due to unsatisfied dependencies? If so, which tasks would have to complete in order for work to be available? Why, in turn, have *those* tasks not yet run? A tool for task-based runtimes should be able to provide answers to these questions.

The premise applies equally to tools for online adaptation: integration with the runtime is required.

As an example, consider the INNCABS benchmarks [213]. These are a port of the Barcelona OpenMP Task Suite benchmarks (BOTS) [65] to the C++11 asynchronous task facility, which can run either directly on the task implementation of the C++ compiler's runtime library, or through HPX. These applications are structured with a very small task granularity, and were designed with the purpose of testing the implementation of tasks in C++11 runtimes. Attempting to measure the performance of the INNCABS benchmarks with traditional performance monitoring tools like TAU [192] or HPCToolkit [2] reveals the limitations of those tools when applied to many-task runtimes. When instrumented with TAU, each of the 14 benchmarks fails to complete due to exhausting resources within TAU, which was not designed for applications creating millions of threads [89]. When sampled with HPCToolkit, 11 of the 14 benchmarks fail to complete due to exhaustion of tool resources. The three benchmarks that do complete take an average of 100.59× longer to complete. In contrast, the INNCABS benchmarks executed on the HPX using APEX for performance monitoring (as described in Chapter IV) successfully completes and produces performance profiles with an average overhead of 6%.

Furthermore, the performance data produced by a tool integrated with the runtime can be designed with the correct abstractions for the runtime. Traditional performance monitoring tools typically use function-level *callpath profiling*, where what is being measured is the *time spent in functions* captured in terms of the call stack. For traditional runtimes, this is an appropriate way to measure and present data, as the runtime is relatively minimal and is primarily *invoked by the application code*. In contrast, many-task runtimes incorporate a considerable amount of functionality directly into the runtime, and invert the relationship between runtime and application: application code is primarily *called by the runtime*. In consequence, a callpath profile of a many-task application identifies the set of runtime functions that ultimately invoked particular application functions, but does not capture the application level dependencies, with user tasks instead being seen as having been separately invoked by the runtime scheduler code. A tool integrated with the runtime can collect dependency data from the runtime, and generally be aware of metadata associated with tasks, thereby providing performance data at the same level of abstraction exposed by the runtime to the application developer.

1.2 Dissertation Outline

This dissertation consists of a series of performance studies based on tool-runtime integration for four different distributed runtimes: UPC [219], HPX [119], Spark [248], and OCR [152]. This may result in the question: why analysis of many different runtimes, as opposed to one? The primary reason for studying many different runtimes is that, while they are all based on the same central abstraction of tasks, they differ considerably along several axes related to *what* is abstracted by the runtime and *how explicit* the developer must be in specifying dependencies and resource allocations – or, equivalently, *how much freedom* the runtime has in distributing work and data. Additionally, I show that the same techniques can also be applied to more traditional runtimes, through a performance study with OpenMP.

UPC is a low-level runtime, providing minimal abstractions over work. The primary abstraction is over data, with the distributed memory of a system being presented as if it were a single global memory. HPX's primary abstraction is over work, with the central premise that *work moves to data*, rather than data moving to work, and with dependencies expressed *implicitly*. OCR abstracts both work and data, and expresses dependencies *explicitly*. Spark provides the highest level of abstraction of these runtimes, with abstractions of both work and data, where even the format of the data is unspecified by the developer and the entire memory hierarchy, from disk to cache, is managed by the runtime.

These differences occur because these runtimes are themselves research projects: we do not know which set of design decisions will maximize performance and programmer productivity, so many different designs are being attempted. It may be that none of the currently existing runtimes is what is eventually used on exascale systems. By designing performance monitoring and online adaptation tools that can work with many currently existing runtimes which have made different design choices, however, we ensure that the tool design is general enough to be adapted to whatever runtimes ultimately end up in wide use. Several tools have been developed for *specific* runtimes, such as Legion Prof for Legion [21] and Projections for Charm++ [121], but there has been little prior work on general, portable tools for task-based runtimes.

Background and Related Work. In **Chapter 2** of the dissertation, I discuss background material and related work in the areas necessary for understanding the work described in this document. I describe the programming models currently in wide use in high performance computing and the tools that exist for capturing performance data and diagnosing performance problems in those rumtimes. I then describe the existing task-based runtimes and emerging many-task runtimes, and techniques and tools for performance monitoring and online adaptation for them.

Online Communications Adaptation in UPC. In **Chapter 3** of the dissertation, I describe tool-runtime integration in Unified Parallel C [219] (UPC). UPC is a language which extends C99 [112] with support for *shared pointers* to a partitioned global address space and a variety of work-sharing constructs. Shared pointers allow a program to reference an address which may be either local or remote. If the memory reference is local, this is handled as an ordinary memory reference. If it is remote, the memory reference is transparently converted into network operations in the form of Remote Direct Memory Access (RDMA) operations. Remote communications are thus made implicit, allowing the runtime freedom to reorder, coalesce, split, or otherwise manipulate communications so long as the programmer-visible result remains unchanged.

I show how the UPC runtime can be instrumented to capture network flow data (which nodes are communicating what volume of data with which other nodes), how the network flow data can be mapped to the application contexts in which they occur (corresponding to phases of computations), and how these measurements can be used post-mortem to understand the performance of UPC applications on different network environments (TCP vs. InfiniBand vs. Cray Gemini vs. Cray Aries), and can be used online to dynamically adjust the level of concurrency in network injection (a runtime parameter) based upon the size and destinations of communication requests (which occur at the application level) in order to maximize throughput and reduce end-to-end application time.

Performance Measurement and Online Adaptation in HPX. In **Chapter 4** of the dissertation, I describe tool-runtime integration in High Performance ParalleX (HPX). HPX is a task-based runtime and C++ library based on the concept of *futures*. A future is an object representing the result of a computation which may or may not be available yet. Task invocation in HPX returns a future. When a task attempts to retrieve the result from a future whose result is not yet available (because the corresponding task has not yet completed), the current task suspends and another task is scheduled in its place; this is a form of *implicit* dependency specification. If the future represents the result of a task which executed on a different node, necessary network operations are automatically performed by the runtime.

I show how the HPX runtime can be instrumented to capture a variety of different application-level and runtime-level metrics. At the application level, I capture task metrics: time each class of tasks spends in staging, execution, and yielded states, and the causes of yields. At the runtime level, I capture scheduler data (lengths of queues, idle time, network flow data). I then show how these metrics can be used to determine optimal task granularity and to generate reports and visualizations that provide the application developer with insight into the performance of their applications. I then show how user-configurable policies can be used to automatically adjust task granularity during runtime. I then show how performance measurements throughout a distributed HPX application can be made available to the runtime and to policies, providing a *global view* of overall system performance.

A particularly important aspect of the HPX integration is that the monitoring tool itself uses HPX runtime features to carry out its measurement, processing, and communications. Processing of measurements occurs inside HPX tasks, and HPX's Active Global Address Space is used to access performance measurements across nodes.

Storage Optimization and Variability in Spark. In **Chapter 5** of the dissertation, I describe tool-runtime integration in Apache Spark [248]. Spark is a data analytics framework based on a generalization of the Map-Reduce model, found in systems such as Hadoop, to problems expressed as general data flow graphs. Operations are carried out on *resilient distributed datasets*, or RDDs, which store data across nodes and which carry sufficient information to recompute their contents. Programs are expressed in terms of RDDs derived from transformations (of which map is only one) applied to other RDDs and actions (of which reduce is only one). The runtime breaks transformations and actions down into tasks operating on partitions of RDDs. A notable feature of Spark is that the *storage* of RDDs is under control of the runtime: results can be discarded and recomputed as needed, or cached in memory or saved to disk. Unlike the other runtimes, Spark is designed not for special-purpose supercomputers but for commodity clusters and cloud environments.

I show how the Spark runtime can be instrumented to capture application-level and runtime-level metrics. I show that by measuring runtime overheads associated with internal runtime operations with phases of the application, I can identify the causes of increased overheads when running on supercomputer systems without local disks in certain applications (those with substantial *wide shuffling*). After identifying inefficient use of remote filesystem resources, I show how both runtime modifications and application-level modifications can improve performance, and how this is shown by generated reports and visualizations. The specific problem in this case is not high average filesystem latency but high *variability* in filesystem latency resulting in straggler tasks. I then perform a sensitivity analysis, showing how susceptible to variability each stage of a Spark application is, and demonstrate an online policy which identifies and mitigates bottleneck stages.

Straggler Analysis in OCR. In **Chapter 6** of the dissertation, I describe tool-runtime integration in the Open Community Runtime [152] (OCR). OCR is a task-based runtime which provides abstractions for both work and data. Unlike HPX, dependencies are specified *explicitly* at task creation time, and data is represented explicitly in the task graph, alongside tasks. Among all of the runtimes I evaluate in the dissertation, OCR thus has the greatest amount of task and datablock metadata with which to make placement and scheduling decisions. While HPX allows tasks to execute until they request data not yet available, OCR does not allow a task to execute at all until all input data is ready; thus, once a task begins executing, it always continues to completion without interruption.

I show how the OCR runtime can be instrumented to capture application-level and runtime-level metrics. Uniquely to OCR, I show that since the runtime is itself aware of all dependencies, I can incorporate dependency data into captured performance profiles and traces. With this dependency data, I develop a tool that can automatically diagnose the causes of idle regions, providing reports of the form: x milliseconds of idleness occurred because no schedulable work was available. The idle region ended at time y, when schedulable work became available; that work could not execute earlier because it depended on task t_1, which depended on task t_2, which was awaiting completion of the running task t_3. Task t_3 was eligible to run earlier than it did; thus, a different schedule would have reduced the duration of the idle region. I show how this analysis can be used to assign task priorities to minimize idleness.

I then show how a monitoring tool can distribute load information partially by piggybacking on top of existing communications, and partially by using a gossip protocol, to provide a low-overhead global view of load imbalance. I show how the performance of an inherently load-imbalanced Adaptive Mesh Refinement application can be improved through the use of online policy-based load balancing, in which the monitoring tool provides feedback to the runtime, which it uses in placement of data and migration of tasks.

Optimizing Scheduling in OpenMP. In **Chapter 7** of the dissertation, I describe tool-runtime integration with OpenMP, a non-distributed, traditional runtime. Although runtime integration is not *necessary* in traditional runtimes, it can nonetheless provide better insights than non-integrated monitoring tools and can enable online adaptation. I show how integrating with OpenMP runtimes through the

OpenMP Tool Interface allows performance measurements to be disaggregated across invocations of parallel regions, and how this can be used to increase the performance of OpenMP applications by providing feedback to the runtime on how loop iterations are scheduled.

Finally, in **Chapter 8**, I describe general conclusions from the tool-runtime integrations described in this document and propose additional use cases for such integrations.

1.3 Coauthored Material

This dissertation includes previously published co-authored material.

Chapter 3 includes co-authored material previously published in the Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2015) [38] and the International Journal of High Performance Computing Applications (IJHPCA) [40].

Chapter 4 includes co-authored material previously published in Supercomputing Frontiers [102].

Chapter 5 includes co-authored material previously published in the Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2016) [41], the 2016 Cray Users Group symposium [39], and the Workshop on Performance and Scalability of Storage Systems [42].

Chapter 7 includes co-authored material previously published in the IEEE International Conference on Cluster Computing (CLUSTER 2016) [191].

CHAPTER II

BACKGROUND AND RELATED WORK

Parallel programming is difficult. Switching from sequential to parallel programming introduces entire new classes of errors for the programmer to make, such as deadlock and race conditions, which are difficult to debug and complicate testing and correctness proofs [154]. Yet there are entire classes of programs with computational demands so great that sequential solutions are infeasible. We do parallel programming because we care about performance.

How do we know if we are getting good performance? We must observe the execution of our programs to determine if they are making good use of the resources available to them. Once we have made observations, how can we use those observations to improve performance? We can use autotuning to identify variants and parameters that give better performance than others, but this process itself is slow, so we can attempt to synthesize performance data into empirical models to guide the process. Alternately, we may ask not simply for better performance, but for an *explanation* of performance: automated performance diagnosis. These techniques are all well-developed for the current high-performance computing environment, but the advent of exascale computers will be a disruptive change which will require new techniques for performance monitoring and analysis.

In this paper, I first introduce the models of parallel programming which are currently in wide use. I then discuss how existing systems collect performance information. I then discuss automated systems which make use of performance data once it has been collected: autotuning systems, which measure the performance of many implementations of a code to identify good-performing variants; automated modeling systems, which construct models from performance data by which the performance of code can be predicted without running it; and performance diagnosis systems, which reason about performance data to arrive at hypotheses about the causes of bad performance. I then discuss the challenges of the coming exascale era of high-performance computing, and discuss new parallel programming models which are emerging to meet those challenges. Finally, I discuss problems with existing systems for collecting and making use of performance data in the exascale era and describe the features necessary for future such systems.

2.1 Current Programming Models

All current supercomputers consist of many nodes, each of which contain many individual processors, which are often supplemented by accelerator devices such as GPUs; the two fastest such systems, Tianhe-2 and Titan, contain 3.12 million spread across 16,000 nodes and 560,640 cores spread across 18,688 nodes, respectively, with the former equipped with Intel Xeon Phi accelerators and the latter with NVIDIA K20 GPUs [216]. Thus we need ways of exploiting available parallelism both on-node and between nodes. By far the most popular solutions for this are OpenMP and MPI [165].

Shared Memory: OpenMP. OpenMP [171] is the most common method of exploiting on-node parallelism. It uses the *fork-join* model: programs begin executing sequentially, eventually *fork* into multiple threads of execution which operate in parallel before *joining* back into a single, sequential thread of execution (Figure 2). It uses a *shared memory* model: all threads of execution within a program share the same address space and access the same memory. It is *directive-based*: parallelism is expressed by taking what would otherwise be an ordinary, purely sequential program and annotating it with directives indicating which parts of the program should be executed in parallel and how access to shared memory should be managed. Thus the code

```
do_work();
```

can be made to run multiple times in parallel by adding an annotation

```
#pragma omp parallel
do_work();
```

causing multiple threads to be spawned, each of which execute the function `do_work` before joining, with the main flow of program execution continuing sequentially once all the threads have finished `do_work`.

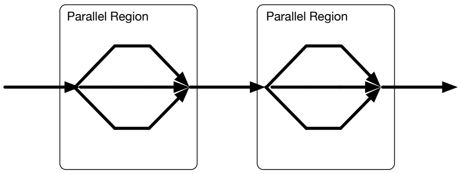


Figure 2. The Fork-Join model as used in OpenMP. There is ordinarily one thread of execution, which *forks* to become multiple threads in parallel regions. When exiting a parallel region, the threads *join* back into a single thread of execution.

When running several instances of `do_work` in parallel, it may happen that the separate instances attempt to use the same memory. OpenMP provides several annotations for controlling access to memory: `#pragma omp critical` marks sections of code which only one thread should be allowed to execute at a time. `#pragma omp single` marks sections which only one thread should execute *at all*, while `#pragma omp master` marks sections which a specific thread – the one which existed when the program started and will continue to exist after leaving the parallel region – should execute. `shared` and `private` clauses indicate whether threads should share one copy of a variable or should each operate on a local copy, while `reduction` clauses specify how per-thread local variables should be reduced to a single value which persists in the master thread after the end of a parallel region.

In the above example, every thread executes the exact same code, which is almost certainly not what we want – different threads should process different data. Threads can be distinguished by a thread number which can be retrieved with a call to `omp_get_thread_num`, so that threads can identify which data they should be processing, but the more common usage is to use work-sharing constructs which automatically distribute work to threads, so that if we have a loop

```
for(int i = 0; i < 1000; ++i) {
  do_work(x[i]);
}
```

we can add a directive

```
#pragma omp parallel for shared(x)
for(int i = 0; i < 1000; ++i) {
  do_work(x[i]);
}
```

which causes the iterations of the loop to be automatically distributed across the threads. Several clauses are provided which allow the programmer to customize this distribution.

Distributed Memory: MPI. MPI [73] is the most common method of exploiting between-node parallelism. It uses the *communicating sequential processes* model: multiple instances of a program begin executing simultaneously, but each instance executes sequentially. These processes coordinate by sending messages to one another (Figure 3). It uses a *distributed memory* model: every process has its own address space, so every process has its own copy of each variable and changing a variable in one process does not change the value in any other process. A process may change the state of another process only by sending it a message. It provides a low-level API: unlike OpenMP, which provides directives which modify execution of an otherwise sequential program, MPI programs contain explicit API calls which carry out communication.

MPI processes all execute the same code. Processes can distinguish themselves by calling `MPI_Comm_rank` to obtain their *rank*. Unlike OpenMP, this is the only way that processes can determine that they should process different data: there is no equivalent to OpenMP's loop constructs, so the programmer is responsible for explicitly partitioning work.

Messages can be point-to-point or collective. Point-to-point messages are sent by a process through a call to `MPI_Send`, whose arguments specify the source, size, type and tag of the data to be sent. A corresponding `MPI_Recv` call must be executed on the destination to receive the message. Both calls are blocking; neither the sender nor the receiver will continue executing until the communication has completed. This limits the possibility for overlapping communication and computation and creates the potential for deadlock when communication is cyclic, so nonblocking `MPI_ISend` and `MPI_IRecv` versions are also provided. A set of collectives are also provided for efficient communication between multiple ranks.

MPI also supports *one-sided communication*, in which data can be sent to (*put*) or retrieved from (*get*) without an explicit call on the remote rank, using Remote Direct Memory Access (RDMA). In this mode, memory must be pre-registered (`MPI_Win_create`) to make it a valid target of subsequent `MPI_Put` and `MPI_Get` calls. These calls are always nonblocking, and explicit synchronization (`MPI_Win_fence`) is required to ensure that the operations have completed before using the values sent or retrieved through one-sided communication.

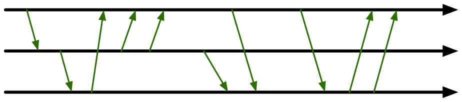


Figure 3. The Communicating Sequential Processes model as used in MPI. There are multiple threads of execution (black), each of which runs sequentially. They communicate with one another by sending messages (green).

Partitioned Global Address Space: UPC. A disadvantage to MPI is lack of orthogonality: local communication occurs through direct access to the local memory, using ordinary features built in to the language, while remote communication occurs through API calls. The *Partitioned Global Address Space* – or *PGAS* – approach uses a common syntax for local and remote communication [245]. The address space is global – every process can access memory in every process – but is also partitioned: every address is *owned* by a particular process, and a pointer consists not only of an address but also a tag indicating who the owner is. When a process reads or writes through a pointer to locally-owned memory, this is translated into a local memory address as normal. When a process reads or writes through a pointer to remotely-owned memory, this is translated into a message sent over the network which triggers a read or write of the address in its owning process and, in the case of a read, a reply message containing the value stored at the address.

Unified Parallel C [219] is a language which extends C99 [112] with support for *shared pointers* to a partitioned global address space and a variety of work-sharing constructs similar to those provided by OpenMP. As in MPI, multiple copies of the same executable are launched, and these execute the same code. In UPC, pointers and arrays can be declared `shared`, making them globally available. For example,

```
shared double a[3*THREADS];
```

declares an array `a` of doubles with three elements per thread (a UPC thread corresponds to an MPI rank) which is globally accessible. By default, ownership – or *affinity* – of memory in an array is assigned cyclically, so that the memory located at the address `a + i` is physically located on thread `i % THREADS`. Arrays can also be divided into blocks of elements which are distributed cyclically, or each thread can be assigned a contiguous block of the array.

Unlike MPI, UPC provides built-in support for partitioning work across threads. The `upc_forall` loop, when encountered by a thread, runs only those loop iterations which have affinity to the thread that encountered the loop. For example,

```
shared double x[N], y[N], z[N];
// initialize x and y
int main() {
  upc_forall(int i=0; i < N; ++i; i) {
      z[i] = x[i] + y[i];
    }
}
```

resembles an ordinary C `for` loop with the exception of an additional parameter to the loop. This parameter specifies the affinity, and the value of `i` here means that a thread encountering the loop will run all iterations for which `i % THREADS == MYTHREAD`. Like MPI, UPC provides synchronization primitives such as `UPC_barrier` and a variety of collective communication operations.

Accelerators. As noted above, the current generation of top supercomputers feature accelerators, as will the next generation of supercomputers which will be installed in 2017 and 2018. Accelerators generally feature a larger number of cores than general purpose CPUs, but each core is less capable than those in a CPU.

CUDA. NVIDIA GPUs are available with up to 4,096 cores, but these cores do not have all features typical of a CPU core: notably, cores are not capable of independently fetching and scheduling instructions. Rather, a group of cores share fetch and schedule hardware and always execute identical instructions during the same clock cycle, differing only in the memory addresses read and written by those instructions. Figure 4 shows the NVIDIA architecture: all of the cores share L2 cache and access to the memory and PCIe buses, while sets of 32 cores share L1 cache, fetch and dispatch units, registers, load-store units and Special Function Units, while each core has its own floating point and integer arithmetic units. AMD GPUs (Figure 5) use a similar architecture.

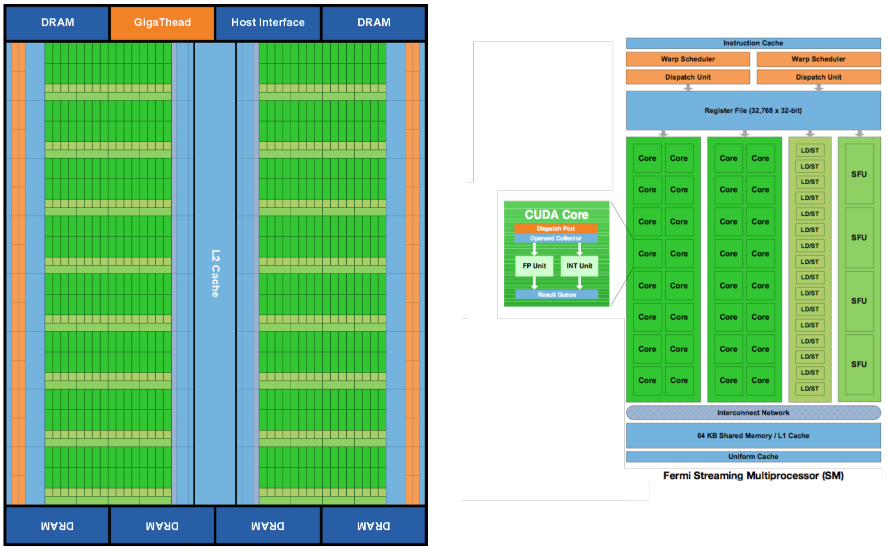


Figure 4. Architecture of the NVIDIA Fermi GPU family.

To allow programming NVIDIA GPUs, NVIDA developed CUDA [167], C language extensions and APIs for writing code which will execute on a GPU and for transferring data between host and GPU memories. CUDA kernels are C functions which are annotated `__global__`, indicating that they will run on a GPU but can be invoked from the host. A kernel function differs from an ordinary function in that many copies of the function will execute simultaneously. A given instance of the function must examine its local copies of the `blockIdx` and `threadIdx` variables to determine which portions of the input data it should process.

CUDA maintains separate memory spaces for the host and each device. Running a kernel on a device then involves the host explicitly allocating memory on the device (`cudaMalloc`), copying input data to the device (`cudaMemcpy`), specifying

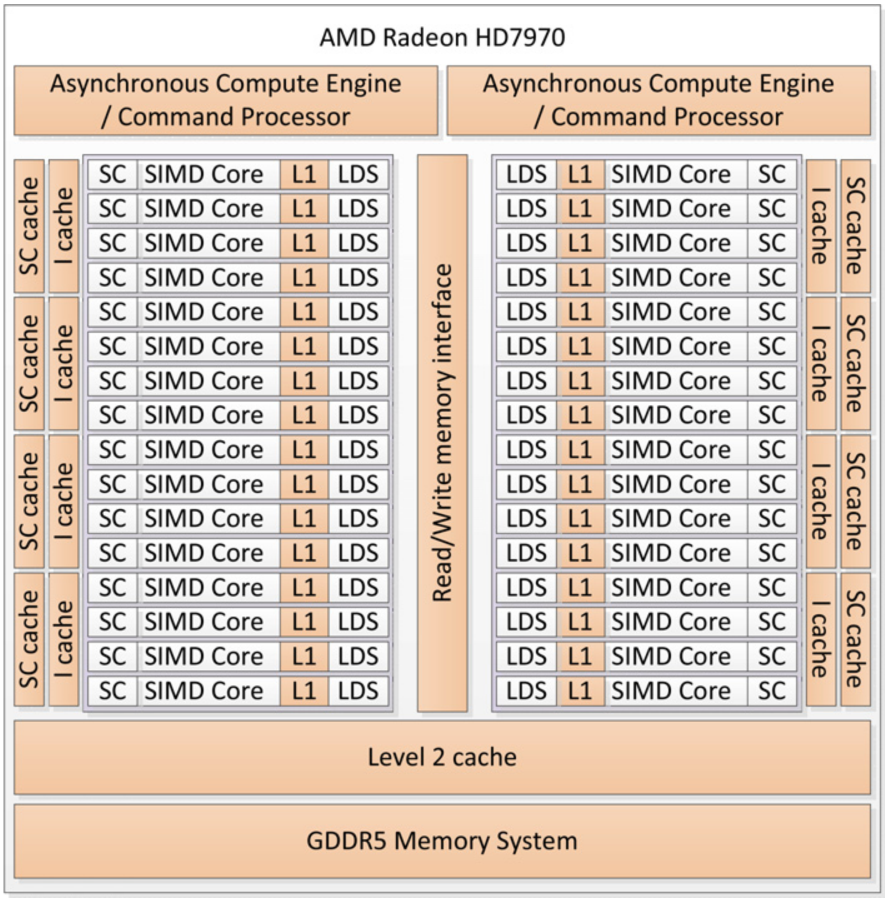

AMD Radeon HD7970
Asynchronous Compute Engine / Command Processor
Asynchronous Compute Engine / Command Processor
SC cache
I cache
SC
SIMD Core
L1
LDS
Read/Write memory interface
Level 2 cache
GDDR5 Memory System

Figure 5. Architecture of the AMD Radeon 7000 GPU family.

how the input data is to be partitioned into blocks, launching the kernel, and copying output data back onto the host.

Xeon Phi. The Intel Xeon Phi accelerator architecture features fewer cores than are found in GPUs (61 cores and 244 hardware threads) which are considerably more complex than GPU cores but which are still simpler than the cores typically found in a host processor [181]. The cores are connected together by a bidirectional ring bus, which they share with a distributed, globally coherent L2 cache (Figure 6). Each core features four hardware threads, can dispatch two instructions per cycle, and is required to switch between hardware threads once per cycle, which results in the requirement that enough work be available that instructions can actually be issued every cycle – if an insufficient number of threads are used, the issuing hardware will remain idle every other cycle. The cores use in-order execution but feature SIMD units with twice the width of current x86-64 chips.

Xeon Phi accelerators themselves run Linux and can be programmed through several mechanisms, including a native mode using traditional MPI and/or OpenMP, as well as an offload mode [166] which uses `pragma` annotations and/or language keywords to specify work which should be executed on the accelerator, from which the compiler will automatically synthesize the necessary memory copy and kernel launch code.

Cross-architecture Programming Models. There are several projects aimed at providing programming models which allow a single code to target multiple types of accelerators. OpenCL [200], an industry standard maintained by the Khronos group, is one such model. Its structure and syntax are similar to those of CUDA, but with additional abstractions for devices, compute units, processing elements, and private, local and global memories which a driver for a device maps onto physical

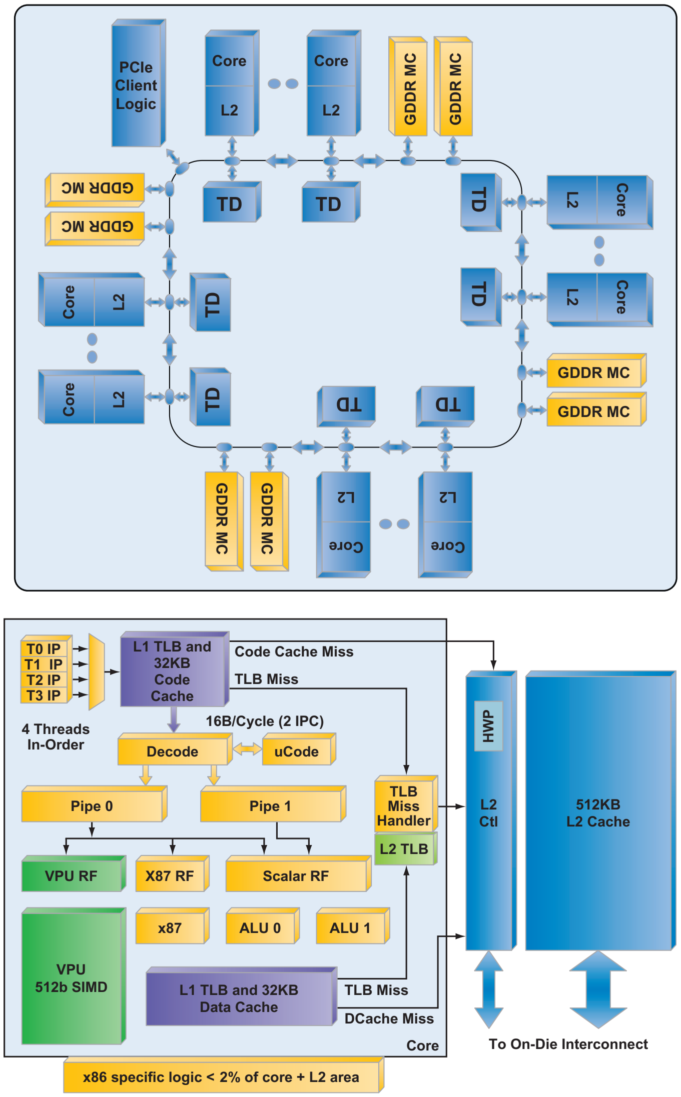


Figure 6. The architecture of the Intel Xeon Phi *Knights Corner* family. Images provided by Intel.

hardware. Drivers are available for many devices, including NVIDIA and AMD GPUs, Intel Xeon Phis, Intel and AMD CPUs, as well as FPGAs [64, 55]. Since the target hardware is not necessarily known at compile time, kernel code is stored as a string and is provided to the device driver for compilation just prior to kernel invocation.

In addition to low-level approaches to portability, higher-level approaches also exist. OpenACC [87] is a `pragma`-based model for device programming, similar to OpenMP, in which loops are annotated to indicate that their iterations should execute in parallel on accelerator devices. As accelerator devices have separate memory spaces from the host, additional `data` directives are added to specify data to be copied and allocated on the device. OpenMP itself is also being extended with device support through `target` directives [136].

2.2 Capturing Performance Data

Once we have a parallel program – most likely written using one of the programming models discussed in Section 2.1 – how can we determine whether it performs well? To do this, we must first determine when events occur during execution of the program by means of *instrumentation* [162].

To instrument code, we cause additional instructions to be executed which record events and facts about those events, such as the time or number of cycles elapsed between two events. There are several ways to accomplish this. We can use *source code instrumentation*, where we modify the source code, inserting function calls at the beginning and end of functions or around loops. In order for facts about events to be useful, we must be able to map events back onto the source code so that we know where changes should be made to address any performance problems found. Directly modifying the source code allows us to most easily map events back onto source, since each event generated by inserted code can be given a unique name. However, source code instrumentation requires that we have the original source available, and that we are able to parse the source code in order to modify it. Source code instrumentation is supported by systems such as TAU [192] and VampirTrace [125].

Alternately, we can modify the binary, either through rewriting prior to execution or dynamically at runtime, through libraries such as Dyninst [180] as used by TAU, or through performance analysis tools which directly implement binary analysis, such as HPCToolkit [2]. Such systems analyze the binary, identifying entry and exit points for functions and inserting calls to log events. This type of approach makes it straightforward to dynamically adjust instrumentation points at runtime through self-modifying code, allows instrumentation of binaries for which the original source is not available or which is written in a language for which automated source instrumentation tools are not available, and eliminates overhead for runs in which instrumentation is not desired (in which case the binary is run unmodified). However, it is more difficult to map events back onto the source code, as compiler optimizations applied in creating the binary may disrupt the relation between instructions and the source line which caused them to be emitted.

For systems such as MPI, OpenMP, and UPC which feature runtime libraries, instrumentation can be performed at the runtime level rather than the application level. This can be accomplished by preloading a library which exposes the same interface as the actual runtime which logs events before forwarding function calls to the actual runtime. Such interposed libraries include mpiP for MPI [222] and ompP for OpenMP [75]. Runtimes can also expose callback interfaces through which a performance monitoring tool can register functions which will be called by the runtime when certain events occur, such as OMPT for OpenMP [69], CUPTI for CUDA [169, 144], and the OpenCL event profiling interface [110] and GPUPerfAPI [3] for OpenCL.

Finally, we can use *sampling*, where we request that an interrupt be called periodically or when a hardware performance counter reaches a certain value or overflows. The interrupt transfers control to the performance monitoring tool, which can record the address which was being executed prior to the interrupt. Sampling allows fine control over the tradeoff between overhead and error: by increasing the sampling rate, we get a more precise picture of what the program is doing and are less likely to miss events which occur infrequently, while at the same time we increase the proportion of time spent running the monitoring routines instead of program code. By decreasing the sampling rate, we reduce overhead at the cost of increased likelihood of missing infrequent events.

Any of these techniques – source-level instrumentation, binary instrumentation, library interposition, runtime instrumentation and sampling – can be used to generate events. When events are generated, what should the performance monitoring system do with them? Generally, they will be used to generate either a profile or a trace [162]. In *profiling*, events mapped to a particular code region are used to create an aggregate measure of performance for that code region [82]. If function-level instrumentation is used, then, the profile might record the number of calls to the function and the time spent in that function aggregated across all calls to it. There are different choices to be made as to the level at which aggregation occurs. For example, in *call-path profiling* [93], the performance monitoring system stores separate profiles for a function depending on the call path through which the function was reached, so that if `A()` calls `Z()` and `B()` calls `Z()` we would see two separate profiles for `Z()`. This can help account for input-dependent behavior, as different uses of a function may use different data and thus exhibit different performance characteristics. In *phase-based profiling* [143], separate profiles are stored for *phases* of an application, such as particular algorithms or iterations of iterative algorithms.

In *tracing*, events are simply separately recorded along with a timestamp [82]. In a distributed system, traces collected on separate nodes must be merged so as to maintain ordering on systems which do not have synchronized clocks. Traces provide a large amount of information with which to diagnose performance problems and allow phases of program execution to be automatically discovered: given the full list of events, we can infer causality between events. However, the volume of data generated can be exceptionally large, particularly for runs using large portions of a supercomputer. Traces grow both with the number of processes used (more processes each generating events) and with the runtime of the application.

2.3 Autotuning

Once we have a mechanism to acquire performance data, how can we use that data to improve performance? One approach is *automatic performance tuning*, or *autotuning* [20]. Autotuning arises from the idea that the best-performing implementation of some code is not the same everywhere: it depends on the architecture of the processors on which the code will execute, the operating system, networking infrastructure, and other system parameters [233], on properties of the input data such as size [194] or the number and distribution of nonzero elements in a sparse matrix [193], and on the interaction between system parameters and input data. Many runtimes, such as MPI and OpenMP, also have parameters which can be set to control scheduling of work or use of network resources [36]. In an autotuning system, we generate code variants and/or modify runtime parameters and perform instrumented runs, which we use to determine which variants and parameters result in the best performance. The space of possible variants and parameters is very large for all but trivial problems, so heuristic search algorithms are used to avoid exhaustive enumeration and testing of the entire space.

Basu et al. [20] identify three categories of autotuning systems: *self-tuning libraries*, in which autotuning support is built directly in to a library and is run at install time or runtime; *programmer-directed autotuning*, in which the programmer of a piece of software exposes runtime parameters to a search system; and *compiler-directed autotuning*, in which a library of code variants are generated by a compiler or source-to-source translator. They envision a system in which all of these techniques can be combined through the use of a centralized search engine and performance database (Figure 7).

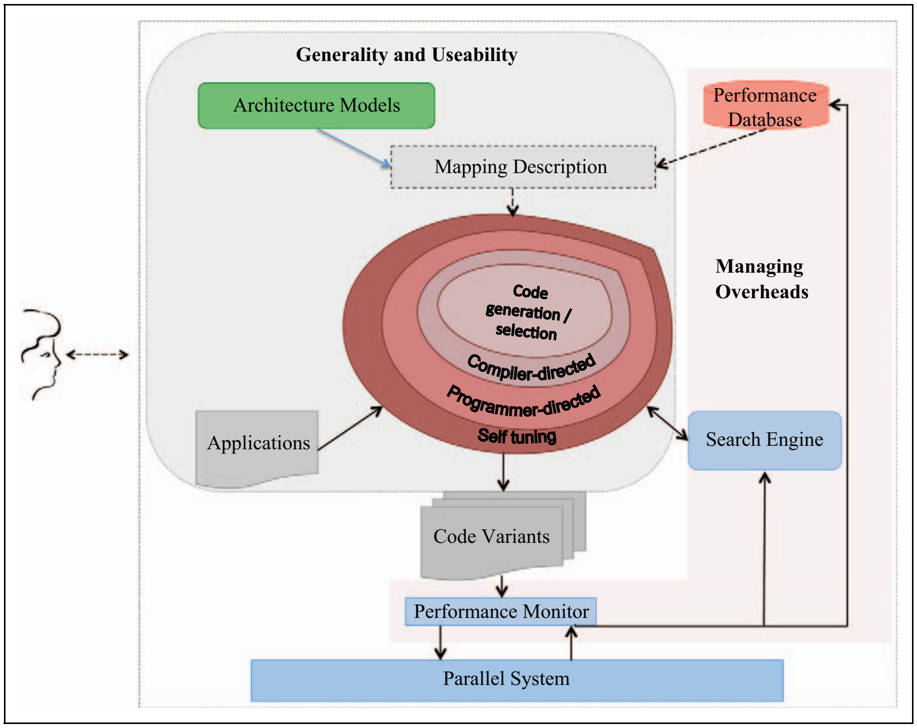


Figure 7. A hypothetical architecture for a unified autotuning system, in which multiple types of autotuning are present in a single application and share a search engine and performance database. From [20]

ATLAS. One approach to autotuning is to build autotuning support directly into a library. An early and widely-used such library is the linear algebra library ATLAS [229] (Automatically Tuned Linear Algebra Software). Traditionally, hardware, operating system and compiler vendors have generated hand-tuned linear algebra routines for developers using their products. ATLAS represents a different approach, shipping a variety of parameterized function implementations which are tested during compilation. The developers of ATLAS identify four requirements for the application of empirical optimization [228]:

- Isolation of performance-critical routines.
- A method of adapting software to differing environments.
- Robust, context-sensitive timers.
- Appropriate search heuristics.

ATLAS performs its tuning at compile time. This is beneficial in that it does not introduce any delays at runtime due to the need to select an implementation at that time, but this also limits the ability of ATLAS to adapt to a changing execution environment or to the input data, which is only known at runtime (for example, to adapt to different sizes of input matrices, if a given program tends to use matrices of one of a few fixed sizes.)

FFTW. Another approach is that used in FFTW3 [74], a Fast Fourier Transform library. In FFTW, the user of the library invokes the library with a description of the problem to be solved (e.g., which discrete transform is to be calculated) and the sizes and memory layouts of the input arrays. FFTW includes code, called the *planner*, which will then test many different functions for calculating the desired transform on problems of the indicated size and layout, and select and return the best-performing one.

This technique allows FFTW to adapt to changes to its execution environment (such as in the case of migration) and to properties of the input data. However, if only a small number of transforms of a particular type and for particular input types are performed, then the cost of performing the tests will outweigh the increased performance from using tuned variants, and overall program execution time will be slower.

SPIRAL. Spiral [177] is a general-purpose digital signal processing library in which DSP algorithms are expressed in a domain specific language, SPL, which is ultimately translated into C or Fortran. Optimizations can be applied at both the DSL and target language levels and can take into consideration properties of the domain that enable optimizations that are not generally applicable to all domains. Some optimizations use a static cost model to determine whether they should be applied, while others use search algorithms to explore the space of optimizations, for which exhaustive and random search, dynamic programming, evolutionary algorithms and hill-climbing search algorithms are provided.

The evolutionary algorithm mode is particularly interesting: genes are represented as *ruletrees*, which specify the recursive structure of a transform with leaf nodes representing particular implementations. Mutations are made by swapping an implementation for another, while cross-breeding occurs by swapping subtrees. Additionally, SPIRAL uses empirically-generated models by timing subtrees within a ruletree.

OSKI. Oski [223] is a sparse linear algebra kernel library which uses a similar approach to FFTW, performing tuning at runtime based upon known input parameters. The library provides a set of functions for specifying *hints* about input sizes, coefficient values, data formats, and the number of times different operations are expected to be performed. The tuning process can then generate specialized variants, and, because the estimated frequency of operations is provided, OSKI can determine how much time should be spent on tuning particular operations based on whether it is likely to be executed enough times to amortize the cost of tuning.

pOSKI [32], a system for generating optimized sparse matrix-vector multiplication routines, combines offline autotuning with model-driven online autotuning combined with a history database. The offline tuning, which happens when the library is initially installed, tests combinations of storage format (CSR or BCSR), size of register blocks, prefetching policy, and SIMDization policy for a set of likely block sizes. At runtime, when the actual matrix is available and its sparsity therefore known, a simple model is used to select a block size, and therefore an implementation from among the pre-generated set of optimized implementations.

Orio. Orio [94] is an autotuning system providing pluggable code generators and search algorithms and using an annotation-based approach to specifying code transformations. Input code in a language such as C or Fortran is annotated with special comments indicating that the annotated code should be replaced with code generated by Orio according to specified transformation. A loop, then, can be annotated with a Loop transformation specifying that a version of the loop written in a restricted subset of C or a domain-specific language should be unrolled by some factor and tiled by some factor.

These annotations can be left with parameters (such as tile factor) unspecified and be wrapped in a *tuning specification*, which specifies the range of values valid for each parameter, what search algorithm should be used, and how the kernel can be tested in isolation: how input data can be generated, and the sizes of input data which should be tested. Each such tuning specification then describes a set of experiments, the output of which are tuned variants which are inserted into the source code, replacing the original implementation. As the tuning specifications and annotations are comments, the original source code can also be compiled unmodified to give the original implementation. Transformations are also available to generate CUDA [145] and OpenCL [37] code for use on accelerators.

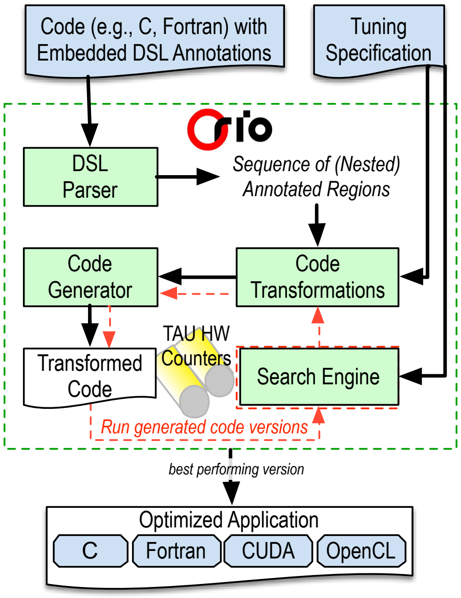
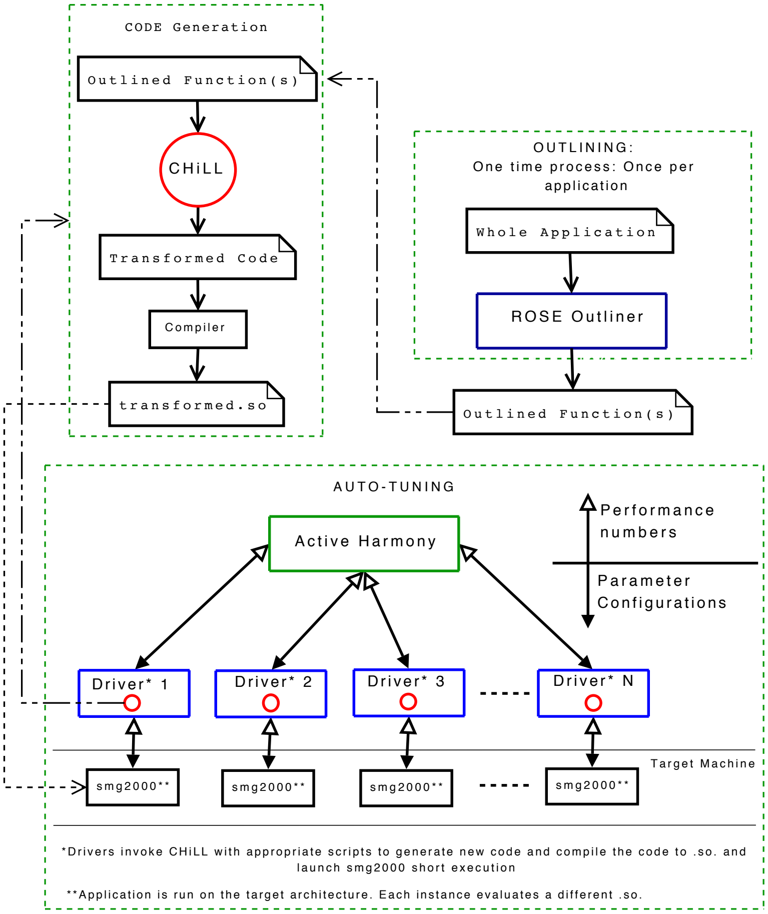


Figure 8. The architecture of Orio. From [37].

CHiLL + Active Harmony. Active Harmony [215] is a general purpose search engine capable of rapidly exploring the parameter search space by testing multiple hypotheses in parallel, using the Parallel Rank Ordering algorithm to evaluate potential parameters, which is used both for online tuning of application and runtime parameters and for offline tuning by providing parameters to an external code generator. The user can specify parameters, ranges for the parameters, and constraints restricting the values parameters can take on. Active Harmony runs using a client-server architecture, in which a centralized Harmony server communicates with, and provides parameters to, multiple clients running on different nodes in a cluster. Using Parallel Rank Ordering, the system can provide different parameter values to different nodes in the cluster, allowing it to evaluate the search space in parallel. When used with a code generator, code servers can also be configured, which perform compilation of code variants and distribute compiled object files to the execution nodes [98].

Active Harmony has been used with CHiLL [48], a code variant generator which uses a "recipe" of high-level loop transformations which are applied together to generate variants of a loop. CHiLL uses the ROSE compiler [178] internally to parse code and applies transformations by making modifications to the ROSE AST. It uses a polyhedral model of loop transformations, in which the order of operations within nested loops are viewed as points inside a polyhedron, from which semantically-equivalent loops evaluating nests in different orders can be generated by applying geometric transformations to the polyhedron representing loop iterations [86]. CHiLL recipes can be parameterized, and autotuning can be performed by searching the space of parameters to available recipes. Transformations are available for generation of CUDA code through CUDA-CHiLL [186]. The combination of CHiLL and Active Harmony has also been used with the ROSE outliner, a system which extracts regions of code within a function into independent functions which can be separately tuned [214].

Periscope. The AutoTune project [159] is developing the Periscope Tuning Framework, an extension to the earlier Periscope [22] performance analysis and diagnosis tool, described in more detail in Section 2.5. The architecture of PTF is shown in Figure 10. In PTF, tuning plugins are registered which interact with a set of *scenario pools*. Plugins can insert new scenarios into the *created scenario pool*; can pull



*Drivers invoke CHiLL with appropriate scripts to generate new code and compile the code to .so. and launch smg2000 short execution

**Application is run on the target architecture. Each instance evaluates a different .so.

Figure 9. The architecture of an autotuning system using ROSE to outline functions, CHiLL to generate code variants, and Active Harmony to direct the search process. From [214].

created scenarios, process them, and insert the result into the *prepared scenario pool*; can create experiments from prepared scenarios, inserting them into the *experiment scenario pool*; and, once the execution engine has run an experiment from that pool and made it available in the *finished scenario pool*, can pull the results and process them to create a human-readable report.

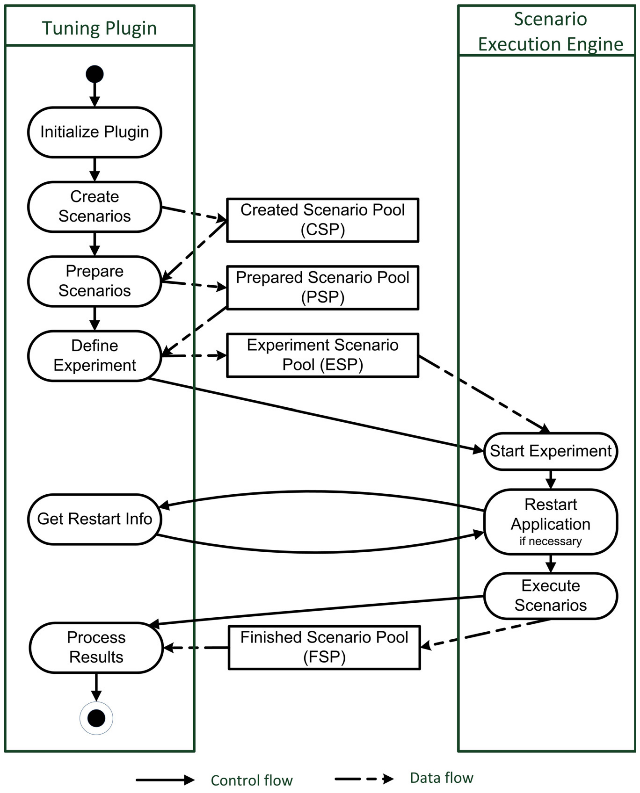

Tuning Plugin
Scenario Execution Engine
Initialize Plugin
Create Scenarios
Created Scenario Pool (CSP)
Prepare Scenarios
Prepared Scenario Pool (PSP)
Define Experiment
Experiment Scenario Pool (ESP)
Start Experiment
Get Restart Info
Restart Application if necessary
Execute Scenarios
Process Results
Finished Scenario Pool (FSP)
Control flow
Data flow

Figure 10. Architecture of the Periscope Tuning Framework. From [159].

Insieme: Multi-Objective Optimization. The Insieme framework [90], unlike most auto-tuning frameworks, is designed specifically for *multi-objective optimization*, which allows for objectives such as "minimize execution time used subject to constraints on the number of cores and the amount of energy used". When multiple objectives are present, the solution found is not a single best-performing configuration but rather a Pareto frontier, a set of points for which no objective can be improved without degrading some other objective. The best configuration given some particular set of tradeoffs is then always found on the Pareto frontier. Genetic algorithms map well onto the problem of finding the Pareto set [66], particularly differential evolution techniques in which the rate of evolution for different parameters itself evolves.

Collective Tuning. An alternate approach is used by Fursin et al. in their *Collective Mind* project [76]. Rather than enforcing a strict schema, they allow the user to encode *measured characteristics*, *choices*, *features* and *system state* in JSON format [28], which can be used without requiring that a schema be provided. When in the course of a project a schema becomes necessary, it can be provided, also in JSON format. The user can provide modules which mediate between Collective Mind data and external tool. These modules are gradually composed into a workflow which specifies the overall experiments to be done. Collective Mind encompasses the earlier *Crowdtuning* [155] and *Collective Tuning* [77] projects, which made available a more restrictive central repository for performance data from the MILEPOST GCC compiler. The compiler generates a library of compiled versions of functions with different optimizations applied. At runtime, when a function of interest is executed, either the currently known best version or a different, proposed version is randomly selected and profiled, with the timings being sent to the central repository.

Online Adaptation. The Abstract Data and Communication Library [78] (ADCL) is used for runtime tuning MPI applications. A library of variant implementations of a communication routine, called a *function set*, is defined either by the library designer or the developer of the application. ADCL then uses either brute-force search or parameter-at-a-time search to evaluate the variants. In one case study [79], it was used to select from a set of neighborhood communication routines (in which each rank communicates with six neighbors in each iteration), which varied along three axes: the number of simultaneous communication partners (e.g., pair-at-a-time or all-to-all), mechanism for handling messages with contents not contiguous in memory (e.g., by packing the data into a contiguous array before communicating, or by defining a custom data type), and the underlying data transfer routines used (e.g, blocking vs nonblocking communication, two sided vs. one sided communication, etc.). Different variants were selected for different architectures, network hardware, and problem sizes. Interestingly, the best-performing variant for some configurations was the worst-performing variant for another, demonstrating the importance of autotuning in this case. The library includes pre-defined function sets for the standard MPI collectives [23].

A later version of ADCL adds the ability to focus the search process using data from previous runs [71]. The authors identify two primary obstacles to the use of historical data: that the system may not have stored performance results for the particular execution environment and problem now being encountered, and that changing conditions (such as degree of congestion on the network, or the physical location of ranks as assigned by a batch scheduler) mean that even if the system is encountering a problem which has been encountered before, the best performing variant as determined in the past may not be the best performing variant now. To work around these problems, ADCL uses a distance metric to select good-performing variants from history which are, according to that metric, most similar to the variant now being encountered, and requires that performance be within a user-specifiable tolerance of that recorded in the history file. If not, search is repeated.

The Open Tool for Parameter Optimization [36] (OTPO) uses search algorithms from ADCL to tune parameters exposed by the OpenMPI runtime. In

OpenMPI, many runtime tasks are delegated to modules, which implement different versions of communication algorithms (such as collectives) and map MPI operations onto lower-level network operations (such as for TCP, InfiniBand, Cray Gemini/Aries, etc.). These modules expose a set of tunable parameters, called MCA parameters, of which a typical installation will have several hundred. Using search algorithms from ADCL, OTPO searches for parameters giving the best performance, as measured by latency or bandwidth of network operations.

OTPO finds good MCA parameter values, but requires a large number of evaluations to do so. To reduce the number of evaluations needed, Pellegrini et al. [174] evaluate the effect of different parameters on performance at compile-time and use this data at runtime to tune only those parameters most likely to have large performance effects. During installation of OpenMPI, a set of kernels, chosen to approximate the communication patterns of typical applications, are run with randomly-chosen parameter values. ANOVA is then used to identify which parameters have the greatest impact on performance.

2.4 Performance Modeling

We can also use performance data to attempt to construct *empirical models* which allow us to predict performance of the code on other systems or datasets. Such models can then be used to guide autotuning or performance diagnosis.

Prophesy [208, 207, 241] is an integrated system for automatically generating analytical performance models, comprising a source-code instrumentation component [241], a database component [207], and a model builder component [208]. Performance data is collected at the basic block level and stored in the performance database as a hierarchy, in which applications are made of modules, modules are made of functions, and functions are made of basic blocks, allowing for measurements to be viewed at an appropriate level of abstraction for the current task. The database stores information on applications (name, version, etc.), executables for applications (how it is compiled, what libraries it uses, and static analysis results such as control flow data), run information for particular runs of applications (machine and input information), and hierarchical performance measurements.

Prophesy then implements three modeling techniques: curve fitting, parameterization, and composition. Curve fitting is fully automated, while parameterization and composition require additional input from the user. Curve fitting attempts to model the performance of the application or functions of the application in terms of input parameters (such as size), but does not incorporate system-specific features and therefore can only be used to evaluate intra-system scalability and not to predict performance across systems. Parameterization incorporates coefficients representing system-specific parameters, but requires manual annotation of kernels to identify and count operations. Composition combines models stored in the database to allow application performance to be represented as the composition of models for the application's constituent kernels. Pairs of kernels are evaluated to determine the effect of running one kernel after another[1], resulting in an *coupling coefficient* C_{ij}, the effect on the performance of kernel j when it runs after kernel i. C_{ij} equals 1 when there is no interaction, is less than 1 when performance of j improves (such as when running i has resulted in data used by j being loaded into the cache), and is greater than 1 when performance of j is degraded.

An early comparison of empirical autotuning with model-based parameter selection was performed by Yotov et al. [246]. Looking specifically at matrix-matrix

---

[1] This is the formulation in the paper, although the same concept could also be used for two kernels running simultaneously, such as in a task based system. Scaling the technique to many simultaneous kernels may present problems, however.

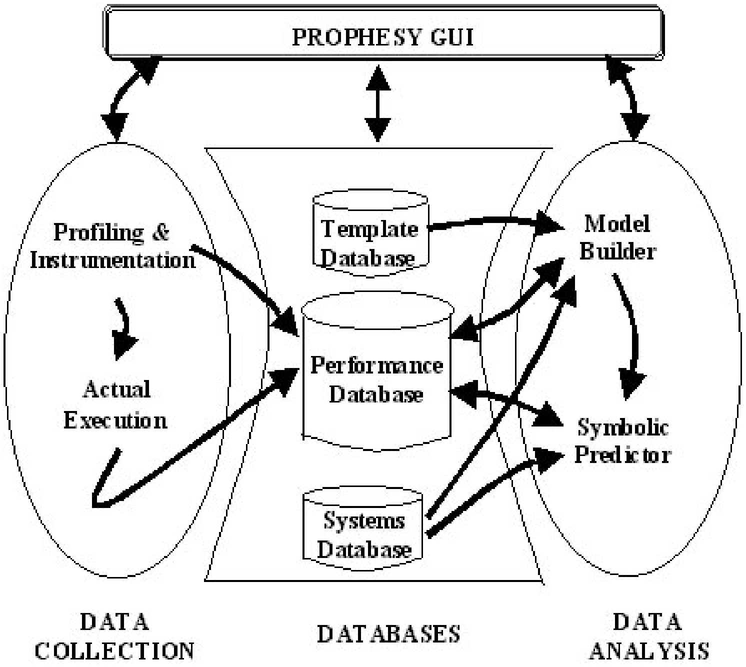


Figure 11. Architecture of Prophesy, from [208].

multiplication codes as generated by ATLAS, described in Section 2.3, they develop a simplified model of how cache behavior is affected by parameters to the matrix multiplication code generator and substitute the search module of ATLAS with the model. On two systems (SGI and Intel) their model yields performance within 1% of that produced by the full ATLAS search, but reduces installation time to 35% of its original value. On a third system (Sun) the model-predicted variant has 20% worse performance than the empirically-determined version. This demonstrates the promise of model-driven approaches, but also its limitations: much effort went into developing the models, which are specific to only one ATLAS routine.

Modeling can also be used in combination with autotuning, rather than strictly as a replacement. One of the major uses of modeling in combination with empirical autotuning is to avoid evaluating variants which a model predicts will have poor performance, thereby focusing the search on variants expected to perform well. Balaprakash et al. [18] used an active learning [190] technique customized for autotuning on HPC systems. They observed that a major problem with existing parallel active learning techniques was that when such an algorithm suggests multiple points to evaluate, the result for one such point can dramatically reduce the information gained from evaluating the remaining points in the proposed set, resulting in wasted effort evaluating code variants corresponding to such points. They modify the algorithm to attempt to avoid suggesting such points by 1) selecting an initial point x_i, 2) retraining the classifier assuming that the prediction for x_i was correct, and 3) selecting another point only from among those points whose informativeness was not substantially reduced by retraining.

Sarangkar and Qasem [187] describe MATS (**M**odel-driven **A**daptive **T**uning **S**ystem), an autotuning system which uses simple architectural models to constrain the set of transformation parameters to consider. Based on static code analysis to calculate reuse distance and models of effective data and instruction cache capacity, register set, and TLB size, parameter values for loop tiling, fusion, fission, interchange, and unroll are selected so as not to violate a user-specified tolerance value, which express, for example, that number of cache misses in considered variants should be no more than some percentage worse than the optimal value.

GROPHECY [156] predicts whether a CPU code is amenable to implementation on GPUs using an analytical model to determine whether the code is compute-bound or memory-bound. To do this, the user must first manually convert the CPU code into a *code skeleton* which lists only loops, memory loads and stores, and generic compute instructions. The skeleton is then converted into a set of possible GPU skeletons parameterized by many of the same parameters used in GPU code generation by autotuning frameworks. Instead of generating and running code, the model is used to estimate memory usage patterns.

Models need not attempt to determine the absolute performance of a code. In autotuning and runtime adaptivity, determining the expected performance of one code relative to another is useful. Models need not be based on performance at all. For example, Tang et al. [204] develop an empirical model of *contentiousness* and *sensitivity* when jobs are co-scheduled on a system and thus share resources. *Contentiousness* is the capacity of a program to degrade the performance of programs with which it is co-scheduled, while *sensitivity* is the propensity of a program to have its performance degraded when co-scheduled with a program of high contentiousness. These properties are distinct because contentiousness results from mere *use* of a shared resource, while sensitivity depends on a program *benefiting* from its use of shared resources. A program which reads large amounts of data from memory, processes it once, and never reuses data will make use of the caches, but will not gain a performance benefit from cache use due to lack of reuse. Such a program is nonetheless contentious. A program which reads a small amount of data and processes it repeatedly benefits greatly from cache use, and is therefore highly sensitive to other programs' use of the cache, whether or not those other programs benefit themselves from using the cache. The authors identify hardware performance counters (L2 and L3 cache lines input rate) and use regression to construct architecture and application-specific models which give relative contentiousness and sensitivity of applications. A scheduler can then use these to schedule high-contentiousness applications only with low-sensitivity applications.

Brainy [115] constructs architecture- and input-sensitive models for selecting the best C++ STL data structure for a given workload. For each architecture, a set of

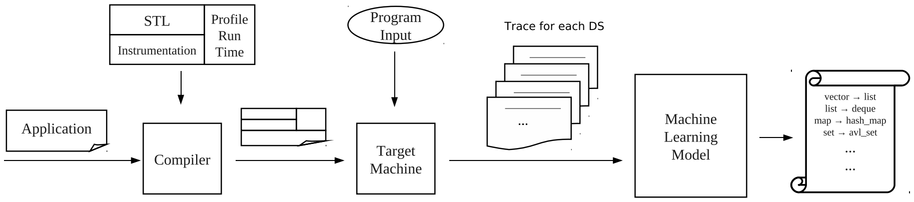


Figure 12. Architecture of the Brainy data structure selection system, from [115]

input programs are generated, instrumented, and tested, with each input program parameterized by the number of calls to each STL container interface function (e.g, *i* insertions, *j* finds, *k* in-order traversals, etc.) This allows the training set to include entries representative of a wide range of use cases. Timing and hardware performance counter data are collected for each call. These data are then used to train an artificial neural network which is used to predict the best-performing data structure for new applications based on the number of calls each makes to the various API functions. The architecture of Brainy is shown in Figure 12.

Rather than training a classifier based on program and system features, an alternative approach is to use clustering to identify programs with similar variation in performance across systems or systems with similar variation across programs. Such an approach was used by Cammarota et al. [33], who consider only program execution time, stored in a matrix M such that $M_{i,j}$ is the execution time of program i on system j. Having collected times for many programs on many systems, hierarchical clustering is used to group programs and systems according to similarity.

A major challenge with machine learning-based technique is in the selection of features. Leather et al. [132] automatically generate and test features using a genetic algorithm approach. A set of mathematical operators and functions operating on the compiler's intermediate representation are provided, together with a grammar describing how expressions using them can be formed. Every expression yields a real number. Genetic algorithms are then used to create new expressions from the existing population. Each proposed expression is tested by training a classifier using it as a feature and determining whether, and by how much, the addition of the feature improved the performance of the classifier. The degree of improvement is used as fitness. To evaluate this technique, the authors considered loop unrolling, exhaustively searching the space for a set of benchmark applications to determine the optimal value, and using the technique to create features, which performed better than human-selected features.

2.5 Performance Diagnosis

Autotuning, as described above, involves trying many variants or parameters, measuring their performance, and identifying variants and/or parameters that lead to good performance. Another approach to improving performance is *automatic performance diagnosis*, in which, rather than simply test a large number of variants, we analyze performance data from one run or a smaller set of runs and attempt to identify the specific *causes* of performance problems, so that we can develop targeted solutions to those problems.

Online Performance Diagnosis. Online performance diagnosis is the process of identifying performance problems *during* the run of a program. It is most useful for large-scale and/or long-running jobs in which collecting and making use of traces is not feasible.

Paradyn [160] is an early online performance diagnosis system designed to identify performance problems within a single run of a program, while minimizing the disruption it itself causes. It is based on a process of iterative search through a search engine called the Performance Consultant, which refines hypotheses, and on dynamic instrumentation: instrumentation is added at runtime when a hypothesis is being evaluated and, when the evaluation is done, the instrumentation is removed.

Search proceeds along three axes - “why”, “where”, and “when”. Along the “why” axis, the system attempts to refine hypotheses; an example of a hypothesis hierarchy is shown in Figure 13. In that example, the system will first insert instrumentation to determine if a synchronization bottleneck is present. If not, it moves to a sibling hypothesis. If so, it will insert more specific instrumentation to test causes of the overall problem – are synchronization operations too frequent, or do synchronization operations take too long, *etc.*. Along the “where” axis, hypotheses are localized to resources, such as places in the program’s code, nodes, particular synchronization objects, *etc.*. Search initially occurs at a high level in these hierarchies – such as, “does the *entire program* suffer from synchronization bottlenecks?” If so, the search is refined to locate parts of the program which suffer from synchronization bottlenecks and those which do not. Along the “when” axis, the system considers phases of execution, as performance problems may exist during some phases but not others.

Paradyn can use information from previous runs to focus future searches on the same code [122]. Inserting instrumentation for bottlenecks which are unlikely to exist unnecessarily perturbs performance, so hypotheses which have been disproved in many prior runs can be pruned from the hypothesis tree. Similarly, hypotheses which have proved true in many prior runs can be promoted so that they are searched earlier during program execution, allowing the most likely hypotheses to be tested even in short runs.

The original implementation of Paradyn is somewhat limited in scalability because the search process is centrally directed: one node is responsible for initiating instrumentation on all the nodes in the system, for processing measurements from all the nodes, evaluating hypotheses, and selecting new hypotheses to test. To increase scalability, a Distributed Performance Consultant was developed [184]. Rather than one central search agent, each node runs its own agent which can communicate with other agents as necessary. In order to determine whether a hypothesis holds for the whole application, neighboring nodes communicate to determine whether a property holds for a local neighborhood. Graph clustering is used to identify neighborhoods with similarly properties, and these summarized data are propagated to other nodes, in order to eventually give an approximate representation of global behavior.

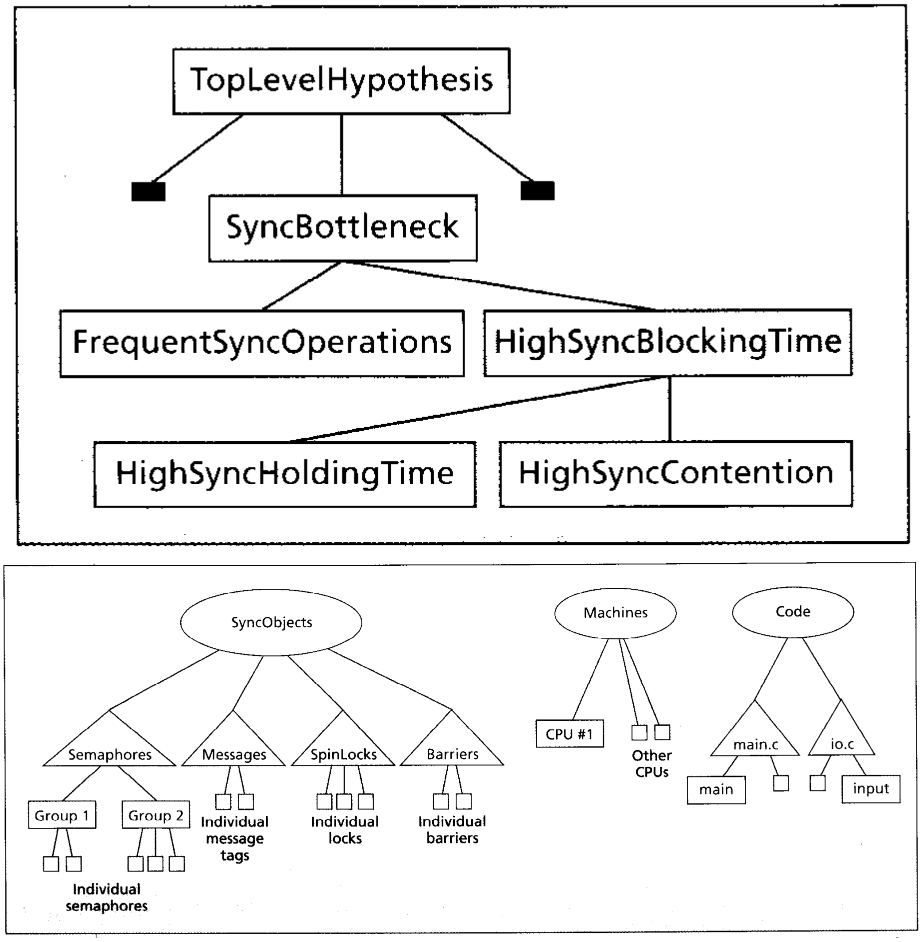


Figure 13. Examples of the Paradyn "why" and "when" hierarchies, from [160].

PERISCOPE [22] is an extensible performance diagnosis system based on a set of interacting agents. Its architecture is shown in Figure 14. Agents consist of several parts: the *search strategy* takes input from source code analysis and previous experiments and produces *candidate properties*, which are properties that would hold if the performance problem detected by the agent exists. These candidate properties are used to formulate experiments, which, when run, result in *measurement requests* being sent to the measurement system, describing what is to be measured (e.g., a set of of PAPI counters for a particular loop). When the results of the measurement request are available, they are stored in a performance database and the candidate property is evaluated in light of the new data. If the property holds, it is added to a set of *proven properties*, which are available to the search engine for its use in formulating new candidates. When no more candidates can be generated, the proven properties are analyzed to determine whether the performance problem is present or not.

Trace Based Systems. Wolf et al. [238] developed a system, KOJACK, to automatically diagnose performance problems in MPI and OpenMP codes. Programs are instrumented so that each process writes events to a process-specific log which are merged at program termination. Events which are logged include MPI sends, receives, and collectives, entry into and exit from OpenMP regions, and acquisition and release of OpenMP locks. A library of rules is constructed specifying patterns which indicate potential causes of performance problems. For example, one rule specifies that when a receive event is encountered while processing the event log, the corresponding send event should be located and the time between send and receive calculated to determine whether a "late sender" problem occurred, where an `MPI_Recv` call was made prior to the corresponding `MPI_Send`, resulting in the receiving process unnecessarily waiting. These rules are applied to the merged event log.

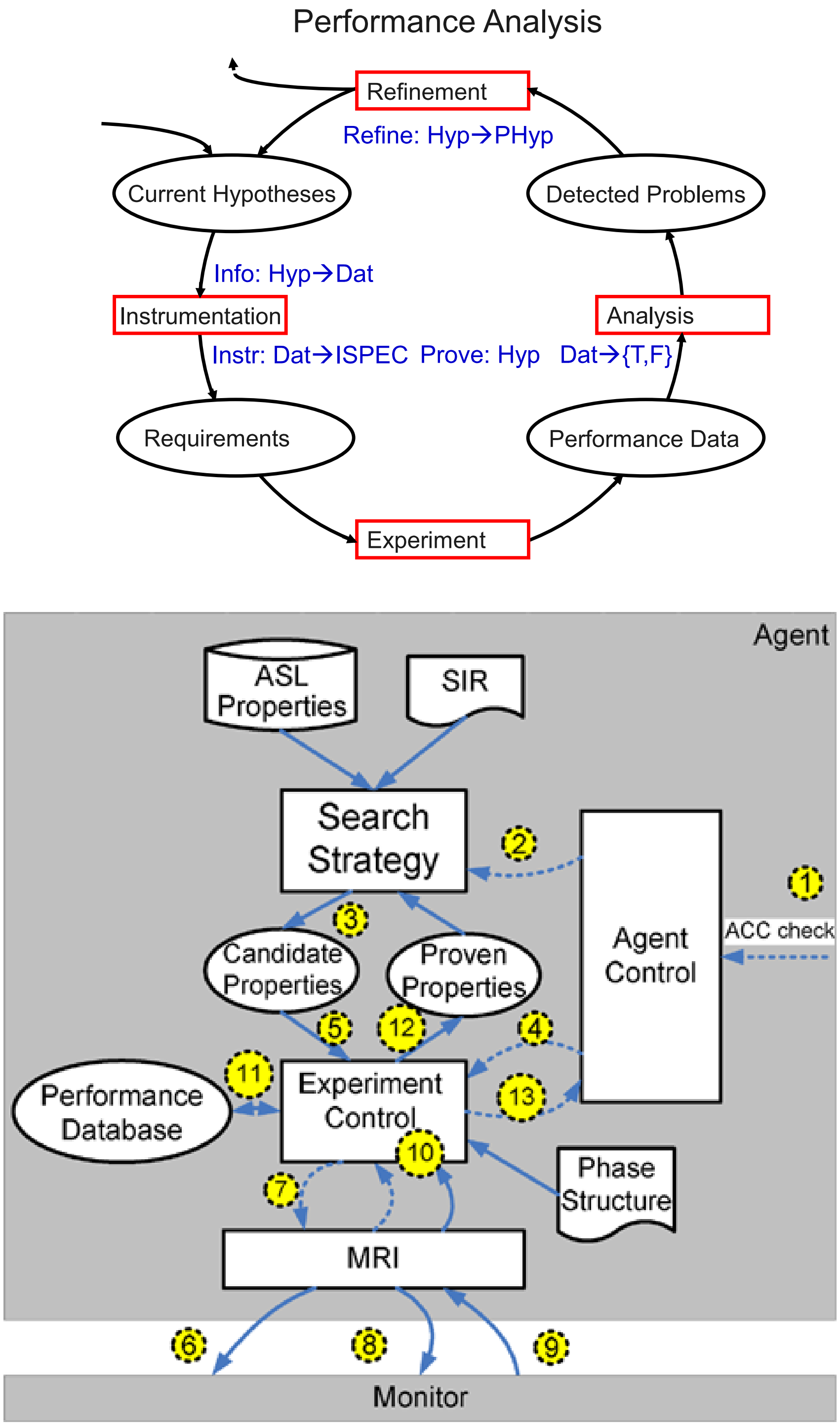


Figure 14. Architecture of the PERISCOPE system, from [159].

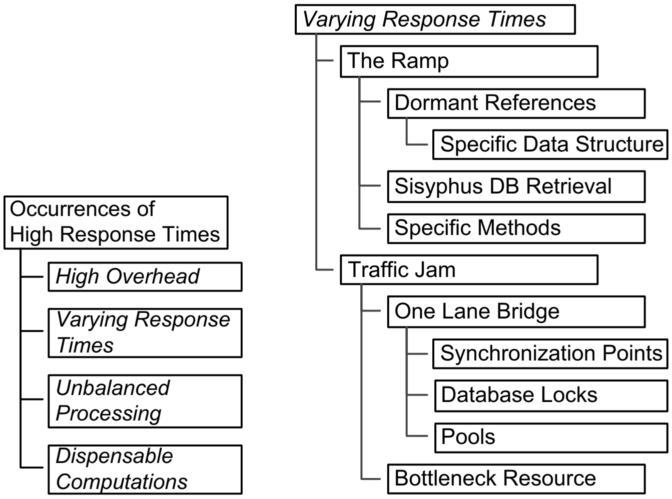
Scalasca [81, 80] is derived from KOJACK and addresses two problems: first, that creating a merged log is time consuming and can result in a file too large for some filesystems, and second, that serially scanning a merged log scales poorly as the number of processors in the traced application increases. In Scalasca, no merging is done; rather, each process writes its own local event log. The log is then processed in parallel, using the same number of processors as the application being analyzed. Rather than reducing all data to one node, the communication patterns of the original application are replayed, so that, for example, an `MPI_Send` in the original application becomes an `MPI_Send` in the replay with a payload indicating the parent events of the send.

Scalasca has been subsequently extended with new analyses. One such analysis, described by Böhme et al. [25], aims to automatically determine the causes of load imbalance in MPI applications. A wait state can be either *direct* if it is caused by a process blocking on communication from another process because that other process has not yet completed a computation, or it can be *indirect* if it is waiting on a receive because the other process is in turn waiting on a communication. The authors extend Scalasca with a *backwards* replay, allowing wait states to be attributed to other wait states or to delays in computation, thereby building a graph showing the root cause of the delays.

Automatically Fixing Performance Problems. Of particular interest are systems which not only automatically diagnose performance problems, but also can suggest solutions to the problem or even automatically modify source code. Cong et al. [49] describe a system with a structure similar to KOJACK, described above, but which is closely integrated with IBM compilers, taking as input reports on what optimizations were applied to blocks of input code, and able to provide optimization settings to the compiler in response to diagnosed problems, as well as transformation recipes to a polyhedral code optimization framework. Modeling or empirical testing are used to determine whether the proposed solution actually addresses the detected problem. Problems which cannot be addresses automatically result in suggestions to the user.

Recent versions of the PerfExpert system also implement automatic optimization [72], incorporating a central database which a set of modules access. Compilation modules encapsulate procedures for compiling and running input code. Measurement modules perform code instrumentation (which may entail cooperation with a compilation module), binary instrumentation, or monitoring through operating system facilities, and write measurements into the database. In this framework, measurements are distinct from metrics: a measurement is raw data collected during execution, while a metric has been further processed and rendered into a standard form. Analysis modules convert measurements into metrics, storing these into the database as well. Recommendation modules query the database, evaluating rules expressed as SQL queries. Each row returned by the query identifies a recommendation for an optimization and gives a ranking to that recommendation. The top-ranking recommendation is then applied using an optimization module, which first checks to verify that the recommendation actually applies and is valid given constraints inferred through static analysis of the input code. The recommendation, having been applied, results in new code which starts the process anew with a compilation module. This process continues until no more valid recommendations remain.

Wert et al. [227] perform automated performance diagnosis in the context of enterprise Java applications. In their system, a hierarchy of symptoms is specified, with each symptom in turn referring to a hierarchy of causes. An example of such a hierarchy is shown in Figure 15. For each symptom and cause, a detection strategy is provided, providing steps by which an automated experiment can be performed which will trigger the problem if the cause under consideration exists in the application being tested. The detection strategies specify a workload to apply to the application, measurements to be made, and a procedure for deciding whether the measurements support the hypothesized cause.



(a) Symptoms of known performance problems.

(b) Performance problems causing *Varying Response Times*.

Figure 15. Symptom and cause hierarchies as used by Wert et al., from [227]

Differential Analysis. Differential, or decremental, analysis is a technique for automated diagnosis of performance bottlenecks, with attribution to specific lines or operations in the original source code. First, binary analysis is performed using MAQAO [60], which produces a series of reports on degree of vectorization, utilization of execution units, and a series of performance estimates assuming that all memory requests are served from L1, that all memory requests are served from L2, that all memory requests are served from RAM, and finally a projection of performance for a fully-vectorized code. These reports are used to determine code regions for further analysis [126]. Selected loops are instrumented and run, with hardware performance counters related to the memory system being recorded. This generates hypotheses about the cause of performance bottlenecks. Finally, DECAN [127] performs differential analysis to determine the specific instructions causing the bottleneck. Given a binary executable, the instructions representing loops of interest are deleted or replaced with other instructions so as to suppress the effect of the instructions. This is done several times, yielding modified binaries in which certain classes of instructions are suppressed, such as one version suppressing load/store instructions and another suppressing floating-point instructions. These versions are then run with performance instrumentation, and the versions are compared to determine, for example, whether load/store (memory) or floating-point (compute) instructions are the performance bottleneck for the loop of interest.

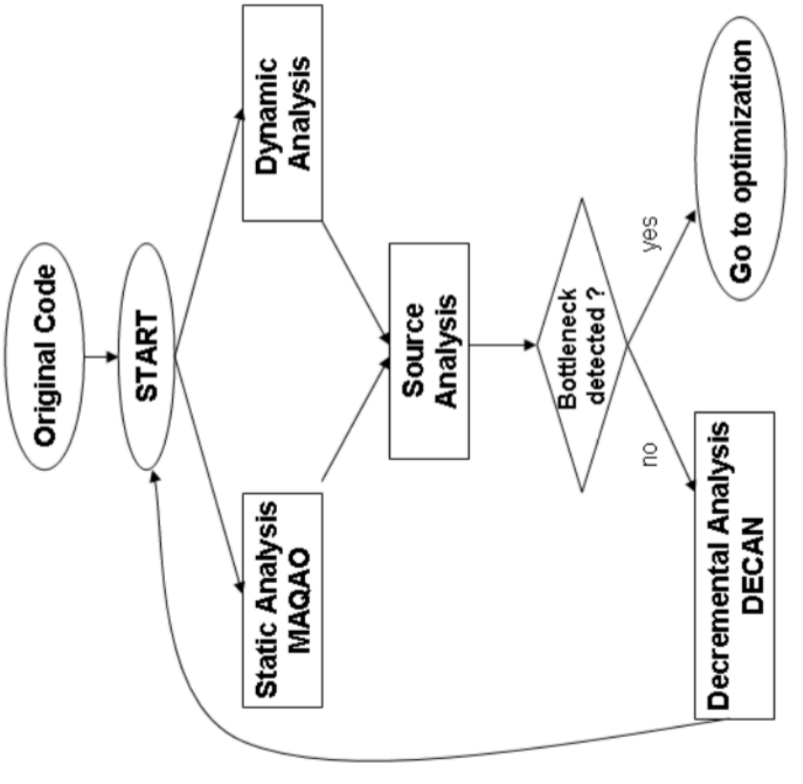


Figure 16. Methodology of DECAN, from [127]

2.6 Exascale Computing and Future Programming Models

All of the work described up to this point in the paper applies to existing supercomputers running existing codes written with traditional programming models such an MPI and OpenMP. The move to exascale, however, is likely to necessitate moves to other programming models [6]. An exascale system is one with peak performance of one exaflop ($10^1$8 floating point operations per second), about 30 times greater than the peak performance of Tianhe-2, currently the world's fastest supercomputer [216]. Yet in order for system deployment to be feasible, total power consumption of the system must be kept below about 20 megawatts. Tianhe-2 uses 17 megawatts, so to reach exascale we must increase performance by 30 times while holding power consumption basically constant. This will require adding substantially more concurrency at every level of the system: nodes must have more cores, cores must have more hardware threads, hardware threads must process SIMD instructions over more data at a time, all of which will result in the number of threads required to saturate the system growing from hundreds of thousands in current systems to tens to hundreds of billions in exascale systems. Providing enough work to generate these threads will require a different approach to programming [56].

Programming models that have been proposed for exascale systems tend to be *task based*. Rather than strictly dividing work across things like loop iterations, or partitioning work across nodes and running the same algorithms on every node on different parts of the data, task parallelism divides work into discrete chunks which carry dependency information. This dependency information can be expressed as a directed acyclic graph, allowing a runtime scheduler to proceed with executing a task as soon as its dependencies have been met. This allows task-parallel programs to spend less time idle compared to those using fork-join parallelism and communicating sequential processes, as shown in Figure 1. They also allow for easier adaptation to system variability by allowing work to migrate to address load imbalance caused by node variability; to do this, units of work are virtualized relative to hardware. Data is often also virtualized, so that data can be moved to work, or work can be moved to data, depending upon whichever is cheaper. Finally, by generating a very large number of tasks, latency can be hidden by swapping out a task waiting on a resource for another task [199].

In this section, I will review a number of task-based programming models. These differ by granularity (whether tasks are lightweight, at the level of loop iterations; medium-weight, at the level of functions; or heavy-weight, at the level of phases or steps in a workflow); by whether parallelism is explicit or implicit; by underlying source of parallelism (e.g, user-level threads, pthreads, systems built on top of MPI, etc.); by technology used by communication; by whether tasks may yield; by whether scheduling decisions are centralized or distributed; and by whether scheduling decisions are made statically or dynamically.

There are a number of node-local task based systems. While these could be combined with some other mechanism for inter-node parallelism, exascale systems will likely require that intra- and inter- node parallelism be expressed using the same model. Therefore I will not describe node-local systems in detail. These systems include OpenMP Tasks [12], Intel Threading Building Blocks [175], Qthreads [230], StarPU [11], Cilk Plus [182], and Concurrent Collections (CnC) [29].

Charm++. Charm++[1] is among the oldest adaptive asynchronous task-based runtimes, first released in 1992. Its central abstraction is the *chare*, a special C++ object encapsulating data and methods which can be invoked remotely by receipt of a message. Programs do not interact with the chare directly. Rather, creation of a chare yields a *proxy object* through which messages are sent, invoking *entry methods*, which are specially designated methods with signatures defined in a domain-specific language from which glue code is generated by a source-to-source compiler. Entry methods are required to run to completion; the scheduler will not interrupt them.

All messages are asynchronous: upon sending a message, the sender immediately continues executing. Any reply to a message is implemented as an additional message. A chare's global ID indicates a home node for the chare; however, chares are *migratable*: at any time a chare may be moved to another node, with the original home node forwarding any messages it receives and notifying senders of the new location of the chare, which is cached by senders for future use. Application developers are encouraged to *overdecompose* their applications by breaking them down into many more tasks than there are processing units on which the tasks will run. This helps with load balancing by keeping a pool of work available to assign to processing units as they become available. Migratability provides additional opportunities for load balancing by enabling the moving of work, along with its associated data, to underutilized nodes [121].

The Charm++ runtime has built-in facilities for runtime adaptation. The Charm++ Load Balancing framework, the architecture of which is shown in Figure 17, is one such facility [249]. A Load Balance Manager runs on each node. During execution, the Manager stores statistics on load and idleness into a database. When criteria for rebalancing are met, the Manager invokes one or more Load Balancing Strategies, which can query the database for information on the load of the local node and remote nodes. Strategy instances are themselves chares and can communicate with one another through message passing. Strategies inform the Load Balance Manager of how chares should be migrated, which occurs through interaction with the Array Managers.

Three types of load balancers are described in [249]: centralized, decentralized, and hybrid. The centralized load balancers send all performance data to one node, which processes all the data and distributes migration decisions. The simplest of these are the Random strategy, which randomly assigns chares to processing units. The Greedy strategy processes chares in order from longest-running to shortest-running, assigning tasks to processors ordered from least-loaded to most-loaded. The Refinement strategy swaps chares to adjust an existing distribution. More sophisticated load balancing strategies take communication into consideration, attempting to place groups of chares which communicate heavily together while still balancing load. These operate on the communication graph and include a Recursive Bisection strategy and a METIS [124] strategy. Variants of the above strategies are provided which consider that an application may be composed of several phases with different performance characteristics, which require gathering and using phase-specific load statistics.

As the size of the system increases, it becomes impractical to collect all the data needed for load balancing onto a single node. At the same time, making good load balancing decisions requires global information – we cannot decide to place work on the least-loaded node unless we know which node that is. Distributed strategies include neighborhood-based methods in which balancing occurs within a subset of nodes. This can be combined with a work-stealing approach, in which nodes in a neighborhood periodically send messages to one another informing them of their load, and idle nodes send messages to nodes which according to its view are overloaded, requesting that chares be migrated from the overloaded node to the idle node. These messages are prioritized for immediate processing, rather than being enqueued for later processing as with normal messages. The Hybrid strategy involves a tree of load balancing domains, with different strategies being used at different levels of the tree.

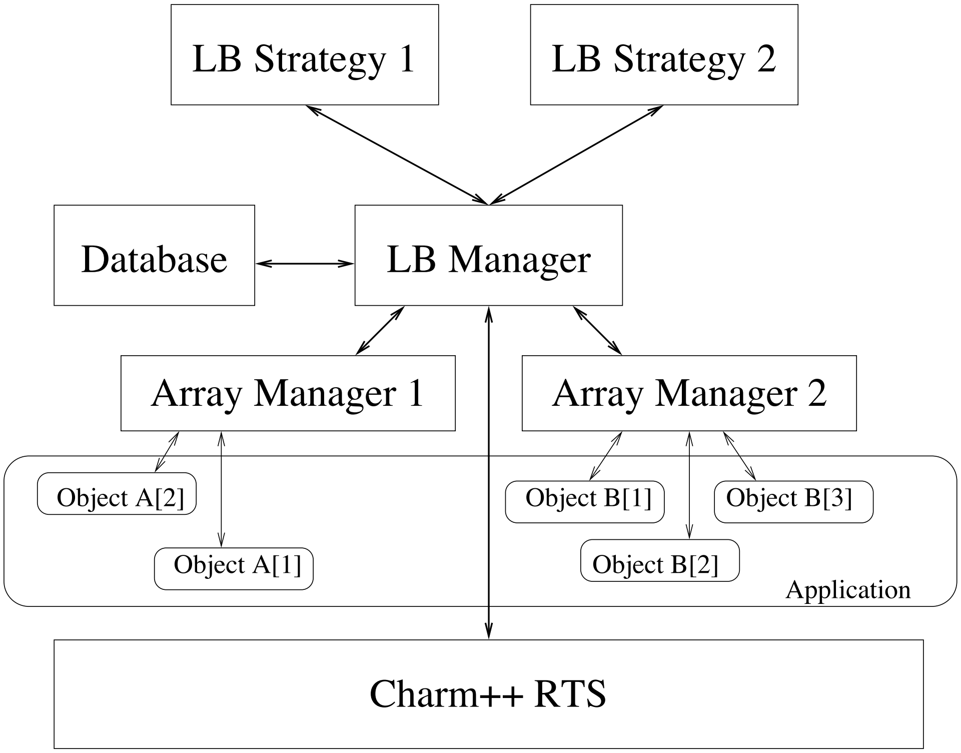


Figure 17. Architecture of the Charm++ Load Balancer (from [249])

An adaptive runtime system called PICS [203] (Performance-analysis-based Introspective Control System) has been implemented, which allows Charm++ applications to register *control points* [62]. Control points specify what effect application parameters have on various categories of performance-effecting properties, a library of which are provided by the system. Control points can be registered for effect types of Degree of Parallelism, Grain Size, Priority, Memory Consumption, Cache Miss Rate, Overhead, Number of Messages, and Message Size. Control points are registered explicitly by the application developer and are not automatically discovered; for example, the application can register that a variable controlling the size of a subproblem will change the grain size and degree of parallelism. Based on runtime performance measurement, the system selects a property to adjust and adjusts registered control points according to a strategy shown in Figure 18.

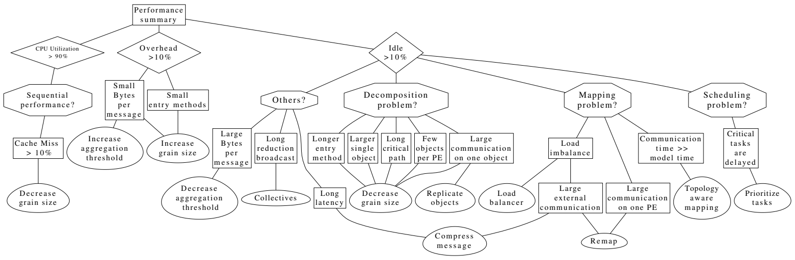


Figure 18. Decision procedure used by PICS to decide which control points to adjust (from [203])

A version of MPI, Adaptive MPI (AMPI), has been developed, which runs on top of the Charm++ runtime [100]. In AMPI, MPI processes are implemented as fully migratable Charm++ tasks, and MPI communications are implemented as Charm++ messages between tasks. The same load balancing strategies described above for native Charm++ programs can also be used for AMPI programs [101].

Swift. Swift [232][2] is a parallel scripting language designed for the specification of scientific workflows. Unlike general-purpose languages, Swift is not intended for performing mathematical operations but rather for sequencing and scheduling calls to external functions or entire executables written in other languages, such as C, C++, or Fortran. Swift is made aware of the types of inputs and outputs to such external computations, but they are otherwise treated as “black boxes” of which the Swift runtime has no knowledge.

A Swift program then consists of a series of parallel constructs, such as `foreach` loops, which contain external calls with specific inputs and outputs. Executions of a

---

[2] Unrelated to the language of the same name from Apple.

parallel construct implicitly specify tasks, so that, for example, two nested `foreach` loops each over 1,000 elements result in the construction of 1,000,000 tasks. Code such as

```
foreach i in [0:N-1] {
  foreach j in [0:N-1] {
    foreach k in [0:N-1] {
      foreach m in [0:N-1] {
        int r = f(i, j, k, m);
      }
    }
  }
}
```

creates N tasks which run independently, while

```
A[0][0] = 0;
foreach i in [1:N-1] {
  A[i][0] = 0;
  A[0][i] = 0;
}
foreach i in [1:N-1] {
  foreach j in [1:N-1] {
    A[i][j] = f(A[i-1][j-1], A[i-1][j], A[i][j-1]);
 }
}
```

creates N-1 initialization tasks which run independently and N-1 tasks, each of which depends on predecessor tasks.

A limitation of the original Swift is that scheduling occurs only on the node executing the driver script, limiting the scalability of scheduling. Swift/T [240] resolves this issue by running Swift on top of a new runtime, Turbine [239]. A small subset of the nodes in a job are reserved as *control engines*, which run *control fragments*, which in turn schedule *leaf tasks* (that is, user-defined external functions or executables) on the *workers*, which are those nodes not reserved as control engines. Workers and control engines communicate through a global address space called the *distributed* *future store* which manages write-once variables by which tasks return results and signal completion.

Static dataflow analysis is used to determine dependencies between tasks, which are made available to the scheduler, which does not schedule a task for execution until all of its inputs are available. As tasks never execute that point, tasks do not yield during execution, instead always running to completion before the scheduler may reuse the resources consumed by the task. Because the scheduler must monitor dependencies itself, scheduling overhead is higher than in dependency-unaware runtimes. Swift is then intended for medium-granularity tasks, with fine-grained parallelism expressed in the native language used to define leaf tasks. This is in contrast to lightweight tasking runtimes, which are intended to support multiple task granularities.

X10. X10 [45] is a PGAS language based on Java, to which it adds the concepts of *places* and *asynchronous activities*. Places contain data and activities run in a place, and both data objects and activities are *not* independently migratable, unlike in Charm++. However, places themselves *may* move: they do not directly correspond to a node or processor. When a place migrates, all data objects and activities in that place move with it. Activities (equivalent to *tasks* in other languages and runtimes) are launched with the code `async (p) S` where `p` is a place and `S` is a code block. An asynchronous activity invocation returns to the invoking process immediately. Waiting for a code block containing asynchronous activity invocations can be accomplished with `finish S`, where `S` is a code block. For example, in this simple implementation of Fibonacci,

```
static def fib(n:Int):Int {
 if(n < 2) return n;
 val f1:Int;
 val f2:Int;
 finish {
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1 + f2;
}
```

the statement `async f1 = fib(n-1)` launches a new activity which executes `fib(n-1)` (in the current place, since none is specified) and immediately continues to the next statement, `f2 = fib(n-2)`, which executes inside the current task. Since both statements are located inside a `finish` clause, once the second statement finishes the current task will wait for any subtasks launched within the block to complete before proceeding. Activities can be suspended during execution, unlike in Swift and Charm++.

As a PGAS language, X10 has support for arrays with elements resident in different places. Arrays are specified by *regions*, which specify the number of dimensions in the array and the extent of each dimension, and by *distributions*, which assign points in an array's region to a place. However, unlike traditional PGAS languages such as UPC, in which any node can access any address in the global address space, X10 restricts access to mutable (non-`final`) data to only the place in which it resides. For one place to access data stored in another place, the first place must launch an activity in the second place. If we have two arrays `A` and `B` such that `A[i]` and `B[j]` are located in different places, and we want to carry out the assignment `A[i] = B[j]`, we must launch multiple activities: one in the place where `B[j]` resides, to read its value, and one in the place where `A[i]` resides, to assign the value read from `B[j]`, as in this example:

```
finish async (B.distribution[j]) {
  final int bb = B[j];
  async (A.distribution[i]) {
    A[i] = bb;
    }
  }
```

Here, the inner activity is able to read the value stored in `bb` because it is declared `final`, and non-mutable values can be read from any place.

Dependencies are managed through futures or through an abstraction called *clocks*, a version of a barrier in which an activity can be registered on an arbitrary number of clocks and can simultaneously advance all clocks on which it is registered, which can be used to implement producer-consumer activities. The `next` statement causes the current activity to suspend until all clocks on which it depends have been advanced by calling `advance` on the clocks.

Chapel. Chapel [43] is a PGAS language providing abstractions which are very similar to those in X10, as described in Section 2.6. The statement `begin S` causes the current task to launch a new task which executes the code block `S`, while the current task immediately continues executing; this is equivalent to the `async` statement in X10. The statement `sync S` executes the statements in the code block S, then blocks until all subtasks created within `S` have completed; this is equivalent to `finish` in X10. As in X10, arrays can have arbitrary indices and customizable assignments of points to locales through user-definable *domain maps*, or *dmaps*. The primary difference between Chapel and X10 is that Chapel supports access to shared objects from any locale, as in traditional PGAS languages, while X10 restricts access to the place in which an object resides. Chapel also supports additional constructs for task creation, such as `cobegin S`, which launches a separate task for every statement in `S`, and `coforall E in C do S`, which launches a task executing the statements in `S` for every element `E` in the iterable collection `C`.

UPC++. UPC++ [250] is a C++ library which implements PGAS functionality as found in UPC along with asynchronous task support, which is not a feature of UPC. Rather than extend C++ with new keywords and types, as UPC did with C, UPC++ adds PGAS support purely as a library through the use of C++ templates. The UPC `shared` keyword applied to value types becomes the template `shared_var<underlying_type>`, while `shared` pointers become the template `global_ptr<underlying_type>`. Using a `shared_var` in a context in which the underlying type is expected is transparently converted to a local or remote memory access as needed by an implicit conversion operator. Dereferencing a `global_ptr` is also transparently converted to a local or remote memory access. A local `global_ptr` can also be cast to a plain pointer to reduce overhead when it is known not to be remote. Direct support is available for allocating memory from one node which is resident in the memory of another node, a feature not found in UPC. Multidimensional arrays are supported similarly to X10 and Chapel.

Asynchronous tasks can be launched using `future<T> f = async(place)(function, args)`, where `function` is a callable object returning `T`. The returned `future` can be used to retrieve the value computed by `function` by calling `f.get()`, which blocks until the task has completed. UPC also provides a `finish` construct analogous to the one in X10, and an event-based system for building a dependency DAG, in which an `async` optionally takes `event` objects to signal completion and to hold execution of a task until a set of events have been signaled. Unlike in Charm++ and Swift, tasks are non-migratable. Tasks are intended to be launched only on remote nodes. Habanero-UPC++ [129] allows both local and remote task invocation and extends the runtime with additional work-stealing support.

Open Community Runtime. The Open Community Runtime [152] is an asynchronous task-based runtime. Unlike the other systems described thus far, OCR provides a runtime *only*; it is not accompanied by a user-facing language or library, and is intended as a target for third-party languages and libraries. OCR is based on three abstractions: *Event-Driven Tasks*, or EDTs, asynchronous tasks which, once started, are required to run to completion; *Data Blocks*, which represent globally-accessible data, and *Events*, which connect EDTs, Data Blocks, and other events together. EDTs have *input slots* and *output slots* which may connect directly to Data Blocks or to Events. An example DAG is shown in Figure 19 for a Fibonacci program.

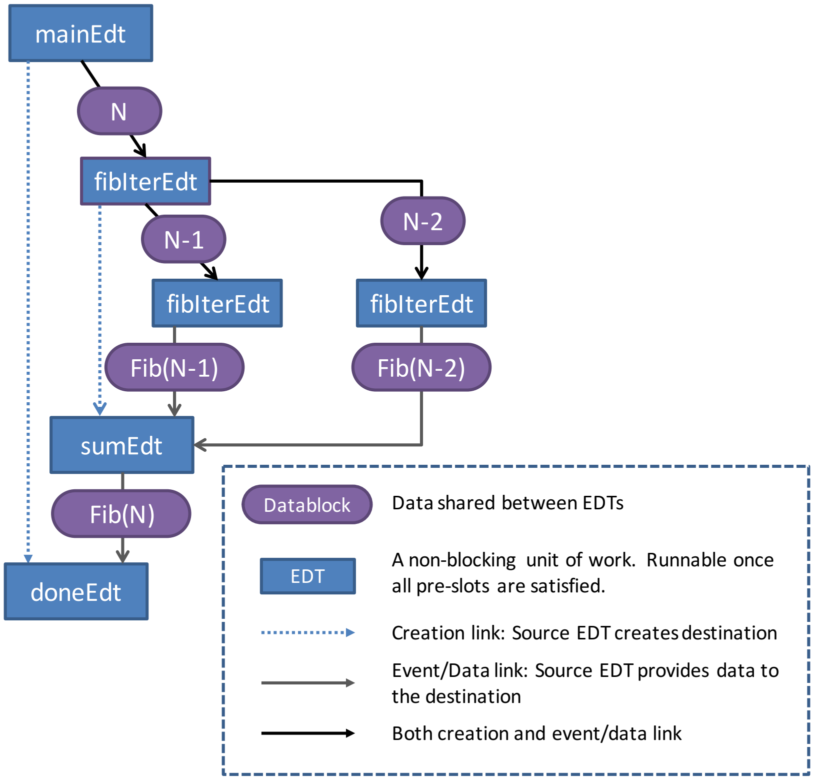

mainEdt
N
fibIterEdt
N-1
N-2
fibIterEdt
fibIterEdt
Fib(N-1)
Fib(N-2)
sumEdt
Fib(N)
doneEdt
Datablock
Data shared between EDTs
EDT
A non-blocking unit of work. Runnable once all pre-slots are satisfied.
Creation link: Source EDT creates destination
Event/Data link: Source EDT provides data to the destination
Both creation and event/data link

Figure 19. DAG for an OCR Fibonacci code (from [209]). Blue rectangles are EDTs, purple rounded rectangles are Data Blocks, and arrows are Events.

With Data Blocks, OCR makes data an explicit part of the dependency graph, unlike most other systems. Events linking Data Blocks to other objects carry information on how they are to be accessed, allowing the runtime additional optimization opportunities: by default, a Data Block is in *read-write mode*, so that the runtime can make no assumptions about which EDTs will access the Block. Also available are *exclusive write*, in which only one EDT may write to the block at a time; *read only*, in which the Data Block provided by the event may not be written to by the target EDT, and *constant*, in which *no* EDT may write to the Block.

Legion. Legion [21] is a task-based runtime with a unique data abstraction called *Logical Regions*. As with OCR's Data Blocks, Logical Regions represent data in a global address space and associates with it access restrictions, namely *privileges* (read-only, read/write, *etc.*) and *coherence* (exclusive access, atomic access, *etc.*). As with arrays in X10 and Chapel, the assignment of ownership of array elements is separate from declaration of the array extent. However, unlike in OCR, X10, or Chapel, Legion's Logical Regions do not impose *any* physical data layout, deferring this decision until a task using the region is to be executed.

A Logical Region encodes what types of data are to be stored, but says nothing about the physical representation of the data. Regions are then *partitioned* into *subregions*, with partition operations being annotated as either *disjoint* (that is, no two subregions of the region share data) or *aliased* (subregions may overlap). At runtime, a *mapper* function determines the distribution of data to nodes and also the physical layout of subregions on a node. Legion provides a default mapper with functionality similar to distributions in X10 or domain maps in Chapel. Custom mappers can be provided which take into account architecture-specific properties (such as choosing structure-of-arrays vs array-of-structures depending on whether a CPU or GPU is targeted) as well as application-specific properties (such as a graph partitioner tuned to the properties of graphs used in an application). Different tasks can use different mappers for the same regions, in which case the runtime will dynamically reformat the physical representation.

Grappa. Grappa [164] is a task-based runtime and C++ library with generally similar features to UPC++, providing a tasking model with a partitioned global address space. As with X10, only the owner of a memory address is allowed to directly access it, with remote access being performed through remote task invocation. In most PGAS systems, such as UPC, UPC++, X10 and Chapel, memory partitions are associated with *nodes*, so that if thread A and thread B are located on the same node, and thread A accesses shared memory located in thread B, the access happens *directly* and does not go through the remote memory subsystem. Grappa does not partition memory in this way: ownership is associated with a *core*, not with a node. If worker A and worker B are running on two cores of the same node, and worker A runs a task which accesses memory owned by worker B's core, then a task must be scheduled on worker B to perform that access and return the result to the task on worker A. Tasks whose only purpose is to access remote memory are called *delegate* tasks and are not allowed to context switch or block. Full-fledged tasks may block, in which case they will be suspended and another task scheduled in their place.

The high-granularity memory partitioning used in Grappa enables an approach to global data structures with low contention, known as *flat combining* [99]. Instead of acquiring a lock to access the shared data structure, per-core lists of pending requests are maintained. When a worker attempts to access a non-local part of a global data structure, it adds the request to the list associated with the core owning the memory to be modified and then blocks, causing another task to be scheduled in its place. Periodically, combining workers are scheduled on each core, which process requests in the order in which they were received.

HPX. HPX [119] is an asynchronous task-based runtime and C++ library based on the ParalleX model [117]. The distinguishing feature of HPX is its adherence in design to C++ standards. C++11 [113] added node-local tasks to the C++ standard library in the form of `std::async` to launch a task, which returns an object of type `std::future` which can be used for synchronization and to retrieve the value returned by the task. HPX makes this same model available for distributed systems, so that an existing C++11 application making use of `std::async` and `std::future` for parallelism can be converted to an HPX application by simply replacing them with `hpx::async` and `hpx::future`. Remote invocation of a task is accomplished by passing an argument to `hpx::async` indicating on which *locality* the task should run. Sending data and work is accomplished by means of a *parcel* abstraction. Notably, HPX provides for transparent task migration, meaning that tasks can migrate without stopping other computations which are occurring on the node. During migration, any incoming messages intended for the tasks or data being migrated will be stored for automatic forwarding once migration is complete. The architecture of HPX is shown in Figure 20.

HPX has recently been extended with a new mechanism for implicitly creating tasks, known as *executors* [118]. With executors, parallel implementations of Standard Template Library algorithms can allow decisions as to how to distribute work to be deferred to external libraries such as HPX. Algorithms which support executors take an executor object as the first argument, which in turn receives lambda functions from which it creates tasks. The executor is free to determine how much work to assign to a given task, and how to distribute tasks in a multi-node setting.

Spark. Spark [248] is based on a generalization of the Map-Reduce model [58] found in systems such as Hadoop to problems expressed as general data

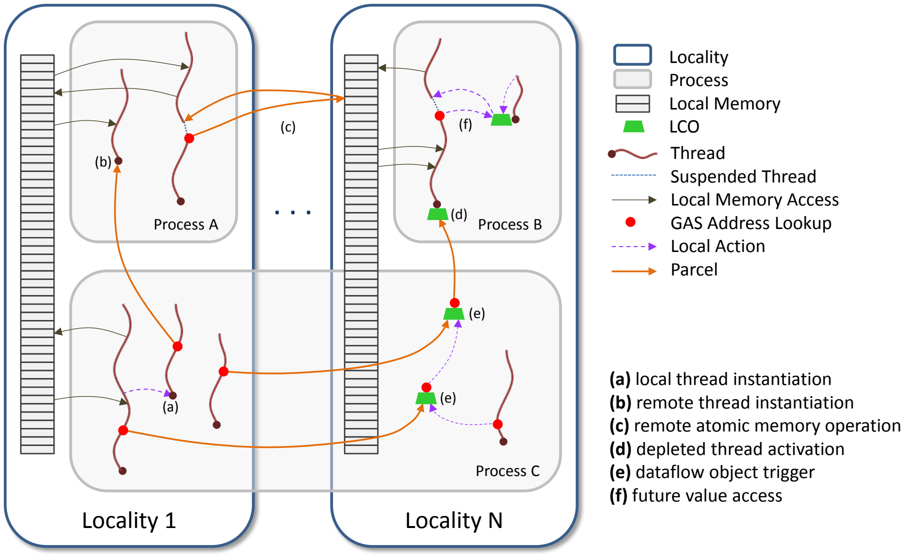

Process A
Process B
Process C
(a)
(b)
(c)
(d)
(e)
(e)
(f)
. . .
Locality 1
Locality N
Locality
Process
Local Memory
LCO
Thread
Suspended Thread
Local Memory Access
GAS Address Lookup
Local Action
Parcel
(a) local thread instantiation
(b) remote thread instantiation
(c) remote atomic memory operation
(d) depleted thread activation
(e) dataflow object trigger
(f) future value access

Figure 20. Computational model of HPX.

flow graphs, relaxing the restriction that the graphs be acyclic. Operations are carried out on *resilient distributed datasets* [247], or RDDs, which store data across nodes and which carry sufficient information to recompute their contents. Programs are expressed in terms of RDDs derived from transformations (of which map is only one) applied to other RDDs and actions (of which reduce is only one). The application developer can choose to request that certain RDDs be cached in memory or saved to disk. The developer therefore has to make decisions based on tradeoffs between the costs of storage (in memory and time) and recomputation (in time). RDDs are lazily evaluated, which can create challenges in attributing performance to particular lines or regions of code, as they do not execute until they are needed.

RDDs are composed of *blocks*, which represent data. Data storage is also managed by the runtime: while the runtime will attempt to keep data in memory, it is also free to evict data from memory, dropping it to disk instead, or to drop it entirely, requiring that it be recomputed if needed again in the future.

2.7 Conclusion

For programs written for current-generation supercomputers and using programming models such as MPI and OpenMP, a wide variety of performance analysis tools are available for collecting profiles and traces, for analyzing and visualizing profiles and traces, for offline tuning and online adaptation using automatic performance tuning, for automatic diagnosis of performance problems, and for construction of models from performance data. The move to exascale, however, will require such a large number of threads that programming using MPI and OpenMP will become difficult, and runtimes being investigated for exascale use a different structure for specifying programs: directed acyclic graphs of light-weight or medium-weight tasks for both intra- and inter-node parallelism. Existing techniques for collecting and making use of performance data are not suitable for analysis of systems of billions of light-weight tasks, so new techniques will need to be developed to go along with new programming models, runtimes, and languages at exascale. It will not be feasible, for example, to collect a trace of the start and stop times of many billions of tasks.

Many-tasks systems have many additional layers of abstraction over systems like MPI, and this can cause us to lose the connection between a source line and why it is executing, or why it is not executing. In MPI, we can observe that we are waiting on a receive and work backwards to a cause, such as a late sender. In a DAG based system, the cause can be far removed from its effect, or can depend instead on scheduling policy: Why has task A not executed? Because it is waiting on data from task B. Why has task B not executed? It is eligible to; the scheduler has simply not scheduled it yet, as there are many tasks eligible for scheduling. What schedule yields the best throughput? Why is this task executing instead of some other task? Why is this worker idle now? How are hardware resources shared between worker threads? How can hardware counter values be attributed to tasks when there are multiple tasks and tasks can suspend and resume?

Because of the huge number of tasks in a system, we will need to answer these questions without using post-mortem analysis, as this would require saving too large a volume of data to disk, yet most existing studies of performance in task-based systems have used post-mortem analysis of short runs or on a small number of nodes [88, 44]. Performance monitoring at exascale will require *in-situ performance analysis* [131] and *online adaptation* [85]. This will require both node-local performance data and decision making as well as a *global view* [104] on performance through which nodes can become aware of the state of other nodes so that they can best make local decisions, as centralized control will likely be infeasible at exascale.

No in-situ system providing online adaptation for a task-based runtime through a global view currently exists. The adaptive load balancing system used in Charm++, described in Section 2.6 is close, but is limited to controlling migration and does not affect other system parameters, while Charm++'s PICS system operates on a per-node basis. Node-local adaptation based on contention for memory controller resources has been demonstrated for OpenMP tasks [5] and HPX [148]. A prototype in-situ performance monitoring tool providing a global view, GTI-OTFX [224], has been developed, but only supports traditional MPI applications.

Chapters 4, 6, and 7 of this dissertation describe tool-runtime integrations using APEX [102]. APEX is built around the concept of a *policy*, which can be registered to respond to events of interest produced by an instrumented runtime. While policies ultimately run on a single node, they run as tasks within the task-based runtime and have access to the same communications infrastructure as any other task; thus, in HPX, they can communicate with one another using one-sided puts and gets in the global address space. Built-in support in HPX for efficient reductions can be used to aggregate performance data. We envision ultimately having a system in which a small portion of localities are reserved for performance analysis and adaptation, running analysis tasks which receive data from lighter-weight tasks which collect and forward performance data from compute localities.

CHAPTER III

ONLINE COMMUNICATIONS ADAPTATION IN UPC

This chapter includes co-authored material previously published in the Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2015) [38] and the International Journal of High Performance Computing Applications (IJHPCA) [40]. Those papers were collaborations with Khaled Ibrahim, Sam Williams, and Costin Iancu. I wrote the threaded version of THOR and ran and analyzed all experiments. Costin Iancu wrote the original process-based version of THOR. Sam Williams wrote the MiniGMG benchmark. Khaled Ibrahim wrote the multi-domain support in UPC's Gemini and Aries network drivers.

3.1 Introduction

UPC is a distributed SPMD language and runtime based on an extension of the C memory model to encompass both local and remote memories. UPC exposes to the application developer a *partitioned global address space*, wherein a pointer may point to local memory (and be equivalent to an ordinary C pointer) or may point to remote memory (in which case dereferencing is transparently converted into network operations).

While traditional performance monitoring tools will provide *correct* results for UPC applications, there are limitations to providing insightful and actionable results, primarily due to the asynchronous nature of communications in UPC. In the *relaxed* consistency mode, the runtime is free to reorder communications within a block, and when one-sided communications are used, message injection occurs on a separate thread from application code, and it is difficult to correlate the time spent in a communication with the application code that triggered the communication without runtime integration. In terms of actionability, this chapter describes a tool, THOR, or Throughput-Oriented Runtime, which integrates with the UPC runtime and receives and processes outgoing communications requests. Based on the sizes and patterns of communication, the tool modifies the runtime policy for consolidating or splitting communications, achieving speedups of up to 55% on a UPC+CUDA geometric multigrid benchmark.

3.2 Maximizing Message Concurrency

Attaining good throughput on contemporary high performance networks requires maximizing message concurrency. As long as flat Single Program Multiple Data (SPMD) parallelism with one task per core has been dominant, this has not been a problem in application settings. Developers first employ non-blocking communication primitives and have multiple outstanding messages overlapped with other communication or computation inside *one* task. By using as many SPMD tasks as available cores, traffic is further parallelized over multiple injecting tasks within the node.

The advent of heterogeneous systems or wide homogeneous multicore nodes has introduced the additional challenge of tuning applications for intra-node concurrency, as well as communication concurrency. Manually tuning or transforming applications to provide the optimal message parallelism is difficult: 1) the right strategy is system dependent; 2) the right strategy is programming model dependent; and 3) parallelizing message streams may be complicated in large code bases. Furthermore, due to the implementation limitations described throughout this chapter, to our knowledge optimizations to parallelize communication within a task have not been thoroughly explored.

We present the design of a runtime that is able to increase the instantaneous network concurrency and provide saturation independent of the application configuration and dynamic behavior. Our runtime alleviates the need for *spatial* and *temporal* application level message concurrency tuning. This is achieved by providing a "multi-threaded" runtime implementation, where dedicated communication server tasks are instantiated at program start-up along the application level tasks. We increase concurrency by offloading communication requests from the application level to the multiple communication servers. The design allows for offloading communication to dedicated cores, as well as a cooperative scheduling approach where servers share cores with application level tasks. The communication servers provide performance portability by using system specific performance models. This work makes the following contributions:

- We provide a detailed analysis of the optimization principles required for multi-threaded message injection. Our experiments on InfiniBand, Cray Aries and Gemini networks indicate that saturation occurs differently based on the network type, the message distribution and the number of cores active simultaneously.

- We describe a runtime capable of maximizing communication concurrency *transparently*, without involving the application developer. The main insight is that after deciding the cores allowed to perform communication, the runtime can maximize the interconnect performance by using message size to guide the assignment of message processing to communicating cores.

- We quantify the performance benefits of parallelizing communication in hybrid codes and identify the shortcomings of existing runtime implementation practices.

Our implementation extends the Berkeley UPC [220, 31] runtime and therefore we demonstrate results for Partitioned Global Address Space (PGAS) applications and one-sided communication primitives. Experimental results indicate that our approach can improve message throughput and bandwidth by as much as 150% for 4KB messages on InfiniBand, by as much as 120% for 4KB messages on Cray Aries and by as much as 54% for 2KB messages on Cray Gemini. Our runtime is able to transparently improve end-to-end performance for all-to-all collectives where we observe as much as 30% speedup. In application settings we observe 23% speedup on 12,288 cores for a NAS FT benchmark implemented in UPC+`pthreads` using FFTW [74]. We observe as much as 76% speedup on 1,500 cores for an already heavily optimized UPC+OpenMP geometric multigrid [236] application using point-to-point communication. For the geometric multigrid GPU implementation in UPC+CUDA, we observe as much as 44% speedup on 512 cores/GPUs.

We demonstrate performance benefits for hybrid programming using a PGAS programming language by exploiting shared memory within the node and one-sided communication. These characteristics are present in other models such as MPI 3 one-sided, as well as implementations of dynamic tasking languages such as X10 [46] or Habanero-C [95, 47]. Besides implicit parallelization, the principles presented apply to cases where communication is explicitly parallelized by the developer using OpenMP or other shared memory programming models. Although this may seem sufficient, to achieve *performance portability* our results argue for the transparent scheduling of these operations inside the runtime.

The rest of this chapter is structured as follows. In Sections 3.3 and 3.4 we discuss the design principles of multi-threaded message injection. In Section 3.5 we discuss network performance emphasizing the relationship between message concurrency and bandwidth saturation. In Section 3.6 we discuss the integration of message parallelization into existing application settings. In particular we quantify the need for dynamic message parallelization and the impact of current core allocation mechanisms on performance. In Section 3.7 we summarize our results, while in Section 3.8 we present related work. We conclude the chapter in Section 3.9.

3.3 Communication and Concurrency

In order to actively manipulate message concurrency, program transformations must address both *spatial* and *temporal* aspects.

Spatial concurrency is controlled by choosing the number of active tasks (or cores) that perform communication operations, e.g. MPI ranks. By selecting a particular programming model, developers effectively choose the amount of spatial concurrency exploited within the application.

Temporal concurrency captures the insight that not all the tasks may want to communicate at the same time and the network may be perennially under-utilized even when a large number of messages are logically available inside a task. Messages within a task are "serialized", even for non-blocking communication: 1) message injection is serialized inside the issuing task; and 2) parts of the message transmission may be serialized by the network hardware for any task. In particular, for load imbalanced or irregular applications, only few tasks may communicate at any given time and the message stream within any task should be further parallelized.

SPMD programs provide spatial concurrency by running one task per core. For well balanced applications, there usually exists communication concurrency, even enough to cause congestion [138]. In this case throttling the spatial concurrency of communication improves performance. To our knowledge temporal concerns have not been explored for load imbalanced SPMD codes.

Hybrid parallelism [34, 153] combines SPMD with another intra-node programming model such as OpenMP. Currently, communication is issued only from the SPMD regions of the code. When compared to pure SPMD, these new hybrid codes run with fewer "communication" tasks per node and consequently exhibit lower spatial concurrency. For example, hybrid MPI+CUDA codes [153, 141, 142] tend to use one MPI rank per GPU for programmability and performance reasons. Hybrid MPI+OpenMP codes tend to use one MPI rank per NUMA domain for locality reasons. Previous work [234] showed that tuning the balance between the number of MPI ranks and OpenMP threads was essential in attaining best performance. Although that work suggested thread-heavy configurations were ideal for those machines (minimize data movement when a single thread can attain high MPI bandwidth), current machines (low MPI bandwidth per thread) can make a more nuanced trade between total inter-process data movement and total MPI bandwidth.

To our knowledge, techniques to further parallelize communication have not yet been shown beneficial in applications. As parallelizing the communication at the application level using OpenMP should be tractable, the main reason is the inability of current runtime implementations to provide good performance when mixing processes with `pthreads`. Communicating from OpenMP within one MPI rank requires running in `MPI_THREAD_MULTIPLE` mode, which has been reported [210] to negatively affect performance.

Applications written in programming models that support asynchronous task parallelism [47, 46] should offer the programmer high message concurrency, as every message can be performed inside an independent activity. However, this is mostly an illusion as communication is usually serialized inside the runtimes due to implementation constraints. For example, HCMPI [47] combines the Habanero-C [95] dynamic tasking parallel programming model with the widely used MPI message-passing interface. Inside the runtime there are computation and communication workers implemented as `pthreads`. To work around multi-threaded MPI's limitations, computation workers are associated with only one communication worker that uses MPI_THREAD_SINGLE. Thus, communication is de facto serialized inside a HCMPI program. X10 [46] implementations running PAMI on IBM BlueGene/Q can provide high message concurrency, but most likely serialize communication on non-IBM hardware.

In this work we argue for transparent parallelization of one-sided communication using a "multi-threaded" runtime implementation. We provide a decoupled parallel communication subsystem that handles message injection and scheduling on behalf of the application level "tasks". As this is designed to maximize network utilization, application level programmers need only to use non-blocking communication primitives without worrying about scheduling optimizations. While we show results for hybrid UPC+OpenMP and UPC+ `pthreads` programming, these principles are applicable to other one-sided communication runtimes such as MPI-3 and map naturally into programming models using dynamic task parallelism such as Habanero-C. Accelerator based programming such as MPI+CUDA is another clear beneficiary of our approach.

From the above discussion, it is apparent that maximizing communication parallelism in programming models beyond SPMD faces several challenges. There is an engineering hurdle introduced by the requirement to mix processes with `pthreads` inside the runtime implementation. As *performance* is poor in most `pthreads` implementations[1], we explore a dual parallelization strategy using either processes or `pthreads` as communication servers. This approach is likely to be required for portability in the medium term future, as fixing `pthreads` on a per runtime basis is non-trivial.

Transparent optimizations are good for programmer productivity, but one may argue that explicitly parallelizing communication using OpenMP is enough. Explicit manual communication parallelization faces *performance portability* challenges. First, performance is system dependent and it also depends on the instantaneous behavior of the application, i.e. how many tasks are actively communicating. Second, it is challenging to parallelize communication in an application already modified to overlap communication with other parallel computation.

3.4 Runtime Design

Contemporary networks offer hardware support for one-sided Remote Direct Memory Access (RDMA) Put and Get primitives. Runtime implementations are heavily optimized to use RDMA and applications are optimized to overlap communication with other work by using non-blocking communication primitives of the form `{init_put(); ... sync();}`.

We target directly the UPC language [218], which provides a Partitioned Global Address Space abstraction for SPMD programming, where parts of the program heap are directly addressable using one-sided communication by any task. Our implementation is designed to improve performance of codes using the new UPC 1.3 non-blocking communication primitives, e.g. `upc_memput_nb()`, `upc_waitsync()`. We modify the Berkeley UPC implementation [31], which runs

---

[1] Exceptions are PAMI on IBM BG/Q and GASNet on Cray Aries.

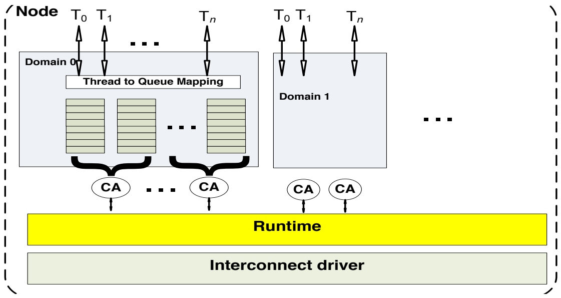


Figure 21. Runtime architecture of THOR.

on top of GASNet [26]. GASNet provides a performance portable implementation of one-sided communication primitives.

Our idea is very simple: we achieve transparent network saturation by using a dedicated communication subsystem that spawns dedicated communication tasks, herein referred to as servers. Any communication operation within an application level task is forwarded to a server. Thus, we increase the parallelism of message injection by controlling the number of servers and we can control serialization deeper inside the network hardware by tuning the policy of message dispatch to servers.

The basic runtime abstraction is a communication domain. As shown in Figure 21, each communication domain has associated with it a number of clients (T_i) and a number of servers (CA). Any client can interact with any server within the same communication domain. In practice, communication domains are abstractions that can be instantiated to reflect the hardware hierarchy, such as NUMA domains or sockets. Clients are the application level tasks (threads in UPC parlance).

Servers are tasks that are spawned and initialized at program startup time. They provide message queues where clients can deposit communication requests. A communication request is a Put or Get operation and its arguments `(src, dest, size)`. While active, the server tasks scan the message queues, initiate and retire any requests encountered. In order to avoid network contention, servers can choose to initiate messages subject to flow control constraints, e.g. limit the number of messages in flight. To minimize interference with other tasks, servers are blocked on semaphores while message queues are empty.

To implement the client-server interaction we transparently redirect the UPC language-level communication APIs, *e.g.* `upc_memput_nb()` or `upc_waitsync()`, to our runtime and redefine the `upc_handle_t` datatype used for message completion checks. For any communication operation at the application level, our runtime chooses either to issue the message directly or to deposit a descriptor in one of the server queues. Both the order of choosing the next message queue and the number of messages deposited consecutively in the same queue are tunable parameters.

Any non-blocking communication operation returns a `handle` object, used later to check for completion. The client-server interaction occurs through messages queues, which are lock free data structures synchronized using atomic operations. In our implementation, application level communication calls return a value (`handle`) which represents an index into the message queues. Currently, we do not dynamically manage the message queue entries and clients have to explicitly check for message completion before an entry is reclaimed. This translates into a constraint at the application level that there is a static threshold for the number of calls made before having to check for message completion.

The UPC language allows for relaxed memory consistency and full reordering of communication operations. This is the mode used in practice by applications and our servers do not yet attempt to maintain message ordering. Strict memory consistency imposes order on the messages issues within a UPC thread. In our implementation this is enforced inside the initiator task.

Implementation Details. Achieving performance requires avoiding memory copies and maximizing the use of RDMA transfers, which at the implementation level translates into: 1) having shared memory between tasks; 2) having memory registered and pinned in all tasks; and 3) having tasks able to initiate communication on behalf of other tasks. We provide a dual implementation where servers are instantiated as either processes or `pthreads`. The underlying UPC runtime implementation provides shared memory between tasks in either instantiation.

Previous work [24] indicates that best RDMA communication performance in UPC is attained by process-based runtime implementations, i.e. the applications run with one process per core. As this is still valid[2] for most other runtimes on most existing hardware, our first prototype spawned servers as stand-alone processes inside the runtime. This required non-trivial changes to the BUPC runtime. However, as discussed later, idiosyncrasies of existing system software determined us to provide a `pthreads`-based implementation for scalability reasons. The use of shared memory within multicore node, for instance using OpenMP, allows less replicated state [17] and reduces the memory usage of runtime, which is critical at scale. While our process-based implementation requires modified UPC runtime, the `pthreads` is written using unmodified UPC runtime and can be distributed as a stand-alone portable library. Furthermore, the latter implementation can take advantage off the good `pthreads` performance of GASNet [108] on Cray GNI messaging library (supported on Gemini and Aries interconnects).

Startup: RDMA requires memory to be pinned and registered with the network by any task involved in the operation. `pthreads` inherit registration information from

---

[2] Except PAMI on IBM BG/Q, GASNet on Aries and Gemini.

their parent processes, thus servers as `pthreads` can be spawned at any time during execution, including user level libraries.

Getting both shared memory and registration working together with servers as processes required complex modifications to the Berkeley UPC runtime code. The BUPC startup code initializes first the GASNet communication layer and then proceeds to initialize the shared heap and the UPC language specific data structures. As RDMA requires memory registration with the NIC, having the communication servers as full fledged processes requires them to be spawned at job startup in order to participate in registration.

Tasks spawned by the job spawner are captured directly by GASNet. Inside the UPC runtime there exists an implicit assumption that any task managed by GASNet will become a full-fledged UPC language thread. Furthermore, there is little control over task placement and naming as enforced by the system job spawner. We had to modify the UPC startup sequence to intercept and rename all server tasks before the UPC specific initialization begins. This cascaded into many other unexpected changes imposed by the BUPC software architecture. Internally, we split the UPC and server tasks into separate GASNet teams (aka MPI communicators) and reimplement most of the UPC runtime APIs to operate using the new task naming schema. In particular, all UPC pointer arithmetic operations, communication primitives, collective operations and memory allocation required modifications.

RDMA and Memory: The new UPC 1.3 language specification provides the `upc_castable` primitive to allow passing of addresses between tasks within a node. `pthreads`-based implementation can perform RDMA on these addresses, albeit with performance loss. Shared addresses are not guaranteed to be legal RDMA targets when passed between processes. GASNet registers at startup memory segments for each known process. Only the process that has explicitly registered the segment can use RDMA on that region. Thus, one solution is to use statically duplicate registration of all application memory segments inside all servers. Another solution is to use dynamic registration inside servers. For un-registered addresses, GASNet uses internally an algorithm that selects between memory copies into bounce buffers for small messages or dynamic registration for large messages. A similar approach [195] is used internally inside MPI implementations.

Duplicate registration required breaking software encapsulation and extending the modifications from the UPC language runtime all the way to GASNet, which aims to be a language independent communication library. Instead, we chose to exploit the dynamic registration mechanism in GASNet. This turned out to be a fortuitous design decision as some underlying communication libraries (Cray uGNI) did not allow unconstrained registration of the same memory region in multiple processes.

Synchronization: The base GASNet implementation requires that communication operations are completed by the same task that has initiated them with the network. This constraint necessitates special handling of non-blocking communication primitives in our runtime. We introduced an extra synchronization step for message completion between clients and servers. Removing this constraint will require a significant redesign of GASNet communication infrastructure, which is not warranted by the observed performance. Furthermore, note that achieving good performance in practice required a careful tuning of atomic operations usage and runtime data structures padding to avoid false sharing.

Although it appears that these restrictions and design decisions are particular to the Berkeley UPC and GASNet implementations, most existing runtimes use similar software engineering techniques. As recently shown [108], combining MPI with `pthreads` still leads to performance degradation. We expect that trying to parallelize communication over processes while preserving shared memory similar in a different code base will encounter the same magnitude problems. Retrofitting spawning separate processes to act as communication servers into an existing runtime is likely to require coordinated changes across all abstraction layers.

3.5 Network Performance and Saturation

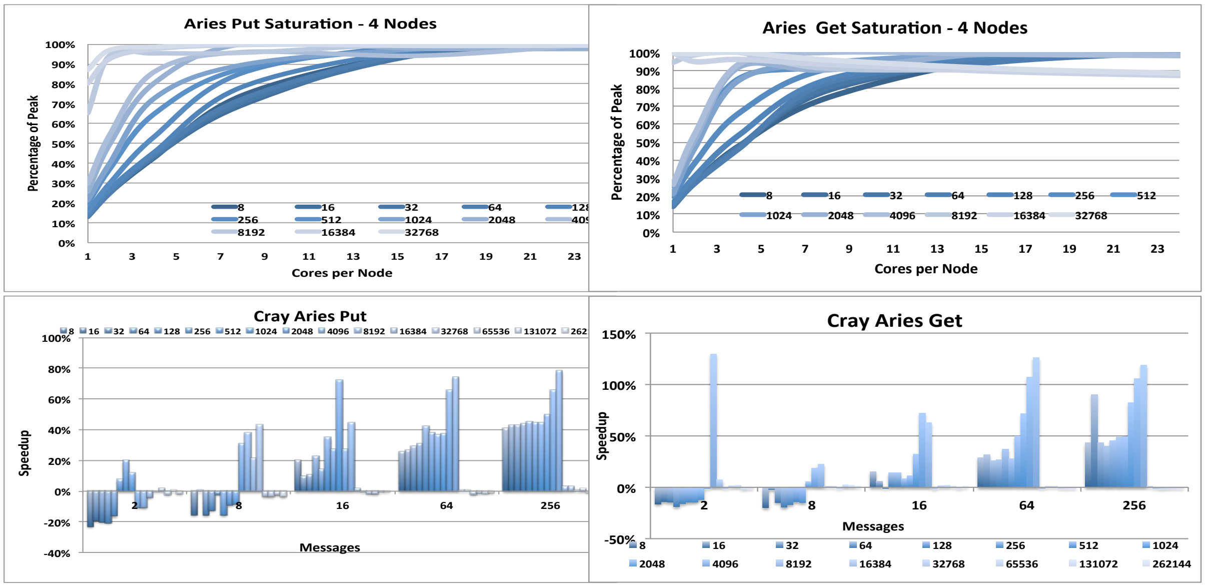


Figure 22. Top: Cray Aries network saturation, four nodes, 24 cores per node. Bottom: Performance improvements on Cray Aries with message size, number of messages. Experiment uses all sockets within node, one rank per socket, two servers per socket. Policy is round-robin of four messages to a server. Only small to medium messages benefit from parallelization, as indicated by the saturation graph.

Performance when using non-blocking communication is determined by the number of cores active within the node, as well as the number of outstanding messages per core. Our microbenchmark takes measurements for different numbers of cores active and reports the percentage of the peak *bi-directional* bandwidth attained at a particular message size and messages per core. *The peak attainable bandwidth*

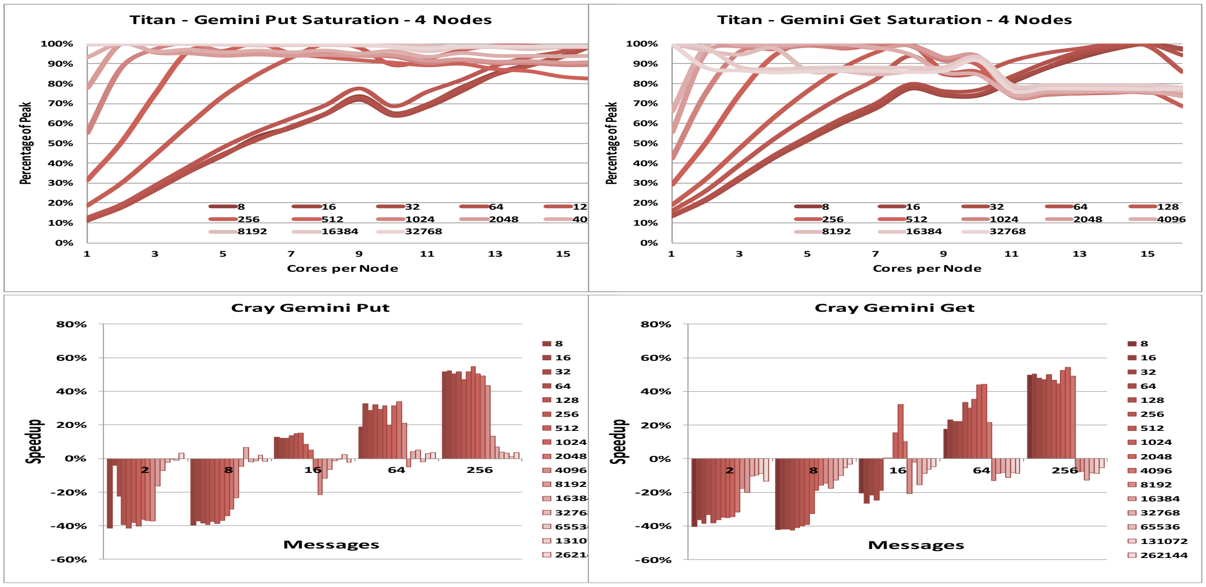


Figure 23. Top: Cray Gemini saturation, four nodes, 8 AMD Bulldozer and 16 integer units per node. Bottom: Performance improvements on Cray Gemini with message size, number of messages. Experiment uses one UPC process per node, three communication servers per node. Policy is round-robin of four messages to a server.

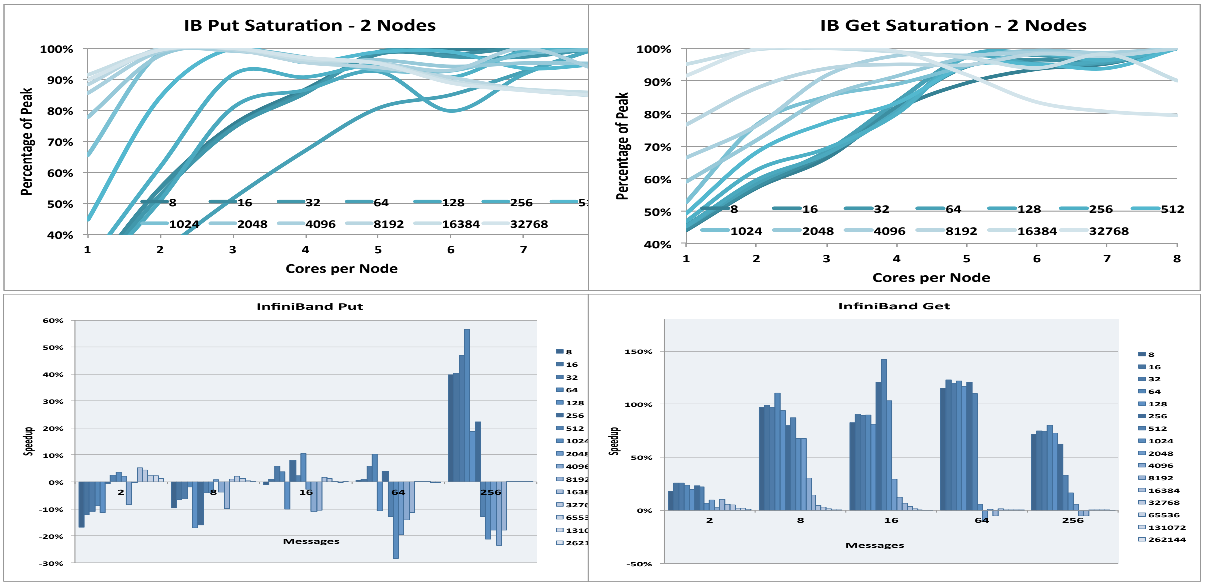


Figure 24. Top: InfiniBand saturation, two nodes, eight cores per node, two sockets. Bottom: Performance improvements of InfiniBand with message size, number of messages. All sockets active, two servers per socket. Only small to medium messages benefit from parallelization, as indicated by the saturation graph.

for a message size is determined as the maximum bandwidth observed across all possible combinations (cores, messages per core) at that size.

In Figures 22, 23 and 24 (top) we present the performance of the Berkeley UPC [31] compiler running on the GASNet [26] communication layer. We report results for InfiniBand and the Cray Aries/Gemini networks when each task is instantiated as a OS level process. `pthreads` are omitted for brevity, they match [108] process performance on Aries/Gemini and are significantly slower in InfiniBand.

Edison: is a Cray XC30 MPP installed at NERSC[3]. Each of its 5200 nodes contains two 12-core Ivy Bridge processors running at 2.4 GHz. Each processor includes a 20 MB L3 cache and four DDR3-1866 memory controllers which can sustain a stream bandwidth in excess of 50 GB/s. Every four nodes are connected to one Aries network interface chip. The Aries chips form a 3-rank dragonfly network. Note that depending on the placement within the system, traffic can traverse either electrical or optical links. While the attainable bandwidth is different, all other performance trends of interest to this study are similar for both link types.

Figure 22 (top) presents the results on Edison for a four node experiment (two NICs). Put operations are usually faster than Get operations, by as much as 25% for medium to large messages. For small to medium messages, Put operations need more cores than Get operations to reach saturation. For example, for 1024 byte messages, Puts require more than eight cores, while Gets require only four cores. For large messages, saturation is reached with only one core active. Increasing the number of active cores determines a bandwidth decrease for large messages.

Titan: is a Cray XK7 installed at the Oak Ridge Leadership Computing Facility [170]. It has 18,688 nodes, containing an AMD Opteron 6274 (“Interlagos”)

---

[3] National Energy Research Scientific Computing Center.

CPU. Each CPU exposes 16 cores, each with 16 integer units and 8 floating-point units arranged as 8 “Bulldozer” units. There are two NUMA domains containing four units each, which share an L3 cache. Nodes have 32GB of system memory. Pairs of nodes are connected to one Cray Gemini network interface chip. The Gemini chips form a 3D torus network. Additionally, each node has an NVIDIA Tesla K20X GPU with 6GB of memory connected via PCI Express 2.0.

Figure 23 (top) shows the results on Titan for a four node experiment (two NICs). Increasing concurrency improves throughput for small messages, reaching peak throughput when using all cores (Put) or one less than all cores (Get). Peak throughput for large messages occurs with only a few (Put) or one (Get) core for large messages. In many cases, peak throughput declines when changing from using only one NUMA domain (up to 8 cores) to using both NUMA domains (9 or more cores).

Carver: is an IBM Infiniband cluster installed at NERSC. Each of its nodes contains two 4-core Xeon X5550 (Nehalem) processors running at 2.67 GHz. Each processor includes a 8 MB L3 cache and three DDR3-1333 memory controllers which can sustain a stream bandwidth of up to 17.6 GB/s. Nodes are connected via QDR InfiniBand using a hybrid (local fat-tree/global 2D mesh) topology.

Figure 24 (top) presents the experimental results for two nodes. For small to medium messages, Put operations are up to 8X faster than Get operations, For “medium” messages we observe a 3X bandwidth difference for 512 byte messages. For Put operations, it takes four or more cores to saturate the bandwidth for messages shorter than 512 bytes. For larger messages, one or two cores can saturate the network, as illustrated for 32KB messages. Get operations saturate the network slower than Put operations, and it takes four or more cores to saturate for messages smaller that 8KB.

For both operations, increasing the number of active cores for large messages decreases performance by up to 20% for 32KB messages.

Both networks exhibit common trends that illustrate the challenges of tuning message concurrency:

- Put and Get operations exhibit different behavior on the same system and across systems. Optimizations need to be specialized per operation, per system.
- For small to medium messages, bandwidth saturation occurs only when multiple cores are active with multiple outstanding messages. Parallelization is likely to improve performance in this case.
- For medium to large messages, bandwidth saturation occurs with few cores per node, may degrade when increasing the number of cores per node. Parallelization may degrade performance in this case.

Evaluation of Parallelization. We evaluate the performance of our approach on the same microbenchmark in settings with one UPC thread per node and with one UPC thread per NUMA domain. The former is the typical setup in distributed applications that use GPUs. The latter is the setup used in manycore systems when mixing distributed and shared memory programming models.

We vary the number of server tasks from one to the number of cores available in the NUMA domain. We consider two strategies for message forwarding. In the first approach, clients forward communication in a round-robin manner to servers and also actively initiate some of their communication operations, similar to a hybrid SPMD+X configuration. In the second approach clients are inactive and forward all operations to servers in a round robin manner, similar to a dynamic tasking configuration such as HCMPI. Another tuning parameter is the number of operations consecutively forwarded to one server, varied from one to ten.

In our experiments the communication domains are confined within the same NUMA domain as their clients. We are interested in determining the optimal software configuration for our runtime which includes: 1) the number of servers per communication domain and NUMA domain; 2) order of choosing a server; 3) the message mix assigned to a server at any given time.

Cray Aries. The Cray Aries network provides two mechanisms for RDMA: Fast Memory Access (FMA) and Block Transfer Engine (BTE). FMA is used for small to medium transfers and works by having the processors writing directly into a FMA window within the NIC. The granularity of the hardware request is 64 bytes. BTE is employed for large messages. The processor writes a transfer descriptor to a hardware queue and the Aries NIC performs the transfer asynchronously. Communication APIs written on top of the Cray GNI or DMAPP system APIs switch between FMA and BTE for transfers in the few KB range. For GASNet the protocol switch occurs at 4KB.

GASNet [108] has been thoroughly re-engineered recently to provide good performance with `pthreads` on Cray systems. Figure 22 (bottom) shows the performance improvements for this instantiation of our server code. Most of the improvements of parallelization are directly correlated with the saturation graph in the same Figure 22. We observe similar behavior when one or both sockets within the Cray nodes are active.

Parallelization does not help much when there are fewer than eight messages available at the application level. For longer message trains parallelization does help and we observe speedups as high as 130%.

Medium size messages benefit most at a low degree of parallelization, smaller messages require more servers. This is correlated with the saturation slope in Figure 22.

For example parallelizing with two servers 64 Gets each of size 4096 bytes yields a 120% speedup, while parallelizing 64 eight byte operations yields only a 30% speedup. Parallelization does not yield great benefits for transfers larger than 4KB. This indicates that for this traffic pattern BTE transfers do not benefit from it.

We omit detailed results for the process based implementation. Transfers smaller than 8KB can be parallelized, while in our implementation larger messages are issued directly by the clients. When pointers to messages larger than 8KB are passed to a server process, GASNet switches to dynamic registration and the Cray uGNI library disallows registration of the same GASNet memory region into multiple processes. We have also experimented with using bounce buffers inside servers for large transfers without any worthwhile performance improvements.

Overall, `pthreads` based parallelization works very well on Cray Aries, while process based parallelization does not.

Cray Gemini. The Gemini interconnect is used in the Cray XE and XK series of supercomputers. The software stack of Gemini is similar to the newer generation Cray XC Aries, where both GNI FMA and BTE protocols are supported. On Titan, the CPUs are connected to the Gemini NICs using HyperTransport. HyperTransport allows lower injection latency for small transfers compared with the newer generation Cray Aries interconnect. Tracking remote completion for Put operations on Gemini is more complex especially with relaxed ordering of transfers [109], making Puts slower than the Get operations by up to 20%.

Figure 23 summarizes the performance observed on Gemini interconnect. The bottom figures show that when sufficient concurrency is available in the application, speedups of up to 50% (Put) and 54% (Get) can be achieved on the microbenchmark for small and medium messages using three communication servers. Unlike the results for Cray Aries shown in Figure 22, which showed increasing speedup for messages up to 4KB in size, on Cray Gemini speedup declines at 4KB.

Comparing the behavior on Gemini and Aries illustrates the fact that parallelizing communication in software better be supported by parallel NIC hardware. The software stack on both systems switches protocols to hardware BTE at 4KB messages. On Gemini, we can improve performance with parallelization only when configuring the runtime to use large (huge) pages; when using small pages hardware resources for memory registration are exhausted. On Aries performance improves when using small or huge pages.

The overhead of our runtime is lower in newer generation Cray XC machines compared with the Cray XK, making the performance benefit on XC evident with fewer outstanding transfers. Moreover, the performance benefit for Put operations is observed for more configurations than Get operations on Gemini. The opposite behavior is observed in newer generation Aries. Obviously, the difference between these interconnects is better handled by a tuning runtime and needs to be abstracted away form applications.

InfiniBand. On InfiniBand, performance is improved only by parallelization over processes.

Figure 24 (bottom) shows performance results on the InfiniBand system when using processes for parallelization. Overall, best performance results are obtained for small to medium messages, up to 4KB, which require multiple cores to saturate the network. Larger messages saturate with only a few cores and should not benefit from parallelization. Furthermore, when passing an address between processes, the underlying GASNet implementation chooses between RDMA using bounce buffers for messages smaller than a page and in-place RDMA with dynamic memory registration for larger transfers. Dynamic memory registration requires system calls which serialize the large transfers. Note that this combination of bounce buffers and dynamic registration also reduces the performance benefits of parallelization.

Parallelization provides best results for Get operations which saturate the network slower than Puts. In the best configuration we observe as much as 150% speedup from parallelization for 4KB messages. The technique is effective for Gets even when very few operations (as low as two) are available. For Gets, increasing the degree of parallelization improves performance and best performance is obtained when using most cores within the NUMA domain.

In the case of Puts the best speedup observed is around 80% for 128 bytes messages and the technique requires at least 32 messages per thread before showing performance improvements when using one socket. For Puts, increasing the degree of parallelization does not improve performance.

Again, understanding the saturation behavior is a good indicator for the benefits of parallelization of communication.

Application Level Behavior. Clearly our technique improves message throughput, but by introducing a level of indirection between the application and the network hardware, it may adversely affect the latency of individual operations.

In Figure 25 we present the impact of parallelization on the latency of eight byte messages on Cray Aries. Similar results are observed on InfiniBand and Cray Gemini. As illustrated, for a single eight byte message, the servers increase latency from $\approx 1.2\mu s$ to $\approx 1.6\mu s$. When multiple messages are overlapped, a single server increases per message latency from $\approx 0.6\mu s$ to $\approx 1\mu s$. Deploying multiple servers improves throughput and we observe per message latency as low as $\approx 0.4\mu s$, compared to $\approx 0.6\mu s$ in the default case.

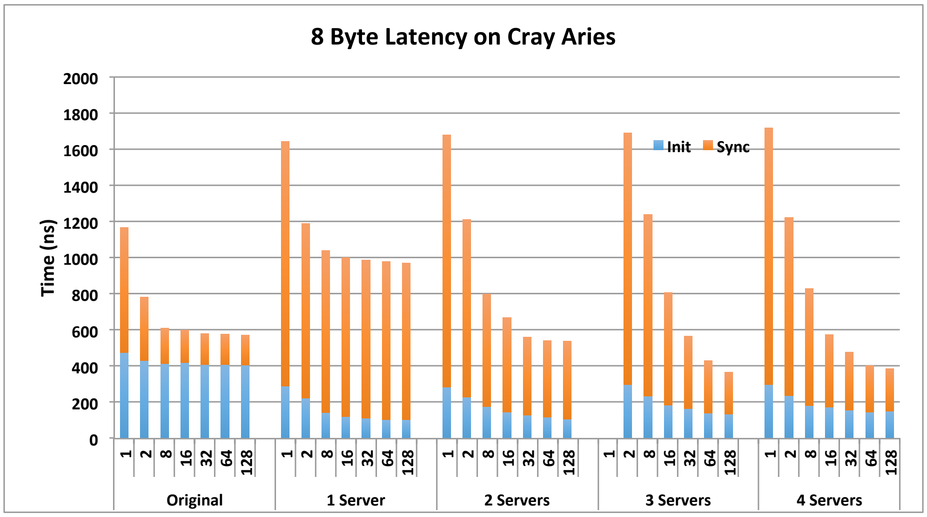


Figure 25. Speedup with parallel injection for an all-to-all microbenchmark on Cray Aries (left) and InfiniBand (right).

We further distinguish between message initiation (*init*) and completion (*sync*). This affects communication overlap at the application level, as this involves executing independent work between the *init* and *sync* operations. The results indicate that interposing servers improves the overlap attainable at the application level. The CPU overhead of *init* for a single eight byte message is reduced from $\approx 0.5\mu s$ to $\approx 0.25\mu s$ when using one server; we overall gained $\approx 0.25\mu s$ for extra independent work on the application CPU. With multiple servers we observe *init* overheads as low as $\approx 0.1\mu s$, compared to $\approx 0.4\mu s$ best case for unassisted communication.

These results reinforce the fact that parallelization is beneficial only after a minimum threshold on the number of message available at the application level. After this (low) threshold, deploying it in application settings is likely improve both message throughput and the amount of communication overlap attained at the application level. Both can improve application end-to-end performance.

3.6 Parallelizing Injection in Applications

Although the performance improvements are certainly encouraging for regular communication behavior, applications may exhibit instantaneous behavior which is adversary to our approach.

In some settings the message mix may be unpredictable and there may exist resource contention between servers and computation tasks. To handle message mixes we have implemented a dynamic parallelization of injection using a performance model that takes into account the expected number of messages, message size and type. To handle core contention we experiment with both "cooperative" scheduling and resource partitioning. All these mechanisms are exposed at the application level through a control API.

We experiment with a UPC+OpenMP multigrid benchmark we developed specifically for this study, as well as a 3D fast Fourier Transformation using the multithreaded FFTW [74] library and all-to-all collective communication.

Selective Parallelization: In order to provide optimal performance we need to know the number of messages, their size and type (Put or Get). We can then decide if parallelization improves performance and if so, we need to decide the optimal number of servers and message injection policy.

Based on the microbenchmark results, for any message size we determine a threshold on the number of messages to enable parallelization. For example, on Aries parallelization should be enabled any time there are more than four Get messages of size smaller than 8KB. For large messages we provide a direct injection policy by client tasks, bypassing the servers entirely. Any message larger than 8KB is directly injected by the client in our Aries implementation. As the actual number of messages does not matter, we provide application level APIs to simply enable and disable parallelization.

We also need to choose the number of servers. Based on the experimental data, the optimal point is different for small and medium messages: small messages require more parallelism, medium messages less. On the other hand, for a fixed number of servers, the instantaneous message concurrency is actually determined by the injection policy. By simply varying the number of consecutive messages assigned to a server, we can directly control their concurrency: the larger this number, the lower the concurrency.

In our implementation, we allow developers to specify a static concurrency for the communication subsystem, based on core availability or application knowledge. For a specific concurrency, we build a control model that decides how many consecutive messages of a certain size are assigned to a server queue, e.g. we assign every other small message to a new server and increase this threshold with message size.

Note that the required server concurrency depends whether the clients can be active or need to be inactive, as determined by the programming model. Same heuristics apply in both cases.

Core Management: The dedicated communication subsystem may run concurrently with computation tasks. In this case the cores may be oversubscribed with computation and communication tasks and performance is also determined by the core allocation. We explore both cooperative scheduling approaches as well as partitioning approaches. For cooperative scheduling, communication tasks in idle states are sleeping and we provide interfaces to explicitly `wind-up` and `wind-down` these tasks. We experiment with different strategies: 1) best-effort, no task pinning; 2) pinning the communication tasks to core domains; 3) partitioning cores between communication and computation tasks and parallelizing all transfers; and 4) partitioning cores between computation and communication tasks and doing selective parallelization.

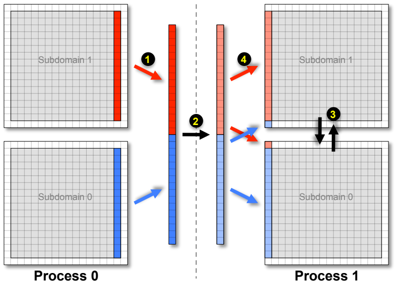


Figure 26. A 2D visualization of the exchange boundary communication phase among two neighboring processes each with two sub-domains. Note, only one direction (of 6) is shown. Only sends from process 0 are shown.

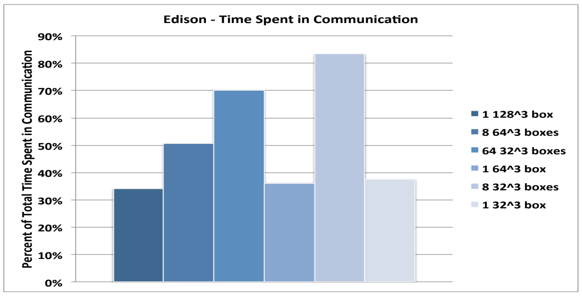


Figure 27. Fraction of time spent doing communication in miniGMG on Edison is heavily dependent on the number of boxes per process (independent messages) rather than total data volume.

miniGMG. Multigrid is a linear-time approach for solving elliptic PDEs expressed as a system of linear equations ($Lu^h = f^h$). That is, MG requires O(N) operations to solve N equations with N unknowns. Nominally, MG proceeds by iterating on V-Cycles until some convergence criterion is reached. Within each V-Cycle, the solution is recursively expressed as a correction arising from the solution to a smaller (simpler) problem. This recursion proceeds until one reaches a base case (coarse grid) at which point, one uses a conventional iterative or direct solver. Multigrid's recursive nature states that at each successively coarser level, the computational requirements drop by factors of 8×, but the communication volume falls only by factors of 4×. As a result, multigrid will see a wide range of message sizes whose performance is critical to guaranteeing multigrid's O(N) computational complexity translates into an O(N) time to solution.

miniGMG is a three thousand lines of C, publicly-available benchmark developed to proxy the geometric multigrid solves within the AMR MG applications [161, 236, 235]. Geometric multigrid (GMG) is a specialization of multigrid in which the PDE is discretized on a structured grid. When coupled with a rectahedral decomposition into sub-domains (boxes), communication becomes simple ghost zone (halo) exchanges with a fixed number of neighbors. In miniGMG, communication is performed by the MPI ranks, while all computation is aggressively threaded using OpenMP or CUDA.

For this chapter, using the publicly-available MPI+OpenMP and MPI+CUDA implementations as baselines, we developed several UPC+OpenMP and UPC+CUDA variants using either *Put* or *Get* communication paradigms with either barrier or point-to-point synchronization strategies. We only report results using the *Get* based implementation with point-to-point synchronization as it provides the best performance in practice. When compared to the original MPI+OpenMP version, our UPC+OpenMP variant always provides matching or better performance.

In order to minimize the number of messages sent between any two processes, miniGMG's ghost zone exchange was optimized to aggregate the ghost zones exchanges of adjacent sub-domains into a single message. Thus, as shown in Figure 26, two sub-domains collocated on the same node will: 1) pack their data into an MPI send buffer; 2) initiate an MPI send/recv combination; 3) attempt to perform a local exchange while waiting for MPI; and 4) extract data from the MPI receive buffer into each subdomains private ghost zone. In each communication round a MPI rank exchanges only six messages with its neighbors. While this approach to communication is common place as it amortizes any communication overheads, it runs contrary to the need for parallelism. The UPC+x implementations use the same algorithm.

As real applications use a variety of box sizes to balance AMR and computational efficiency with finite memory capacity and the desire to run physically realistic simulations, we evaluate performance using box sizes of 32^3, 64^3 and 128^3 distributed as one, eight or 64 boxes per UPC thread for both communication strategies. Some of the larger configurations will be limited by on-node computation, while smaller problems will be heavily communication-limited. Overall, due to varying degrees of required parallelism, aggressive message aggregation optimizations and different message sizes miniGMG provides a realistic and challenging benchmark to message parallelization.

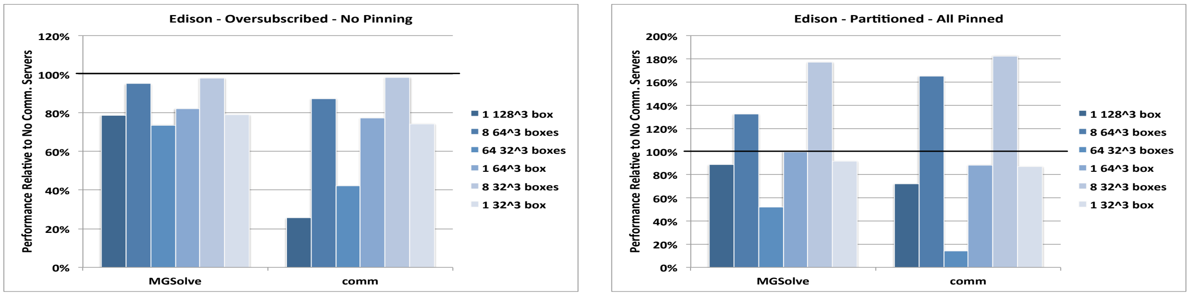


Figure 28. Performance of UPC miniGMG with parallelization relative to UPC miniGMG without parallelization on Cray Aries. Left: Oversubscribed, best effort 12 OpenMP tasks, 3 servers on 12 cores. Right: partitioned, pinned, 8 OpenMP tasks, 3 servers on 12 cores. Best performance requires partitioning and explicit pinning of all tasks. Parallelization results in performance improvements for some problem sizes.

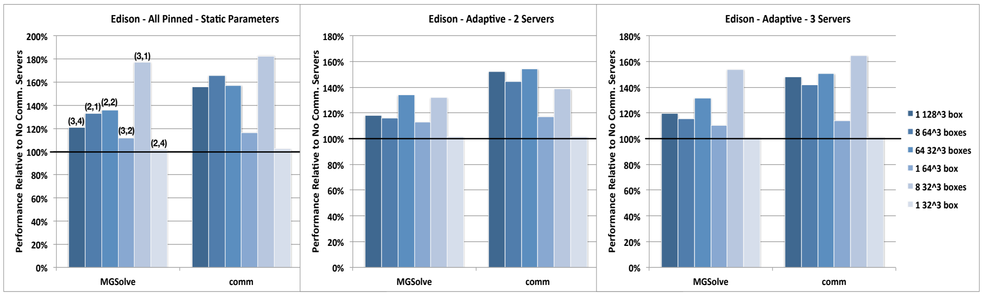


Figure 29. Performance of UPC miniGMG with **selective parallelization** relative to UPC miniGMG without parallelization on Cray Aries. Left: optimal settings (server, batch size) are annotated on the figure. Center and right: adaptive parallelism with two and three servers. Allocating more cores to communication improves performance.

miniGMG UPC+OpenMP Performance: Figure 27 presents the fraction of our UPC miniGMG solve time spent in communication for a variety of problem and box sizes.

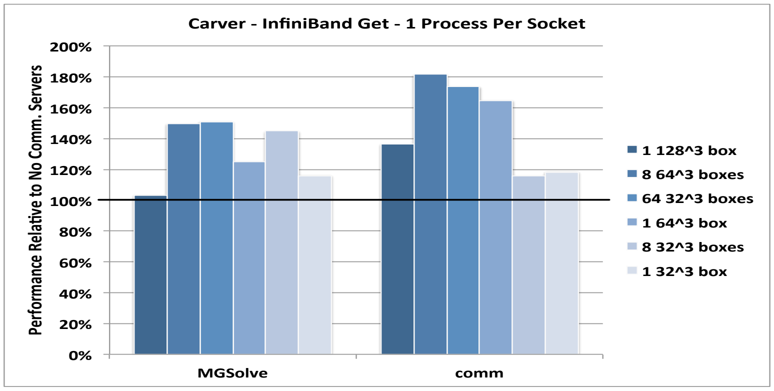


Figure 30. Performance of UPC miniGMG with communication servers relative to UPC miniGMG without parallelization, on InfiniBand, with 2 client processes per node (1 per socket).

As illustrated, the code can transition from computation-dominated to communication dominated with a sufficient number of boxes per process.

Both OpenMP threads and our communication server tasks require cores to run on. Figure 28 (left) presents the impact of using three communication threads compared to the baseline UPC+OpenMP implementation. In both cases, there are 12 OpenMP threads, but in the latter, the operating system must schedule the resultant 15 threads on 12 cores. *MGSolve* records the overall speedup on the multigrid solver while *comm* records the speedup in the communication operations. Inside the benchmark we explicitly use cooperative scheduling for the communication subsystem, i.e. communication tasks are sleeping when not needed. No thread is explicitly pinned and we have experimented with different OpenMP static and dynamic schedules. As illustrated, for all problem settings we observe performance degradation up to 25%.

Rather than oversubscribing the hardware and giving the scheduler full control to destroy any cache locality or to delay message injection, we experimented with eliminating oversubscription and pinning just the communication tasks. In this case we use 8 OpenMP threads and 3 pinned communication tasks. Although superior to oversubscription, performance is still less than the baseline. Detailed results are omitted.

Figure 28 (right) presents the speedup when hardware resources are partitioned among 8 OpenMP threads and 3 communication threads. Both OpenMP and communication threads are explicitly pinned to distinct cores and all communication is parallelized. Some problems observe substantial speedups (by as much as 70%), while some slow down by as much as 47%. On average we observe 2% slowdown and any performance degradation is explained by slowdown in communication.

Figure 29 (left) presents the best performance attained using selective parallelization at its optimal setting in a partitioned node. We now observe performance improvements for all problem settings, with a maximum of 76% and an average improvement of 40%. Figure 29 (center) shows results for selective parallelization using the adaptive strategy with two servers. Figure 29 (right) shows results of the adaptive strategy with three servers, giving a maximum improvement of 64% and an average improvement of 36%. As illustrated, allocating more cores to the communication subsystem improves performance and the adaptive strategy provides most of the possible performance gains.

For brevity we did not present detailed results on InfiniBand; they are similar to the results presented on the Cray system. Figure 30 shows an experiment with partitioned resources and parallelization enabled over three servers. Again we observe application speedup up to 80%.

These results indicate that under the current OS scheduling techniques, parallelizing communication successfully requires partitioning and pinning.

miniGMG UPC+CUDA Performance: The UPC+CUDA implementation of miniGMG offloads all computation to GPUs. In order to affect inter-process

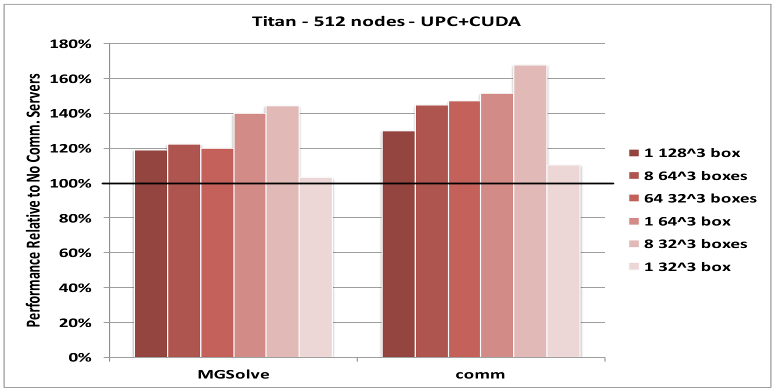


Figure 31. Performance of UPC+CUDA miniGMG with communication servers relative to UPC+CUDA miniGMG without parallelization, on Cray Gemini, with 1 client process per node.

communication, data is packed on the GPU and copied to the host. Unlike the MPI+CUDA version, the UPC+CUDA version can leverage the communication servers to maximize network performance as described in section 3.6. CUDA and GASNet have different memory alignment requirements for optimal performance; in this case alignment is according to CUDA's preference.

Table 1. Sizes of messages sent by UPC+CUDA miniGMG for each problem size tested. Bold sizes are those for which performance improvement is expected from adding concurrency on Cray Gemini

Problem Size	Message Sizes (bytes)
1×128^3 box	**128**, **512**, **2K**, 8K, 32K, 128K
8×64^3 boxes	**512**, **2K**, 8K, 32K, 128K
64×32^3 boxes	**2K**, 8K, 32K, 128K
1×64^3 box	**128**, **512**, **2K**, 8K, 32K
8×32^3 boxes	**512**, **2K**, 8K, 32K
1×32^3 box	**128**, **512**, **2K**, 8K

UPC+CUDA miniGMG experiments were run using the same sizes used in the previously described experiments. We use one UPC thread per node and one GPU per UPC thread. As only one CPU core is used per node and all computation is offloaded to the GPU, there are 15 idle cores which can be tasked as servers. Table 1 shows the message sizes that are sent for each problem size. We use parallel injection for message sizes shown in bold, which are the sizes for which improvement is seen on the point-to-point communication microbenchmark. Other message sizes are sent directly by the client. Figure 31 shows the performance with communication servers, relative to the same problem size without communication servers, for each problem size on 512 nodes (each with one GPU) on Titan. We observe overall speedups of up to 40% by using communication servers.

Collective Operations. Optimizing the performance of collective operations [211, 128, 243] has seen its fair share of attention and implementations are well tuned by system vendors. Due to their semantics, collectives are an obvious beneficiary of our techniques in application settings as they mostly require tasks to contribute equal amount of data to a communication pattern with a large fan-out.

In Figure 32 we show the aggregate bandwidth of an `all-to-all` operation implemented using UPC one-sided *Get* operations[4] with and without parallel injection on 1,024 nodes of Edison, accounting for 12,288 total cores in a hybrid setting. Our implementation initiates non-blocking communication in a loop and throttles the number of outstanding messages to 128 for scalability with nodes. Parallelizing injection improves performance up to 30% over the baseline UPC case for messages smaller than 4KB.

For reference we include the performance of the Cray tuned `MPI_alltoall`. This implementation selects different algorithms for small (Bruck's algorithm) and large (pairwise exchange) messages, while our microbenchmark uses a single algorithm for all message sizes. Parallel injection allows our implementation to provide greater bandwidth in the region of messages sizes where MPI and our implementation use

---

[4] Note that this implementation provides better performance than a *Put* based implementation.

similar algorithms. Performance is better than the MPI version for messages between 8B and 2KB. On a smaller (64-node) run, performance was better than MPI for messages between 16B and 32KB.

We observe similar performance improvements up to 30% on InfiniBand, detailed results omitted for brevity. We expect to see similar trends while parallelizing other operations such as reductions and broadcasts.

NPB UPC-FT. This benchmark implements the NAS Parallel Benchmarks [14] discrete 3D Fast Fourier Transform, using UPC for inter-node transpose communication and multi-threaded FFTW [74] for intra-node parallelization [221]. UPC-FT goes through two rounds of communication. For a problem of size $N_X \times N_Y \times N_Z$ run on a $P_X \times P_Z$ process grid, messages are $16 \cdot N_X/P_X \cdot N_Y/P_X$ bytes in the first round and $16 \cdot N_Y/P_Z \cdot N_X/P_X$ bytes in the second round.

Figure 33 shows the relative performance of UPC-FT on a class A size (256 × 256 × 128) problem on 1,536 cores of Edison, with the partitioning of the problem across nodes varied to produce first-round message sizes from 256B to 256KB while holding second-round message size constant at 8KB. FFTW is build with threading support using OpenMP and configured to use 8 threads per process. OpenMP and communication servers threads are pinned to cores. The “2 Servers” and “3 Servers” columns show the performance effect of using that number of communication servers and parallelizing everything, while “Adaptive” shows the performance with selective parallelization.

Speedups of up to 49% are seen for the smallest messages, they decrease with increasing message sizes, with speedups of 11% for 4KB messages. Incidentally, the best original performance is obtained for the AA 32 × 4 setting. The rightmost section of the figure shows results on a class D-$1/8$ size (1024 × 512 × 512) problem distributed across a 128 × 8 process grid on 12,288 cores of Edison, with first-round message sizes of 512B. A speedup of 23% is achieved on this problem used by NERSC for system procurements.

3.7 Discussion

Figure 34 summarizes the overall performance trends uncovered by this work. On the left hand side we compare the performance of a setting with one task per NUMA domain (hybrid parallelism in application) with our parallel injection. Parallelization occurs over two servers and for reference we include the peak bandwidth attainable on the system in any combination.

For both systems parallelization is effective for small to medium messages, up to 8KB on Aries and 32KB on InfiniBand. Parallelization does not improve the performance for large messages. For any message size, there is a gap between the parallelized injection and the *peak* attainable bandwidth. Most of this gap is accounted by injection concurrency and not by our implementation overhead.

For small to medium messages increasing the number of servers in 'PAR' closes the gap between attained and peak performance. For large messages, parallelization does not improve performance when compared to the original setting, yet there is a noticeable difference from peak bandwidth. In this case orthogonal concurrency throttling techniques as described by Luo [138] are required. Note that due to decoupling the communication into a stand-alone subsystem, these techniques are easy to implement in our architecture.

The right hand side graph in Figure 34 illustrates an intriguing opportunity. Medium messages at high concurrency achieve similar bandwidth to the best bandwidth achieved by large messages at any concurrency. This means that concurrency throttling or flow control techniques for large messages may be replaceable by message decomposition and parallel injection. We are currently investigating this tradeoff.

Overall we make the case for decoupling communication management from the application itself and transparently applying injection parallelization in conjunction with throttling in order to maximize throughput. Having a separate communication subsystem enables dynamic management on a node wide basis. This architecture fits naturally in both SPMD and dynamic tasking runtimes such as Habanero-C or HCMPI.

In our experiments, dedicating cores to communication affected only marginally, if at all, the end-to-end benchmark performance. Furthermore, for any problem where communication was present, its parallelization provided by far the best performance. We believe that dedicating a small number of cores to communication is feasible for many applications on existing systems. Of course, there may be computationally intensive applications that perform very little communication or synchronization.

Hardware evolutionary trends are also favorable to a decoupled parallel communication subsystem in application settings. There is likely to be enough core concurrency that a runtime system can instantiate a partition dedicated to communication management. This avoids scheduling problems when cores are oversubscribed. There also exists an expectation that in future systems the memory per core will decrease while the number of nodes will significantly increase. This implies that hybrid parallelism algorithms will have to use a small number of "traditional" communication tasks per node due to memory scalability problems inside runtimes (connection information), as well as the application levels (boundary conditions buffer space).

For hybrid programming such as UPC+OpenMP, it may seem that one can just fix `pthreads` and retrofit the principles we describe inside the applications themselves. The caveat is that the requirement to have both process and `pthreads`-based implementations for portability is unlikely to disappear in the foreseeable future. The first hybrid MPI+OpenMP studies [34] were published circa 2000. Fixing `pthreads` is not easy as illustrated by the performance in 2014. Furthermore, the low-level networking APIs and system software make this distinction necessary for performance portability, and unlikely to change.

3.8 Other Related Work

As already explained in Section 3.3, explicit communication parallelization for hybrid SPMD+{OpenMP,CUDA} codes has not been thoroughly explored due to implementation constraints. Rabenseifner et al [179] discuss its potential and implications on algorithm design, without detailed performance results. Similarly, communication parallelization has not been yet explored in dynamic tasking runtimes. There has been work inside the MPI implementation [63, 84] to improve performance for MPI_THREAD_MULTIPLE. These studies demonstrate improved performance only for microbenchmarks, mostly on IBM BG/P hardware. Recent work by Luo et al [139] describes an MPI implementation able to provide improved performance for hybrid MPI+OpenMP parallelism on InfiniBand networks. They use multiple endpoints for parallelism and show results for microbenchmarks and all-to-all operations. Dinan et al [59] discuss extensions to improve MPI interoperability with other programming models and `pthreads`, but no performance results are presented. Without performance portability, developers are unlikely to adopt explicit parallelization in their codes.

Multi-threading the runtime implementation has been explored for both one-sided and two-sided communication paradigms. Recent efforts by Si et al [195] examine multi-threading the MPI runtime implementation. This implementation uses OpenMP multi-threading to accelerate internal runtime routines such as buffer copying and derived datatypes, while maintaining the conventional serialized message injection. They report a tight integration of MPICH with the Intel OpenMP runtime and demonstrate results only for shared memory programming on a single Intel Xeon Phi. ARMCI [168] implements a portable one-sided communication layer that runs on most existing HPC platforms and uses `pthreads` for network attentiveness. While Put/Get operations are performed by their callers, ARMCI uses one separate thread per process for progress of *accumulate* operations.

The implementation of collective operations has received its fair share of attention. Yang and Wang [242, 243] discussed algorithms for near optimal all-to-all broadcast on meshes and tori. Kumar and Kale [128] discussed algorithms to optimize all-to-all multicast on fat-tree networks. Thakur et al [211] discussed the scalability of MPI collectives and described implementations that use multiple algorithms in order to alleviate congestion in data intensive operations such as all-to-all. All these algorithms initiate non-blocking communication with a large number of peers, thus our approach can be transparently retrofitted.

3.9 Conclusion

In this chapter we have explored the design aspects of a dedicated parallel communication runtime that handles message injection and scheduling on behalf of application level tasks. Our runtime is able to increase the instantaneous communication concurrency and provide near saturation bandwidth, independent of the application configuration and its dynamic behavior.

We strive to provide performance and portability by: 1) using a dual "parallelization" strategy where tasks dedicated to communication are instantiated as either processes or `pthreads`; 2) using a selective parallelization strategy guided by network saturation performance models; and 3) implementing either cooperative scheduling or core partitioning schemes.

This architecture is well suited for hybrid parallelism implementations that combine intra- and inter-node programming models, as well as dynamic tasking programming models. We show very good performance improvements for collective operations, as well as hybrid parallelism codes. As HPC systems with many cores per chip are deployed, such as the 72-core Intel Knight's Landing, core partitions dedicated to communication become feasible. This alleviates the need for improving the load balancing and cooperative kernel level task scheduling mechanisms.

Unfortunately, if performance portability is a goal, a dual parallelization strategy seems to be required for the near to medium future. Furthermore, during this work we uncovered limitations in existing system software in the area of memory registration and job spawning. These unnecessarily complicate the implementation of multithreaded runtimes such as ours.

3.10 Bridge

This chapter has described a tool-runtime integration with UPC, showing how such an integration can support online adaptation of communications parameters. UPC is a low-level language providing data abstractions but no explicit work abstractions. The next chapter describes a similar integration with HPX, a much higher-level library, with both data and work abstractions, and shows how the availability of work abstractions enables runtime tuning of the partitioning of work into tasks.

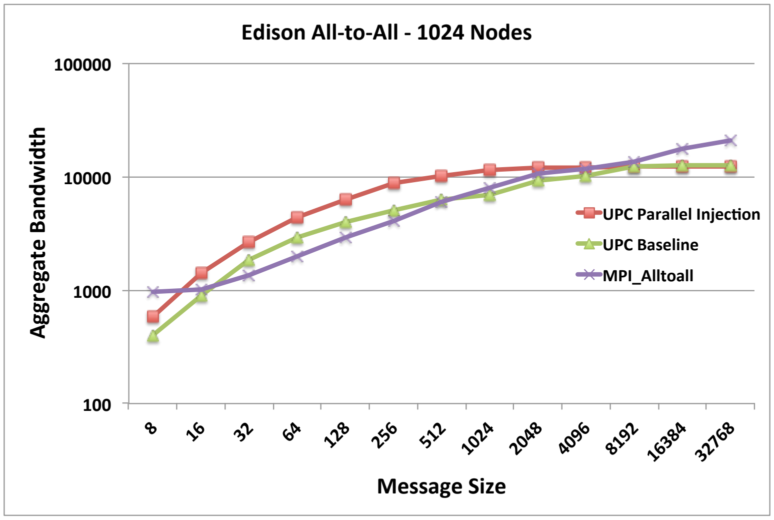


Figure 32. Aggregate bandwidth achieved with one-sided Put and Get operations using UPC without parallel injection, UPC with parallel injection, and MPI.

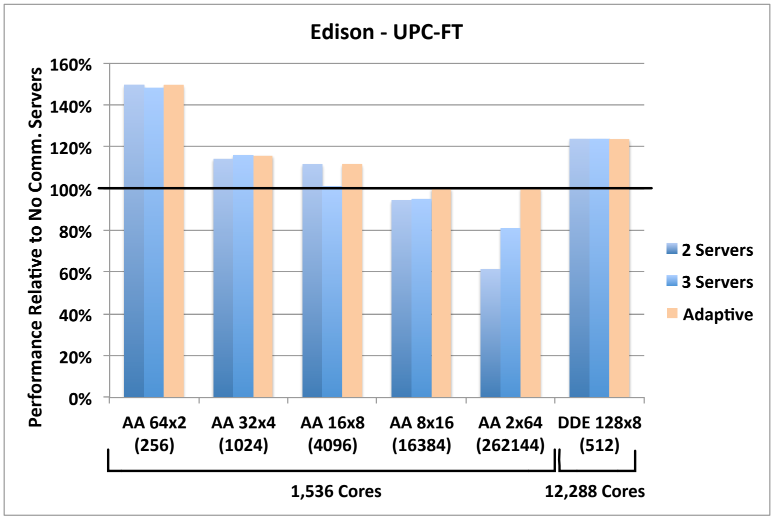


Figure 33. Performance of the UPC-FT benchmark with Class A problem sizes on 1,536 cores of Edison for different first-round message sizes with two or three communication servers, relative to performance without communication servers, and for a class D-1/8 problem size on 12,288 cores of Edison.

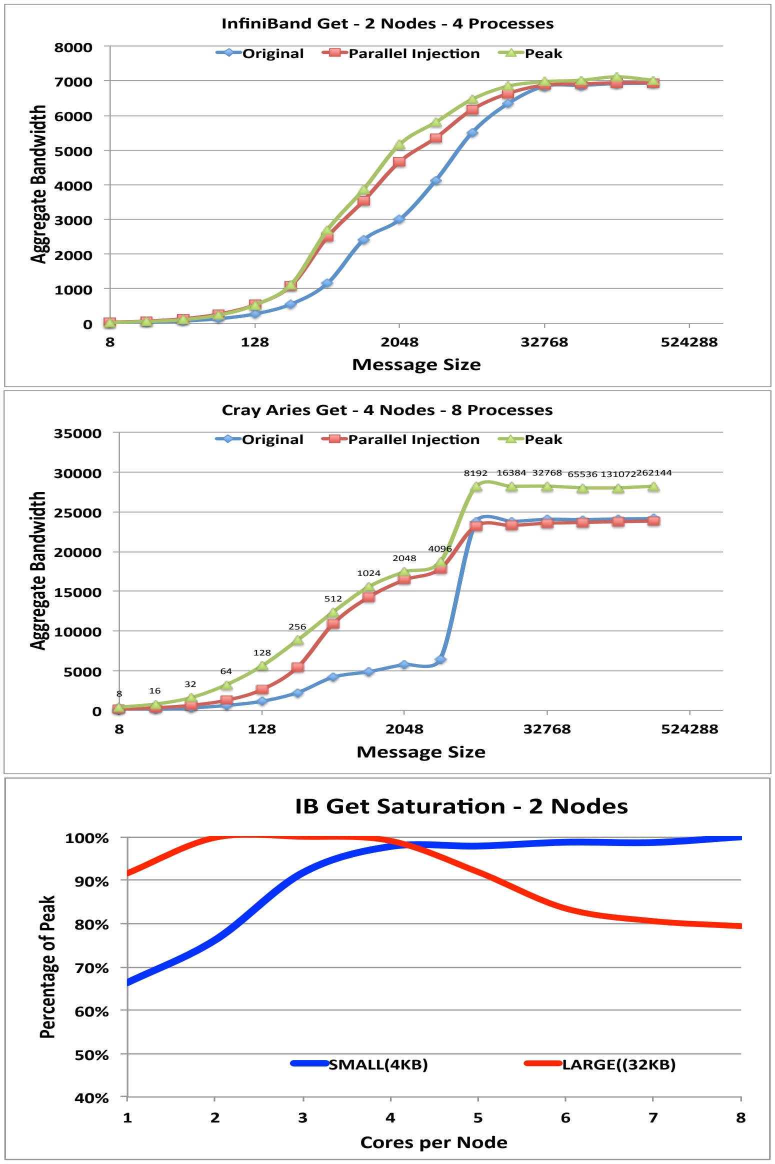


Figure 34. Overall trends comparing hybrid setup with 1 task per NUMA domain('Original'), with parallelization ('PAR') and peak attainable bandwidth on the system. 'PAR' uses only 2 servers. InfiniBand saturation with active cores for small and large messages.

CHAPTER IV

PERFORMANCE MEASUREMENT AND ONLINE ADAPTATION IN HPX

This chapter includes co-authored material previously published in Supercomputing Frontiers [102]. That paper was a collaboration with Kevin Huck, Allen Porterfield, Hartmut Kaiser, Allen Malony, Thomas Sterling, and Rob Fowler. I integrated APEX with HPX-3, developed the APEX policy engine, the APEX custom tuning framework, and ran and analyzed the HPX-3 experiments. Kevin Huck is the lead developer of APEX. Allen Porterfield and Rob Fowler developed the RCR framework used to collect power data. Hartmut Kaiser is the lead developer of the HPX-3 runtime. Thomas Sterling is the lead developer of HPX-5. I was minimally involved in the portions of the paper describing HPX-5, and those portions are not included in this document.

4.1 Introduction

The HPX runtime is a future-based many-task runtime. A task is a unit of work which, when created, produces a *future*, which can be used for synchronization and to retrieve results from the task. Data is provided to a task by passing that task a future. When a task is executing and it requests a value from a future, either the data is retrieved, locally or remotely, if the data is a result from a task whose execution has already completed, or the task *yields* if the data is not yet available. Tasks can therefore run for very short periods of time as they begin, request data not yet available, and yield. HPX provides a very low overhead scheduler to handle these short-running tasks. Dependencies between tasks are expressed implicitly at runtime, through waiting on futures.

Using a traditional performance monitoring tool with HPX will often produce incorrect results due to the unacceptably high overhead of profiling every task start, task yield, or task completion. Results will not be insightful because the dependencies between tasks are not captured. In this chapter, we describe the APEX performance monitoring tool, policy engine, and tuning framework, and use it to provide runtime feedback to the HPX runtime and HPX applications, dynamically adjusting task granularity to minimize idleness and scheduler overhead, and keep power consumption under a cap by dynamically adjusting the number of workers used by the runtime in scheduling tasks.

4.2 APEX Design

Overview. APEX aims to enable autonomic behavior in software by providing the means for applications, runtimes, and operating systems to observe and control their performance. Autonomic behavior requires performance awareness (introspection), and performance control/adaptation. APEX is designed around these two main components. APEX provides introspection from both top-down and bottom-up perspectives, including node-wide resource utilization data, energy consumption, and health information, all accessed in real-time. The introspection results are combined and associated with policy rules in order to provide the feedback control mechanism.

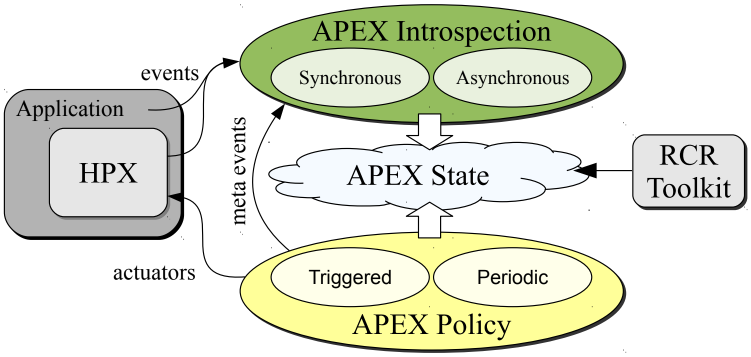


Figure 35. Design of the APEX introspection system.

Introspection. APEX collects *top-down* introspection data from a runtime system, library, or high-level application through an event-based *inspector* API. The software to be controlled is instrumented with this event API. APEX recognizes several types of logistic events such as initialization, termination, setting a process rank (*e.g.,* an MPI rank, or HPX locality ID), and creating a new thread. For measurement, APEX has instrumented timer-start and timer-stop calls, as well as sampled counter values (*e.g.,* bytes transferred, queue length, idle rate). These API calls enter APEX as events. Internally, APEX has several event *listeners* that perform actions based on the types of events that are passed in to APEX. Events are either handled by listeners immediately using synchronous code execution or are handled using asynchronous method invocation. For the asynchronous processing, the event is stored internally on a queue for background processing, and execution control is quickly returned to the code that called the APEX API. Custom events are also available to trigger specific policy engine rules. Further explanation of this behavior is presented in Section 4.2.

Bottom-up introspection data is collected from the operating system and hardware using periodic sampling. These measurements do not use events, but rather additional OS threads are spawned to periodically read values directly from available sources. On Unix-like systems, the `/proc` virtual filesystem files provide access to CPU, memory, network, disk, process, and operating system statistics. Resource Centric Reflection (RCR) [146, 147] provides a user-level API to access any counter available through PAPI, PERF_EVENTS, or a hardware instruction. RCRdaemon runs on protection ring 0 and supplies information about hardware resources shared by more than one core (*e.g.,* energy consumption, Last Level Cache events, or memory-controller usage) in a data structure that can be read at user-level. RCRdaemon uses a self-describing hierarchical data structure in a shared memory region to transmit protected counter values in an application-agnostic manner. The power interface reads these values and can be used by any application to acquire power/energy information. RCR calipers can be placed around any code region (up to the entire application) to measure energy used by that region. On Cray systems protection level 0 access is denied, but the Cray PM Counters [150] facility is available. RCRdaemon was therefore modified to get its data from this source. The values were then placed into the same data structure previously used. The user API was unchanged. Updates occur at the same rate as Cray updates /proc.

Event Listeners. As mentioned in Section 4.2, APEX events are processed by event listeners. Each listener is implemented as a C++ class, and as events pass through APEX, each instantiated listener is given access to the event object. The listeners implement handler methods for each event type available in the system. Notable event listeners in APEX include the *Profiling Listener*, the *Concurrency Listener*, the *Policy Engine Listener*, and the *TAU Listener*.

The profiling listener implements timer and counter measurement back-end processing in APEX. The salient events processed by the profiling listener include the `timer_start`, `timer_stop` and `sample_value` events. When the profiling listener gets a `timer_start` event, it creates a profiler object, generates a timestamp, and returns a handle to the profiler object. When the profiling listener gets a `timer_stop` event, it takes a second timestamp, puts the profiler object in a single-producer-single-consumer (spsc) queue for back-end processing, and returns. Each OS thread in the process has its own spsc queue to avoid contention. Similarly, when the profiling listener gets a `sample_value` event, it creates a profiler object, puts it in the spsc queue for back-end processing, and returns. The profiling listener has a background *consumer* thread that waits for a signal that indicates that data has been pushed onto one of the queues. When the consumer thread has been signalled, it clears all of the spsc queues of pending work by removing a profiler object from the queue, and updates the per-thread and per-process statistical profile for the running application. The currently executing profile can be queried subsequently at runtime through an introspection API. The optional TAU listener is similar to the profiling listener, with the exception that all processing is done synchronously through the TAU measurement library in order to generate a detailed profile or trace for offline, post-mortem performance analysis.

The concurrency listener works as follows. The salient events processed by the profiling listener are the `timer_start` and `timer_stop` events. When the concurrency listener gets a `timer_start` event, it pushes the timer ID onto a thread-specific stack, and returns. When the profiling listener gets a `timer_stop` event, it pops a timer ID off of the thread-specific stack. The concurrency listener also has a background consumer thread that periodically examines the top of each thread's timer stack and builds a histogram reporting the task currently being executed by each thread during that time quantum. At the end of execution, the histograms are written to files on disk and `gnuplot` [237] is used to visualize a concurrency graph of the application. Figures 36 through 38 are examples of concurrency graphs. The concurrency listener does not have a role in runtime adaptation, and is instantiated only when concurrency graphs are desired.

The Policy Engine. The most important listener component in APEX is the Policy Engine. The policy engine provides autonomic controls to an application, library, runtime, or operating system using the introspection measurements described in Section 4.2. Policies are rules that decide on outcomes based on the observed state captured by APEX. The rules are encoded as callback functions that are registered with APEX, and are either *triggered* or *periodic*. Triggered policies are invoked by an APEX event, whereas periodic policies, by definition, are executed at set intervals. The policy rule functions have access to the APEX API in order to request profile values from any measurement collected by APEX. Using these values to make logical decisions, the functions can change the behavior of the application by whatever means available, such as throttling threads, changing task granularity, or triggering data movement such as mesh refinement or repartitioning. In this way, the policy engine enables runtime adaptation using introspection data, engages actuators across stack layers, and can be used to invoke online auto-tuning support.

Global Performance Views. Thus far in the discussion performance introspection has been limited to local node observations. No performance information from remote nodes or processes is available implicitly to the local policy functions. However, there are situations in which global performance information is necessary to make runtime adaptation decisions for problems such as load balancing. In those cases, APEX provides a skeleton interface for exchanging local information in a distributed application scenario. The global exchange of local performance data in APEX is similar to that provided by TAUg [105], in which TAU performance data collected by an MPI application was exchanged using MPI functions. Rather than be tied directly to a specific communication infrastructure, APEX provides a skeleton interface to be populated using the distributed communication library used in the application to be controlled. Examples implemented so far include HPX-3, HPX-5 and MPI. The interface that the runtime has to implement includes two functions; `action_apex_get_value()` – each node gets local data to be reduced and performs an optional *put* (if implementing a push model); and `action_apex_reduce()` –– each node performs an optional *get* (if implementing a pull model), all remote node data is aggregated at root node, and an optional push broadcasts the aggregated result back out to the non-root nodes. Ideally, puts and gets are performed using one-sided communication such as remote distributed memory accesses (RDMA) or by using a Global Address Space (PGAS or AGAS).

HPX Integration. APEX is integrated with operating systems, runtime systems, libraries, and applications by instrumenting the code with calls to the APEX introspection API, as well as by registering desired policy functions and global communication. Because both HPX-3 and HPX-5 are task-based runtime systems, we added the instrumentation in the respective task schedulers, placing timer start/stop calls just before and after task functions are executed, taking special care to avoid measuring internal lightweight tasks such as “no-op”. `Sample_value()` calls were added to capture internal runtime statistics (*i.e.,* number of yields, steals, spins, *etc.*) and we added other instrumentation for initialization, thread creation and termination. Where applicable, we wrote policy functions and added the code to register the policy functions to perform adaptation of the runtime system. All the examples described in Section 4.3 modify runtime behavior in the same way, by setting a cap on the maximum number of active worker threads, so we also modified the HPX thread scheduler loop for worker threads to check the cap value and de-activate a worker thread if the number of active threads is greater than the thread cap. Even though we are measuring nearly every task executed by the runtime, our measurements show that the overhead introduced by APEX does not exceed 2%, and is usually less than 1%, depending on the granularity of the executed tasks. We believe that this is due to our asynchronous profile-processing combined with the small but sufficient amount of available processing capacity headroom when executing on many-core nodes. Global performance data is exchanged in HPX using the Active Global Address Space (AGAS).

4.3 Experimental Results

In order to demonstrate the features and capabilities of APEX, we integrated it with the HPX-3 runtime. We implemented a variety of policy rules, and we present a selection of them here, along with the applications that best demonstrate them. In this section, we present the following examples:

- HPX-3 1-D stencil code, runtime optimized for best performance
- HPX-3 miniGhost kernel, runtime modified to stay under a user-specified power cap

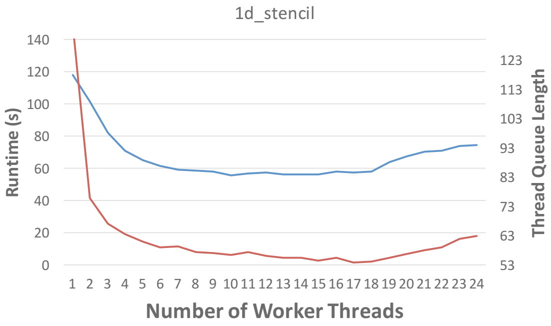
All of the experiments described below were conducted on Edison, a Cray XC30 system deployed at NERSC [212]. Edison has 5576 nodes with two 12-core Intel "Ivy Bridge" processors operating at 2.4 GHz, with a total of 48 threads per node (24 physical cores w/hyperthreading). The network on Edison is a Cray Aries interconnect with Dragonfly topology, with 23.7 TB/s global bandwidth. As LXK was not yet integrated with HPX, the applications were executed on the Compute Node Linux (CNL) operating system.

HPX-3 1-D Stencil Code. The 1D stencil code is a simple, iterative heat-diffusion solver using a 3-point stencil, used as an example code for HPX-3, and for which multiple versions are available with different optimizations applied. The simplest version represents the computation for each data point as an individual future, but the performance of this version is extremely poor as the task granularity is far too small. The version with good performance partitions the data into a user-configurable number of equally-sized chunks, with the computation on each chunk being represented as a future. Within a node, performance initially increases with an increasing number worker threads, but then decreases.

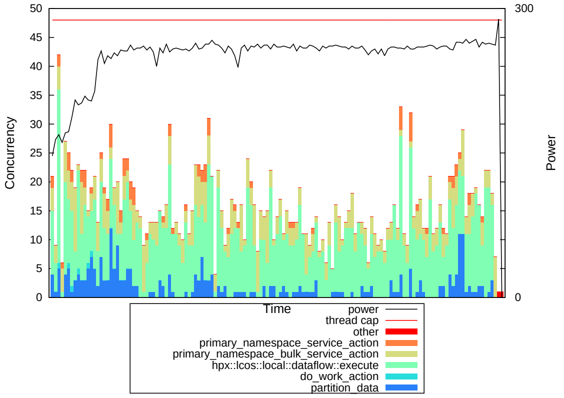
Figure 36a shows the runtime (blue line) of the 1D stencil code as function of number of worker threads from 1 to 24, which is the number of physical cores available on Edison nodes. It also shows that runtime is highly correlated with the average thread queue length (red line), which is a counter exposed by the HPX-3 runtime representing the number of tasks waiting to execute on worker threads. APEX can query the thread queue length while the program is executing and adjust dynamically the number of worker threads allocated to minimize runtime.

Figure 36b shows the concurrency graph for the execution of the 1D stencil code run on 100,000,000 elements partitioned into 1000 chunks with 48 worker threads, which is the number of logical cores available on an Edison node with hyperthreading enabled. Actual concurrency is substantially lower, as many tasks are waiting on dependencies to complete before becoming eligible to run, and there is substantial variability in actual concurrency over time. This execution takes 138 seconds to run. Figure 36c shows the concurrency graph for an execution of the same problem size but with 12 worker threads, which produces the shortest runtime of any number of worker threads. That execution takes 61 seconds to run.

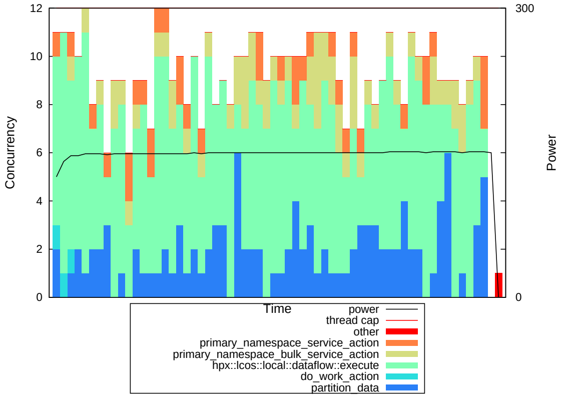
Figure 36d shows the concurrency graph for the same problem size and an initial number of worker threads of 48, but using discrete hill-climbing search to minimize the average thread queue length. This converges on 13 worker threads (*vs.* the optimal value of 12) and does so quickly enough that the overall runtime is nearly as fast (64 seconds) as starting with the optimal number.

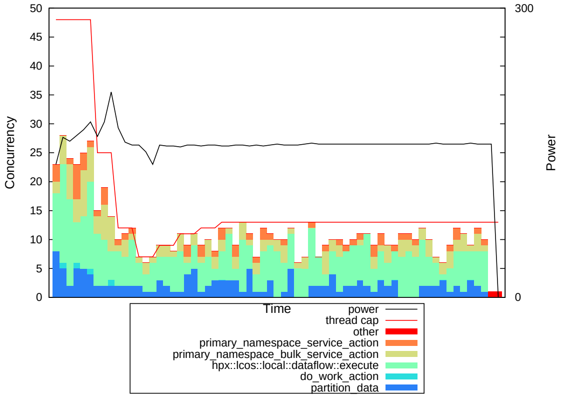


(a) 1D stencil strong scaling. This chart shows the correlation between the execution time (blue line) and the queue lengths (red line) when running with different numbers of threads on Edison.



(b) 1D Stencil unthrottled. This concurrency chart shows a stacked bar chart with the periodic (1 Hz) status of each OS thread. The max number of threads is 48, and the instantaneous power for each sample is the black line.





(c) 1D Stencil with ideal number of threads. This concurrency chart shows the periodic (1 Hz) status of each OS thread. The number of threads is fixed at 12, and the instantaneous power for each sample is the black line.

(d) 1D Stencil throttled by APEX. This concurrency chart shows the periodic (1 Hz) status of each OS thread. The number of active threads starts at 48, but is throttled while APEX searches for an optimal number of active threads to minimize execution time. The evolving thread cap is the red line, and the instantaneous power draw is the black line.

Figure 36. Performance behavior of HPX 1D Stencil under different throttling policies.

HPX-3 miniGhost kernel. MiniGhost [19], developed as part of the Mantevo project [96], is a finite difference miniapp simulating heat diffusion over a three-dimensional domain. The original version uses OpenMP intra-node and MPI inter-node. It has been ported to HPX-3 [7]; this version uses HPX for both intra- and inter-node parallelism. The HPX version provides better performance than the original OpenMP version.

Figure 37 shows that there are diminishing returns from allocating additional worker threads to MiniGhost. This suggests than we can throttle the application by cutting back on the number of worker threads to reduce energy usage while avoiding substantial performance degradation. Figure 38a shows the concurrency with 48 worker threads, the number of logical cores on an Edison node. While not all available worker threads are used, the application will often use slightly more than the 24 physical cores available. With 48 worker threads, MiniGhost runs in 92 seconds and uses about 275 Watts of power. Figure 38b shows the concurrency when the initial number of worker threads is set to 48 but the thread cap is dynamically adjusted to keep power at or below 200 Watts. APEX converges on a thread cap of 20, yielding 200 Watts of power usage, a 33% reduction in power, and a runtime of 103 seconds, a 12% increase in runtime.

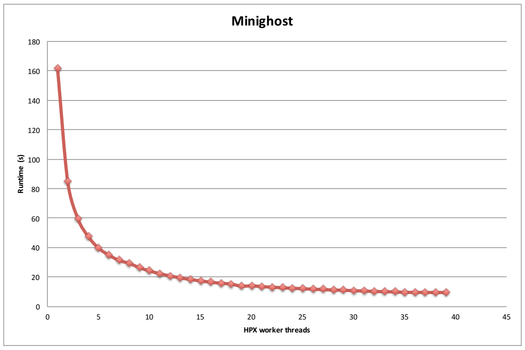
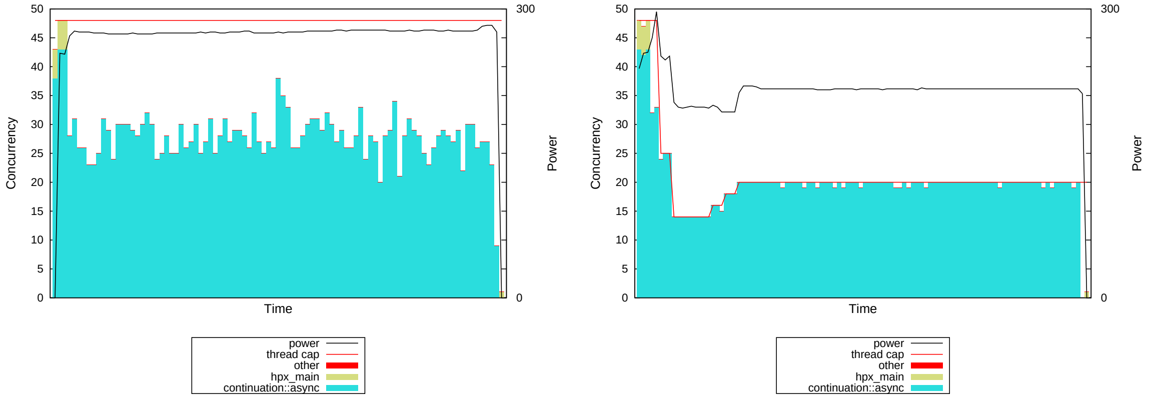


Figure 37. HPX miniGhost strong scaling.



(a) miniGhost Baseline. This concurrency chart shows a stacked bar chart with the periodic (1Hz) status of each thread. The max number of threads is 48 (red line), and the instantaneous power for each sample is the black line.

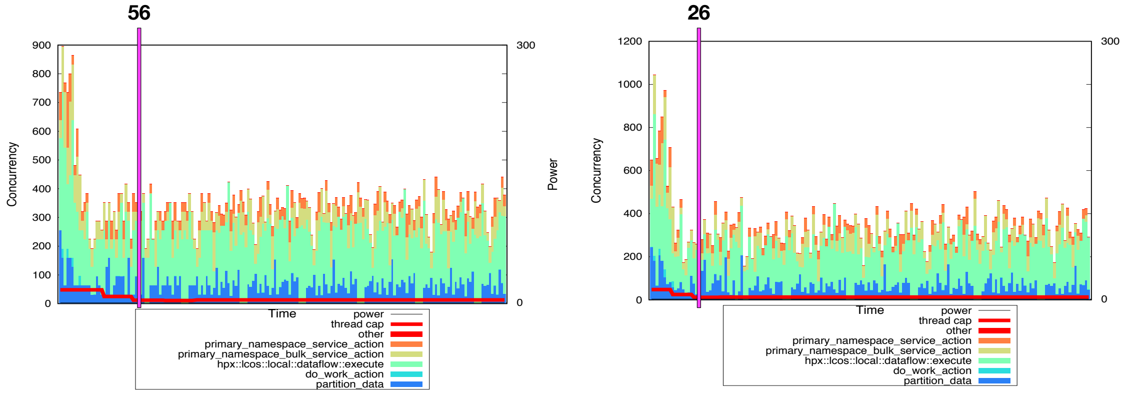
(b) miniGhost Throttled. This concurrency chart shows a stacked bar chart with the periodic (1Hz) status of each thread. The max number of threads starts at 48, but is throttled while APEX searches for an optimal number of active threads to keep under the power cap. The evolving thread cap is the red line and the instantaneous power for each sample is the black line.

Figure 38. Energy usage of HPX miniGhost under different throttling policies.

4.4 Tuning with a Global View

As part of the HPX-3 integration, APEX exposes its counters through the HPX-3 performance counter interface, allowing nodes to share performance information through the Active Global Address Space. This enables the opportunity to perform a global rather than local tuning run. In the tuning runs described previously in this chapter, the tuning is purely local: each node runs it own tuning session, which does not communicate with other nodes.

If we expose counters globally, we can instead run a global tuning session in which we use the Parallel Rank Ordering search strategy provided by Active Harmony to explore multiple parameter settings simultaneously. Here, we use the 1D Stencil version 8 benchmark, which is fully distributed, and tune over task granularity. The “bad” regions of the search space are particularly bad, so we want to minimize time spent searching them. In a local search replicated on each node, each node will explore the “bad” regions. With a global search, locality 0 retrieves performance data for other localities through the AGAS and reports these values to the tuning engine. The tuning engine then proposes multiple parameter settings to evaluate simultaneously, and these are retrieved by other localities, also over the AGAS. Figure 39 shows the evolution of a local tuning run (left) and a global tuning run (right) on 1D Stencil version 8. The global tuning run converges in 24% fewer overall iterations than the local tuning run, and in 46% of the wallclock time.



(a) Concurrency view of a 1D Stencil local tuning session.

(b) Concurrency view of a 1D Stencil global tuning session

Figure 39. Local and global tuning sessions of HPX 1D stencil.

4.5 Conclusion

The quest for exascale brings fundamentally new challenges to performance and to productivity. The solutions that will likely usher in the exascale era will require software designers and users to embrace performance heterogeneity and variability. We believe that any successful implementation will have to integrate performance introspection, *in situ* analysis, and adaptation in an exascale system stack. The XPRESS project has developed a prototype of APEX integrated with HPX-3 and HPX-5 for use in OpenX. We have demonstrated APEX with several benchmark examples, and we believe that the APEX framework is generally applicable to other X-stack runtime efforts.

There is considerable work that can be done with respect to APEX. In the short term, we would like to conduct more robust application experiments and to explore behavior larger scales on different platforms. As more applications are developed using HPX, we hope to have a greater opportunity to demonstrate the APEX capabilities for runtime adaptation. With that in mind, new applications will present more and better policy (optimization) rules, both for specific applications and to generalize these in the operating system and runtime libraries. In particular, we are interested in possible policy rules that address heterogeneous HPX-3 code that can be executed on GPGPUs, as well as many-core architectures such as the Intel Phi. We plan to develop more policy rules that specifically address the SLOWER design principles of the ParalleX model [199]. We soon will be exploring the multi-objective optimization opportunities available in the development branch of Active Harmony. With that support, we can tune with respect to both performance and energy efficiency, as well as to any other application-specific metrics. Finally, we believe that APEX has applications outside of the XPRESS project, and that it can be successfully integrated into other runtime systems and parallel execution models with controllable parameters, including OpenMP, MPI, and OmpSs. It can serve as a framework for triggering application-specific optimizations such as adaptive mesh refinement, load balancing, and other dynamic behavior.

4.6 Bridge

This chapter has described a tool-runtime integration with HPX, showing how such an integration can support online adaptation of work partitioning, and how this is supported by the presence of high-level work abstractions in HPX. The next chapter describes a similar integration with Spark, which provides a still higher level of abstraction in which work is expressed *declaratively*, describing *what* is to be computed but not how, and in which the runtime is free to move data within a storage hierarchy. This affords additional freedom to the runtime to alter the distribution of work and data across nodes in a distributed system.

CHAPTER V

STORAGE OPTIMIZATION AND VARIABILITY IN SPARK

This chapter includes co-authored material previously published in the Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2016) [41], the 2016 Cray Users Group symposium [39], and the Workshop on Performance and Scalability of Storage Systems [42]. This work was performed by myself, Costin Iancu, Khaled Ibrahim, Shane Canon, Jay Srinivasan, and Allen Malony. I developed all the instrumentation and analysis code and ran and analyzed all experiments. Khaled Ibrahim did the initial port of Spark to Cray Extreme Scale Mode. Shane Canon developed the Shifter container system. Jay Srinivasan assisted with installation of software and configuration of computational resources at NERSC. Costin Iancu provided valuable feedback on experimental design. I wrote the papers with Costin Iancu. Costin Iancu, Khaled Ibrahim, and Allen Malony edited the papers.

5.1 Introduction

Spark is a data analytics framework with a declarative style of programming. The application developer expresses operations on data without reference to parallelism. The runtime then partitions and distributes work, as well as handling resiliency through checkpointing and/or recomputation. A notable feature of the runtime is that it manages data in memory as well as on disk.

A traditional performance monitoring tool will not provide useful performance results for Spark applications largely due to the Spark runtime's lazy evaluation and the potential for a given partition to be computed multiple times. Without runtime-tool integration, it is impossible to determine which line of application code is responsible for a particular task executing at a particular time. In fact, tasks with no dependence relation can have a performance influence on one another through their shared use of the Block Manager, which is in charge of evicting data from memory and/or disk when storage is exhausted.

In this chapter, we describe the instrumentation of the Spark runtime in a way that allows us to correlate application-level directives with their eventual effect on the storage hierarchy. In so doing, we discover the cause of a major scalability bottleneck when Spark is deployed on supercomputers: the distributed filesystem. We evaluate techniques for mitigating this bottleneck, and show that the distributed filesystem also exacerbates other types of performance problems, such as recomputation resulting in excessive reads from disk of the same data. We develop a policy which automatically persists the correct partitions to disk to avoid these excessive reads.

5.2 Motivation

Frameworks such as Hadoop [231] and Spark [248] provide a productive high level programming interface for large scale data processing and analytics. Through specialized runtimes they attain good performance and resilience on data center systems for a robust ecosystem of application specific libraries [83, 157, 9]. This combination resulted in widespread adoption that continues to open new problem domains.

As multiple science fields have started to use analytics for filtering results between coupled simulations (e.g. materials science or climate) or extracting interesting features from high throughput observations (e.g. telescopes, particle accelerators), there exists plenty incentive for the deployment of the existing large scale data analytics tools on High Performance Computing systems. Yet, most solutions are ad-hoc and data center frameworks have not gained traction in our community. In this chapter we report our experiences porting and scaling Spark on two current very large scale Cray

XC systems (Edison and Cori), deployed in production at National Energy Research Scientific Computing Center (NERSC) [163].

In a distributed data center environment disk I/O is optimized for latency by using local disks and the network between nodes nodes is optimized primarily for bandwidth. In contrast, HPC systems use a global parallel file system, with no local storage: disk I/O is optimized primarily for bandwidth, while the network is optimized for latency. Our initial expectation, was that after porting Spark to Cray, we can then couple large scale simulations using $O(10^4)$ cores, benchmark and start optimizing it to exploit the strengths of HPC hardware: low latency networking and tightly coupled global name spaces on disk and in memory.

We ported Spark to run on the Cray XC family in Extreme Scalability Mode (ESM) and started by calibrating single node performance when using the Lustre [27] global file system against that of an workstation with local SSDs: in this configuration a Cray node performed up to 4× slower than the workstation. Unlike clouds, where due to the presence of local disks Spark shuffle performance is dominated by the network [183], file system metadata performance initially dominates on HPC systems. Perhaps expected by parallel I/O experts [140], the determining performance factor is the file system metadata latency (e.g. occurring in `fopen`), rather than the latency or bandwidth of read or write operations. We found the magnitude of this problem surprising, even at small scale. Scalability of Spark when using the back-end Lustre file system is limited to $O(10^2)$ cores.

After instrumenting Spark and the domain libraries evaluated (Spark SQL, GraphX), the conclusion was that a solution has to handle *both* high level domain libraries (e.g. Parquet data readers or application input stage) *and* the Spark internals. We calibrated single node performance, then we performed strong and weak scaling studies on both systems. We evaluate software techniques to alleviate the single node performance gap in the presence of a parallel file system:

- First and most obvious configuration is to use a local file system, in main memory or mounted to a single Lustre file, to handle the intermediate results generated during the computation. While this configuration does not handle the application level I/O, it improves performance during the Map and Reduce phases and a single Cray node can match the workstation performance. This configuration enables scaling up to 10,000 cores and beyond, for more details see Section 5.6. We have extended and released the Shifter [114] container framework for Cray XC with this functionality. Deploying Spark on Shifter has unexpected benefits for the JVM performance and we observe 16% performance improvements when running in memory on ≈ 10,000 cores.
- As the execution during both application initialization and inside Spark opens the same file multiple times, we explore "caching" solutions to eliminate file metadata operations. In Spark, the number of files used grows linearly with the number of cores, while the number of file opens grows quadratically with cores. We developed a layer to intercept and cache file metadata operations at both levels. A single Cray node with pooling also matches workstation performance and overall we see scalability up to 10,000 cores. Combining pooling with local file systems also improves performance (up to 17%) by eliminating system calls during execution.

On Cori we also evaluate a layer of non-volatile storage (`BurstBuffer`) that sits between the processors' memory and the parallel file system, specifically designed to accelerate I/O performance. Performance when using it is better than Lustre (by

3.5× on 16 nodes), but slower than RAM-backed file systems (by 1.2×), for *GroupBy*, a metadata-heavy benchmark. With `BurstBuffer` we can scale Spark only up to 1,200 cores. The improvements come from better `fopen` scalability, rather than read/write latency and illustrate the principle that optimizing for the tail is important at scale: the `BurstBuffer` median open latency is higher than Lustre's, but its variance is much smaller than on Lustre.

Besides metadata latency, file system access latency in `read` and `write` operations may limit scalability. In our study, this became apparent when examining iterative algorithms. As described in Section 5.7, the Spark implementation of PageRank did not scale when solving problems that did not fit inside the node's main memory. The problem was the interplay between resilience mechanisms and block management inside the shuffle stage in Spark, that generated a number of I/O requests that increased exponentially with iterations. This overwhelmed the centralized storage system. We fixed this particular case at the algorithmic level, but a more generic approach is desirable to cover the space of iterative methods.

Overall, our study indicates that scaling data analytics frameworks on HPC systems is likely to become feasible in the near future: a single HPC style architecture can serve both scientific and data intensive workloads. The solution requires a combination of hardware support, systems software configuration and (simple) engineering changes to Spark and application libraries. Metadata performance is already a concern for scientific workloads and HPC center operators are happily throwing more hardware at the problem. Hardware to increase the node local storage with large NVRAM will decrease both metadata and file access overhead through better caching close to the processors. Orthogonal software techniques, such the ones evaluated in this chapter, can further reduce metadata impact. In fact, at the time of the publication, our colleagues at NERSC have demonstrated Spark runs at $\approx 50,000$ cores using Shifter with our Lustre mounted local file system configuration. An engineering audit of the application libraries and the Spark internals will also eliminate many root causes of performance bottlenecks.

5.3 Spark Architecture

Apache Spark [248] and Hadoop [231] are open-source data analytics frameworks, designed to operate on datasets larger than can be processed on a single node while automatically providing for scheduling and load-balancing. They implement the Map-Reduce model [58] using an abstraction in which programs are expressed as data flow graphs. The nodes in the graph are of two types: *map operations*, which are purely local, and *reduce operations*, which can involve communication between nodes.

The traditional MapReduce framework [58] is limited to acyclic graphs, preventing efficient representation of iterative methods, and it uses data redundancy to provide resiliency. Spark can handle cyclic and acyclic graphs, and provides resiliency through *resilient distributed datasets* [247] (RDD), which carry sufficient information (lineage) to recompute their contents. In particular, the ability to express iterative algorithms accelerated Spark's adoption.

Programs are expressed in terms of RDDs derived from transformations of other RDDs (e.g. Map) and actions (e.g. Reduce). The application developer can choose to request that certain RDDs be cached in memory or saved to disk. The developer therefore has to make decisions based on tradeoffs between the costs of storage (in memory and time) and recomputation (in time). RDDs are lazily evaluated, which creates challenges [13] in attributing performance to particular lines or regions of code, as they do not execute until they are needed.

In Spark, the *Master* node executes a driver program, which creates the data flow graph by applying transformations and actions to RDDs, and partitions ownership of data to worker nodes within the cluster. When the result of an uncomputed RDD partition is needed, a *job* is created, consisting of multiple *stages*. Within a stage, only intra-partition communication can occur. All inter-partition communication happens at stage boundaries, through a process called *shuffling*, as shown in Figure 40. By deferring any computation until a result is needed, the scheduler can schedule work to compute only what is necessary to produce the result. In the event of the loss of a partition, only that partition needs to be recomputed.

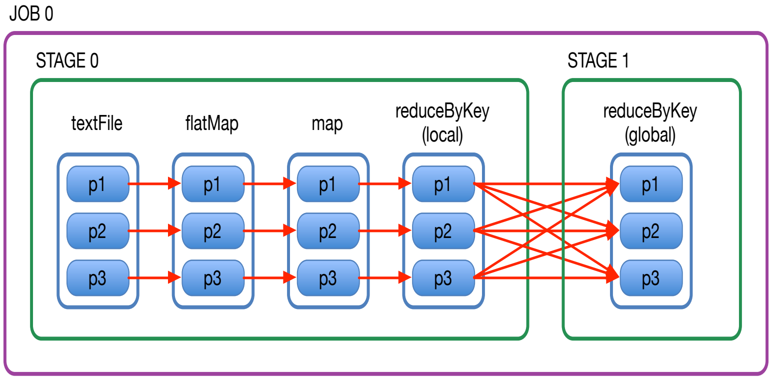


Figure 40. Decomposition of a Spark job into stages and tasks on partitions, with inter-partition communication limited to stage boundaries.

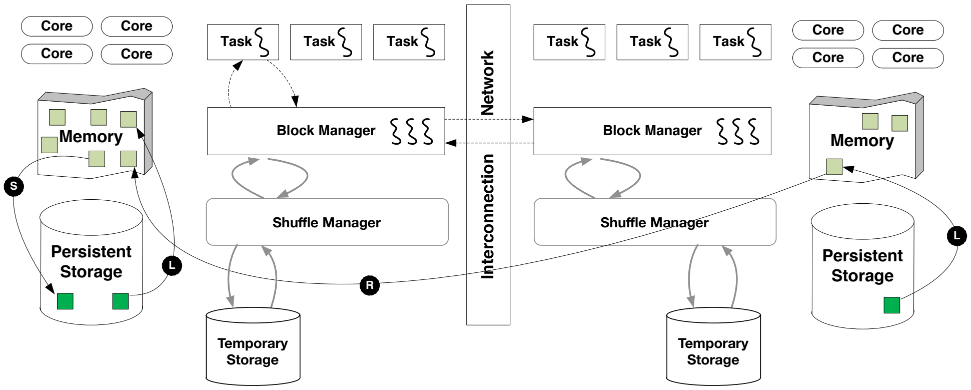


Figure 41. Data movement in Spark and the interaction with the memory hierarchy.

Data Movement in Spark. Data movement is one of the performance determining factors in any large scale system. In Spark, data is logically split into *partitions*, which have an associated worker task. A partition is subdivided into *blocks*: a block is the unit of data movement and execution. Figure 41 shows the interaction of the Spark compute engine with the block and shuffle managers, which control data movement. The BlockManager handles application level input and output data, as well as intermediate data within the Map stages. The ShuffleManager handles runtime intermediate results during the shuffle stage.

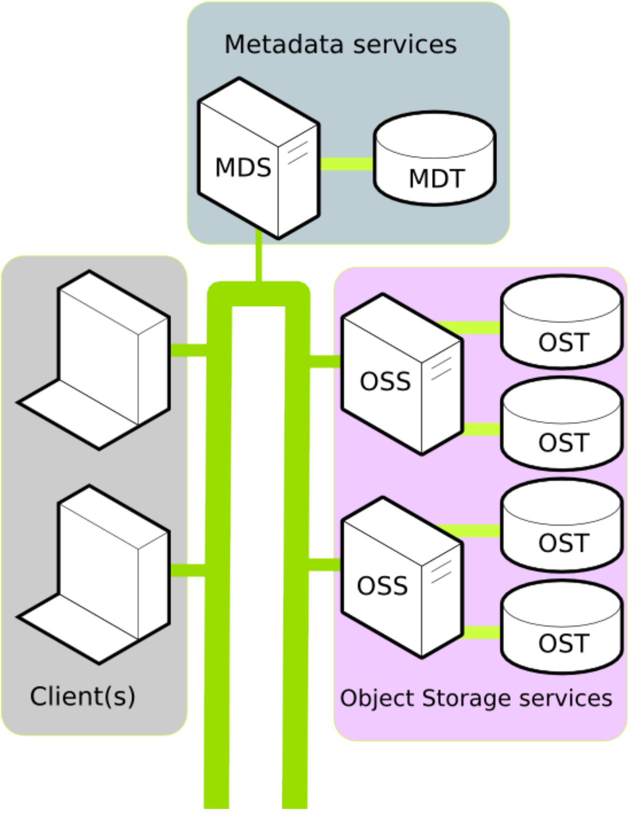


Figure 42. Architecture of the Lustre filesystem. (Courtesy of Intel Wiki.)

Data Objects and Naming: Spark manipulates data with global scope, as well as local scope. Application level data (RDDs) are using a global naming space, intermediate data blocks generated throughout execution have a local scope and naming scheme.

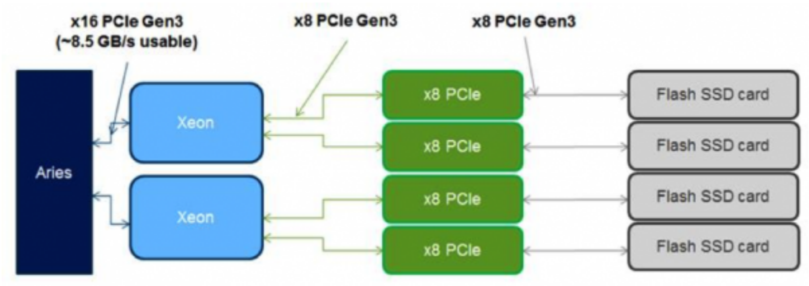


Figure 43. Node architecture of the Cori Burst Buffer. (Courtesy of NERSC.)

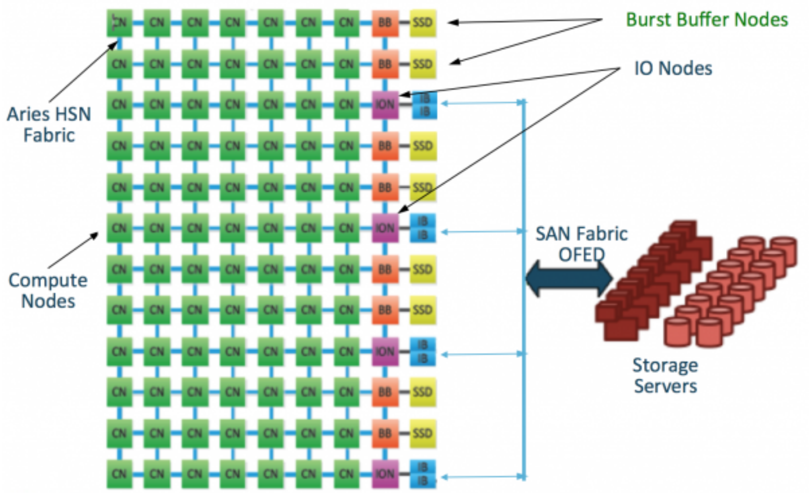


Figure 44. Network topology of the Cori Burst Buffer. (Courtesy of NERSC.)

Objects may exceed the capacity of the physical memory and need to be efficiently moved through the storage hierarchy; the typical challenge when managing naming schemes is mismatch with underlying system architecture. For instance, when global object is distributed (partitioned) across multiple storage spaces a long latency naming service may be needed to locate its physical location. Conversely, any locally named object stored in a physically shared storage may experience undue contention while servicing requests. A current research direction in the Spark community is providing an efficient global naming service, which can reduce network traffic. Note that the global file system in HPC installations provides global naming.

Vertical Data Movement: Vertical data movement refers to the movement through the entire memory hierarchy, including persistent storage. It is needed to move input data blocks into the memory for processing and for storing output data to the persistent storage. To minimize vertical movement for RDDs, Spark allows persisting data in the fast level of memory. As fast memory is capacity constrained, the Spark runtime assigns the task of moving objects across the memory hierarchy to a block manager. Whenever the working set size (input data or intermediate results) exceeds memory capacity, the block manager may trigger vertical data movement. The block manager may also decide to drop a block, in which case its later access may trigger additional vertical data movement for recomputation. Research efforts such as Tachyon [135] aim to reduce expensive (to storage) vertical data movement by replacing it with horizontal (inter-node) data movement. In network-based storage systems, a critical [4, 35] component to the performance of vertical data movement is the file setup stage (communication with the metadata servers).

Horizontal Data Movement - Block Shuffling: The horizontal data movement refers to the shuffle communication phase between compute nodes. Spark assigns the horizontal data movement to the shuffle manager and the block manager. A horizontal data movement request of a block could trigger a vertical data movement because a block may not be resident in memory. Optimizing the performance of horizontal data movement has been the subject of multiple studies [226, 111, 137], in which hardware acceleration such as RDMA is used to reduce the communication cost. The benefit of these techniques is less profound on HPC systems with network-based storage [198] because the performance is dominated by vertical data movement.

System Architecture and Data Movement. Data centers have local storage attached to compute nodes. This enables fast vertical data movement and the number of storage disks scales linearly with the number nodes involved in the computation. Their bandwidth also scale with the number of compute nodes. The archetypal file system for data analytics is the Hadoop Distributed File System (HDFS) which aims to provide both fault tolerance and high throughput access to data. HDFS implements a simple coherency for write-once-read-many file access, which fits well the Spark and Hadoop processing models. In Spark with HDFS, global naming services are implemented in a client-server paradigm. A request is generated for the object owner, subject to the network latency. The owner services it, maybe subject to disk latency (or bandwidth) and the reply is subject to network latency (or bandwidth). Vertical data transfers access the local disk. Horizontal data are subject to network latency/bandwidth, as well as disk latency/bandwidth.

HPC systems use dedicated I/O subsystems, where storage is attached to a “centralized” file system controller. Each and all nodes can see the same amount of storage, and bandwidth to storage is carefully provisioned for the system as a whole. Given that these network file servers are shared between many concurrently scheduled applications, the servers typically optimize for overall system throughput. As such individual applications may observe increase in latency and higher variability. The Lustre [27] architecture, presented in Figure 42 is carefully optimized for throughput and implements a generic many-write-many-read coherency protocol. The installation consists of clients, a Metadata service (MDS) and Object Storage service. The Metadata service contains Metadata Servers, which handle global naming and persistence and the Metadata Targets which provide the actual metadata storage (HDD/SSD). In Spark with Lustre, global naming services access the metadata servers and are subject to network latency and MDS latency. Most existing Lustre installations in production (prior to Lustre 2.6) use a single MDS, only very recent installations [50, 52] use multiple MDSes for improved scalability. Vertical data transfers are served by the Object Storage service, which contains the object Storage Server (OSS) and the Object Storage Target (OST), the HDD/SSD that stores the data. Bandwidth is provisioned in large scale installations by adding additional OSSes.

In our quest to introduce Spark into the HPC community there are two main questions to answer.

1. *How does the differences in architecture between data centers and HPC influence performance?* Previous performance studies of Spark in data center environments [183] indicate that its performance is dominated by the network, through careful optimizations to minimize vertical data movement and maximize the memory resident working set. Ousterhout et al. [172] analyzed the performance of the Big Data Benchmark [217] on 5 Amazon EC2 nodes, for a total of 40 cores, and the TPC-DS benchmark [176] on 20 nodes (160 cores) on EC2. These benchmarks both use Spark SQL [9], which allows SQL queries to be performed over RDDs. By instrumenting the Spark runtime, they were able to attribute time spent in tasks to several factors, including network and disk I/O and computation. They found that, contrary to

popular wisdom about data analytics workflow, that disk I/O is not particularly important: when all work is done on disk, the median speedup from eliminating disk I/O entirely was only 19%, and, more importantly, when all RDDs are persisted to memory, only a 2-5% improvement was achieved from eliminating disk I/O. Upon introduction to HPC systems, we similarly need to understand whether access to storage or network performance dominates within Spark.

2. *What HPC specific features can we exploit to boost Spark performance?* Previous work optimizing data analytics frameworks on HPC systems [137, 111] proposes moving away from the client-server distributed paradigm and exploiting the global file name space already available or Remote Direct Memory Access (RDMA) functionality. Upon introduction to HPC systems, we are interesting in evaluating the potential for performance improvement of adopting such techniques into Spark. Besides providing an initial guide to system researchers, we are also interested in providing configuration guidelines to users and system operators.

We explore these questions using three benchmarks selected to cover the performance space: 1) *BigData Benchmark* uses SparkSQL [9] and stresses vertical data movement; 2) *GroupBy* is a core Spark benchmark designed to capture the worst case scenario for shuffle performance, it stresses both horizontal and vertical data movement; and 3) *PageRank* is an iterative algorithm from GraphX [83] and stresses vertical data movement.

5.4 Experimental Setup

We conducted our experiments on the Edison and Cori Cray XC supercomputers at NERSC [163], and the XSEDE Comet cluster at SDSC. Edison contains 5,576 compute nodes, each with two 2.4 GHz 12-core Intel "Ivy Bridge" processors. Cori contains 2,388 Haswell compute nodes, each with two 2.3 GHz

16-core Intel "Haswell" processors, and 9,688 "Knights Landing" compute nodes, each with one 1.4 GHz Intel Xeon Phi 7250 ("Knights Landing") processor. Each node of Cori is equipped with 128 GB DDR4 2133Mhz MHz memory, and both systems use a Cray Aries interconnect based on the Dragonfly topology.

Comet, installed at SDSC, is Dell cluster consisting of 1,944 compute nodes, each with two 2.5 GHz 12-core Intel "Haswell" processors (Intel Xeon E5-2680 v3). Each node of Comet is equipped with 128 GB DDR4 DRAM. Nodes are connected using InfiniBand FDR. A Lustre filesystem is provided, and additionally each node is equipped with a 320 GB SSD for fast scratch storage.

Cray provides a Cluster Compatibility Mode (CCM) for compute jobs requiring specialized services, such as secure connection, etc. CCM runs Linux and allows an easy path to configure Spark, but imposes limits on the number of nodes per job. More importantly, it disables network transfer mechanisms accelerated by the Aries hardware.

In this study, we ported Spark 1.5.0, 1.6.0, and 2.0 to run on the Cray Extreme Scalability Mode (ESM) to allow better scaling of resources. In ESM, a lightweight kernel runs on the compute nodes and the application has full access to Aries. Spark 1.6 has been subsequently released: as file I/O patterns did not change the optimizations we describe in this chapter remain applicable to it. We use one manager per compute node, based on YARN 2.4.1. This required additional porting efforts to allow TCP-based services. Compared to Spark's standalone scheduler, YARN allows better control of the resources allocated in each node. The Mesos [97] resource manager provides similar control as YARN, but requires administrative privilege. Job admission is done through a resource manager on the front-end node where Spark runs as a YARN client with exclusive access to all resources.

Both Edison and Cori use the Lustre file system. On Edison, the Lustre file system is backed by a single metadata server (MDS) and a single metadata target (MDT) per file system. On Cori, a master MDS is assisted by a 4 additional Distributed Namespace (DNE) MDSes. The DNEs do not yet support full functionality, and for all Spark concerns Cori performs as a single MDS system.

On Cori we also evaluate a layer of non-volatile storage (`BurstBuffer`) that sits between the processors' memory and the parallel file system, specifically designed to accelerate I/O performance. The NERSC hardware is based on Cray DataWarp and presented in Figures 43 and 44. The flash memory for Cray DataWarp is attached to Burst Buffer nodes that are packaged two nodes to a blade. Each Burst Buffer node contains a Xeon processor 64 GB of DDR3 memory, and two 3.2 TB NAND flash SSD modules attached over two PCIe gen3 x8 interfaces. Each Burst Buffer node is attached to a Cray Aries network interconnect over a PCIe gen3 x16 interface. Each Burst Buffer node provides approximately 6.4 TB of usable capacity and a peak of approximately 5.7 GB/sec of sequential read and write bandwidth. The `BurstBuffer` nodes can be accessed from the compute nodes in *private* mode and in *striped* mode. Ours is the first evaluation on such technology at scale. However, since the hardware is new and not tuned yet for production, the `BurstBuffer` results are only indications of its potential and it features; we expect them to evolve and improve.

We evaluate *BigData Benchmark*, *GroupBy* and *PageRank* in both weak and strong scaling experiments. Together they provide good coverage of the important performance factors in Spark. *BigData Benchmark* has inputs up to five nodes and we'll concentrate the node level performance discussion around it. *GroupBy* scales and we evaluate it up to 10,240 cores. For *PageRank* we have only small inputs available and evaluate it only up to 8 nodes. Each benchmark has been executed at least five times and we report mean performance. Some `BurstBuffer` experiments were very noisy and we report only the best performance.

5.5 Single Node Performance

To calibrate initial performance, we evaluated a single node of Cori and Edison against a local workstation with fast SSDs: eight 3.5GHz Xeon i7-3770K cores with 1TB fast SSD. Figure 45 shows the performance of queries 1-3 of the Big Data Benchmark [217] using both on-disk and in-memory modes. The results are quite similar on Edison and Cori. As shown, a single node of Edison when running with eight cores and accessing the file system is roughly twice as slow than the workstation. When data is preloaded in memory, eight cores of Edison match the workstation performance; this is expected as the workstation contains server grade CPUs. When scaling up the Edison node and using all 24 cores, performance is still 50% slower than the workstation. This slowdown is entirely attributed to the file system; performance scales with cores when running with data preloaded in memory, as illustrated when comparing eight cores with the full node performance.

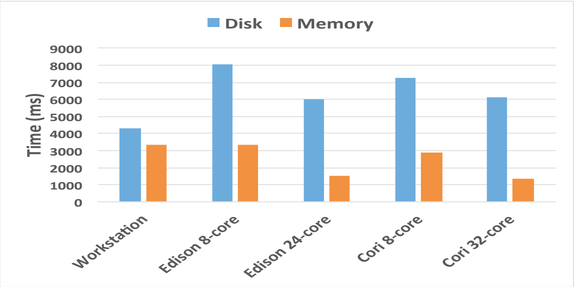


Figure 45. BigData Benchmark performance on workstation and a single node of Edison and Cori. Input data is pre-cached in memory or read from disk.

To quantify the difference in I/O performance, we instrumented the Hadoop LocalFileSystem interface used by Spark to record the number of calls and the time spent in `open`, `read`, `write`, and `close` file operations. The time spent in `read`, `write`, and `close` operations did not significantly differ between the systems, while file `open` operations were *much* slower, as shown in Figure 46. On the workstation the mean file open time was 23 μs; on Edison it was 542 μs, almost 24 times greater. Some file open operations on Edison took an extreme amount of time to complete: in the worst case observed, a single file open operation took 324 *ms*.

The Big Data Benchmark illustrates the application level I/O bottlenecks. At this stage, the number of open operations is linear in the number of partitions. The dataset for Query 1 consists of a single directory containing one data file per partition in Parquet format: there are 3,977 partitions/files. Each file is accompanied by a checksum file used to verify data integrity. These all must be opened, so a minimum of 7,954 file opens must occur to run Query 1. The data format readers are designed to operate in series in a state-free manner. In the first step, the data and checksum files are opened and read, the checksums are calculated are compared, and the data and checksum files are closed, completing the first task. Then, each partition file is opened and the footer, containing column metadata, is read, and the partition file is closed, completing the second task. Finally, the partition file is opened again, the column values are read, and the partition file is closed again, for a total for four file opens per partition, or 15,908 file opens.

5.6 Scaling Concerns

On a data center system architecture with local disks, one does not expect file open (or create) time to have a large effect on the overall time to job completion. Thus, Spark and the associated domain libraries implement stateless operation for resilience and elastic parallelism purposes by opening and closing the files involved in each individual data access: *file metadata operations are a scalability bottleneck on our HPC*

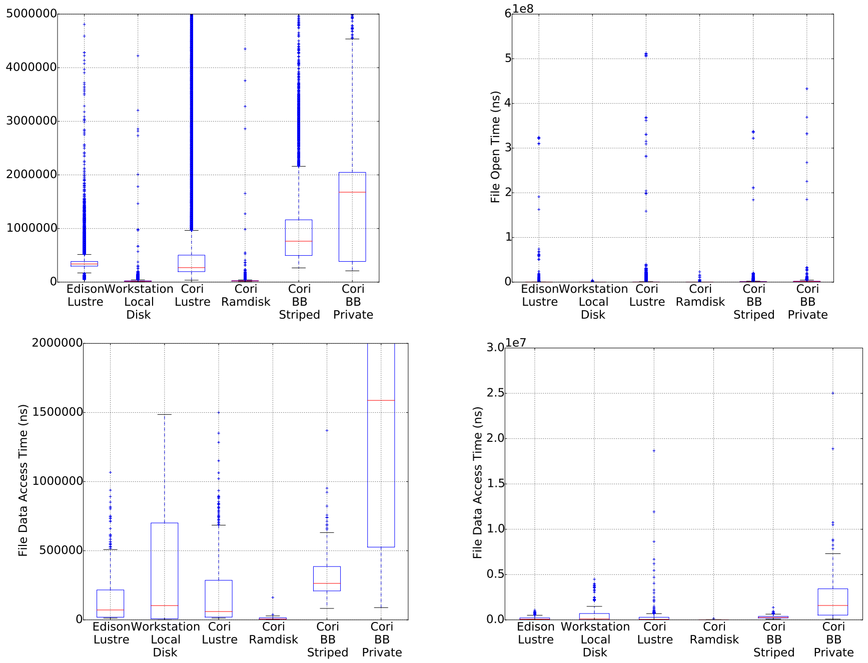


Figure 46. Distribution of file I/O on the Lustre filesystem vs. a workstation with ext4 local disk, during the execution of Big Data. Left, median file open time is 24× higher on Lustre. Second, range of file open time, ≈ 14, 000× larger on Lustre. Third, median of file read time for all BigData reads – latency similar between workstation and Lustre. Right, range of file open time – Lustre exhibits much larger variability than workstation.

systems. Any effort scaling Spark up and out on an HPC installation has first to address this concern.

There are several Spark configuration alternatives that affect file I/O behavior. We were first interested to determine if the larger number of cores in a HPC node allows for a degree of oversubscription (partitions per core) high enough to hide the MDS latency. We have systematically explored consolidation, speculation, varying the number of partitions and data block sizes to no avail.

In the time honed HPC tradition, one solution is to throw bigger and better hardware at the problem. The first aspect is to exploit the higher core concurrency present in HPC systems. As the previous Section shows, increasing[1] the number of cores per node does improve performance, but not enough to mitigate the effects of the file system.

For the Lustre installations evaluated, metadata performance is determined by the MDS hardware configuration. Although Cori contains multiple MDSes, the current Lustre 2.6 version does not exploit them well[2] and performance for the Spark workload is identical to that of a single MDS. When comparing Cori with Edison, the former contains newer hardware and exhibits lower metadata access latency (median 270μs on Cori vs 338μs on Edison), still when using the full node (32 and 24 cores) both are at best 50% slower than a eight core workstation. Enabling multiple MDSes will improve scalability but not the latency of an operation [52], thus over-provisioning the Lustre metadata service is unlikely to provide satisfactory per node performance.

A third hardware solution is provided by the `BurstBuffer` I/O subsystem installed in Cori. This large NVRAM array situated close to the CPU is designed to

---

[1] Cori Phase II will contain Intel Xeon Phi nodes with up to 256 cores per node. This will become available circa Oct 2016 to early users.

[2] Supports a restricted set of operations that are not frequent in Spark.

improve throughput for small I/O operations and for pre-staging of data. The question still remains if it is well suited for the access patterns performed by Spark.

Besides hardware, software techniques can alleviate some of the metadata performance bottlenecks. The first and most obvious solution is to use a memory mapped file system (e.g. `/dev/shm`) as the secondary storage target for Spark. Subject to physical memory constraints, this will eliminate a large fraction of the traffic to the back-end storage system. In the rest of this chapter, we will refer to this configuration as `ramdisk`. Note that this is a user level technique and there are several limitations: 1) the job crashes when memory is exhausted; and 2) since data is not written to disk it does not provide any resilience and persistence guarantees.

HPC applications run in-memory so it may seem that `ramdisk` provides a solution. For medium to large problems and long running iterative algorithms Spark will fail during execution when using `ramdisk`, due to lax garbage collection in the block and shuffle managers. To accommodate large problems we evaluate a configuration where a local file system is mounted and backed by a Lustre file, referred to as `lustremount`. This requires administrative privilege on the systems and due to operational concerns we were initially granted access to only one node. Based on the results of this study, this capability was added to Shifter [114], which is NERSC developed software that enables Docker containers to be run on shared HPC systems.

To understand scaling with large problems we develop a software caching layer for the file system metadata, described in Section 5.6. In the rest of this chapter we refer to this configuration as `filepool`. This is a user level approach orthogonal to the solutions that mount a local file system. Since data is stored on Lustre, `filepool` provides resilience and persistence guarantees.

I/O Scaling in Spark. I/O overhead occurs due to metadata operations, as well as proper data access read/write operations. All these operations occur in both the application level I/O, as well as inside Spark for memory constrained problems or during the shuffle stage.

In Section 5.5 we have illustrated the impact of `fopen` metadata operations on the performance of *BigData Benchmark*.. There, the benchmark performed during the application input stage a number of open operations linear in the number of partitions $O(partitions)$. Big Data Benchmark did not involve a large amount of shuffle data.

Because Spark allows partitions to be cached in memory, slow reading of the initial data is not necessarily problematic, particularly in an interactive session in which multiple queries are being performed against the same data. Assuming that the working set fits in memory, disk access for input data can be avoided except for the first query. In this case, the `BurstBuffer` can be also used for data pre-staging.

In Figure 47 we show the scalability of the GroupBy benchmark up to 16 nodes (384 cores) on Edison for a weak scaling experiment where the problem is chosen small enough to fit entirely in memory.

GroupBy measures worst-case shuffle performance: a wide shuffle in which every partition must exchange data with every other partition. The benchmark generates key-value pairs locally within each partition and then performs a shuffle to consolidate the values for each key. The shuffle process has two parts: in the first (map) part, each node sorts the data by key and writes the data for each partition to a partition-specific file. This is the *local* task prior to the stage boundary in Figure 40. In the second (reduce) part, each node reads locally-available data from the locally-written shuffle files and issues network requests for non-local data. This is the *global* task after the stage boundary in Figure 40.

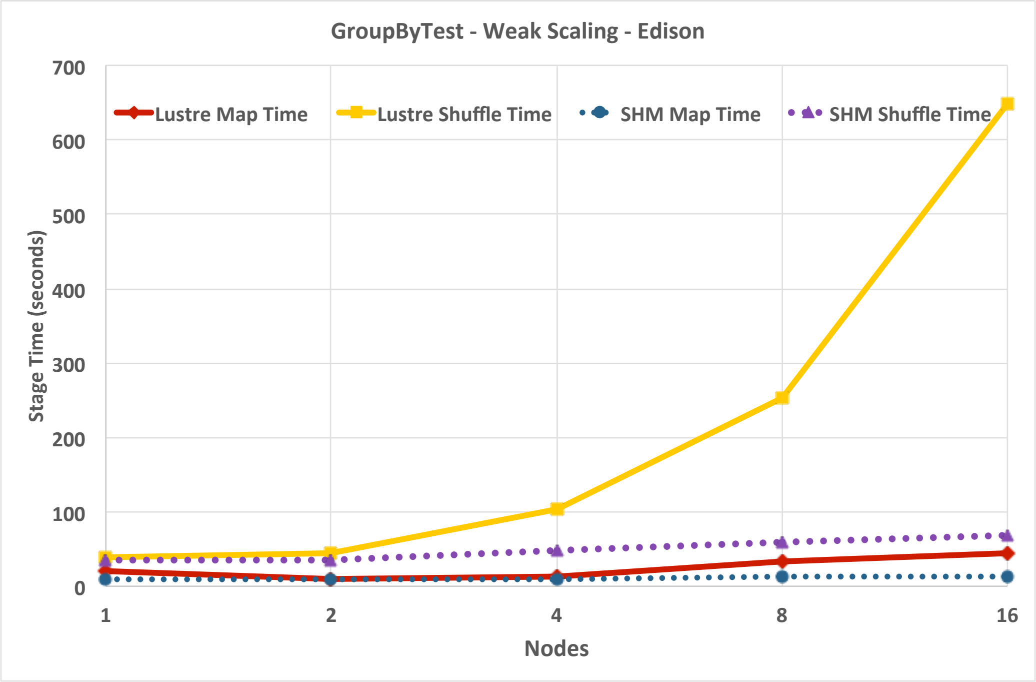


Figure 47. Time for the map and reduce phases of GroupBy on Edison for Lustre and `ramdisk` as we use additional nodes to process larger datasets (weak scaling).

When running entirely in memory (`ramdisk`) performance scales with nodes, while scalability is poor when using Lustre. As illustrated, the Map phase scales on Lustre, while the Shuffle phase does not. For reference, on the workstation, mean task duration is 1,618 ms for `ramdisk` and 1,636 ms for local disk. On the Edison node, mean task duration was 1,540 ms for `ramdisk` and 3,228 ms for Lustre.

We instrumented Spark's Shuffle Manager component to track file I/O operations. During the write phase of the shuffle, a shuffle file is created for each partition, and each shuffle file is written to as many times as there are partitions. An index file is also written, which contains a map from keys to a shuffle file and offset. During the read phase, for each local partition to read and each remote request received, the index file is opened, data is read to locate the appropriate shuffle data file, which is then opened, read, and closed. The number of file open operations during the shuffle is quadratic in the number of partitions $O(partitions^2)$.

To enable load balancing, the Spark documentation suggests a default number of partitions as 4x the number of cores. On 16 nodes of Edison, with a total of 384 cores, then, we have 1,536 partitions, giving us 1,536 shuffle data files, each of which is opened 1,536 times during the write phase and another 1,536 times during the read phase, resulting in 4,718,592 file open. Not only is the number of file opens is quadratic in partitions, but the cost *per* file open also grows as we add nodes, as shown in Figure 48.

As the number of file I/O operations is linear with the number of partitions/cores during the application I/O and quadratic during the shuffle stage, in the rest of chapter we concentrate the evaluation on the shuffle stage.

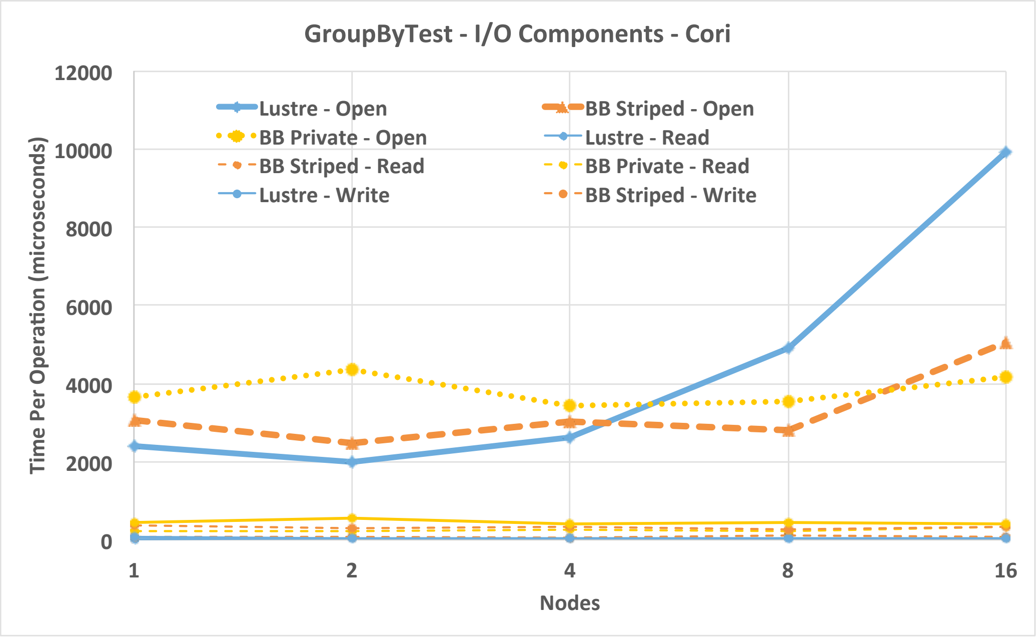


Figure 48. *Average time for a* `open`, `read`, *and* `write` *operation performed during the GroupBy execution with weak scaling on Cori.*

As for each read/write operation Spark will perform a file open, the performance ratio of these operations is an indicator of scalability. Figure 49 shows the performance penalty incurred by repeatedly opening a file, performing one read of the indicated size, and closing the file, versus opening the file once, performing many reads of the indicated size, and closing the file. Using many open-read-close cycles on a workstation with a local disk is 6× slower for 1 KB reads than opening once and performing many reads, while on Edison with Lustre, many open-read-close cycles is 56× slower than opening once and reading many times. Lustre on Cori is similar, while the Burst Buffers in striped mode reduce the penalty to as low as 16×. All of the filesystems available on our HPC systems incur a substantial penalty from open-per-read.

The scalability problems caused by the large number of file opens are exacerbated by the potentially small size of each read. Many data analytics applications have a structure in which many keys are associated with a small number of values. For example in PageRank, most write operations are smaller than 1KB. This reflects the structure of the data, as most websites have few incoming links. The data is structures as key-value pairs with a site's URL as the key and a list of incoming links as the value, so most values are short.

Improving Metadata Performance With File Pooling. For problems small enough to fit in the main memory, the `ramdisk` Spark configuration scales. However, in our experiments many large problems ran out of memory at runtime, particularly iterative algorithms where the block garbage collection inside the shuffle manager is not aggressive.

In order to accommodate large problems at scale we have simply chosen to add a layer for pooling and caching open file descriptors within Spark. All tasks

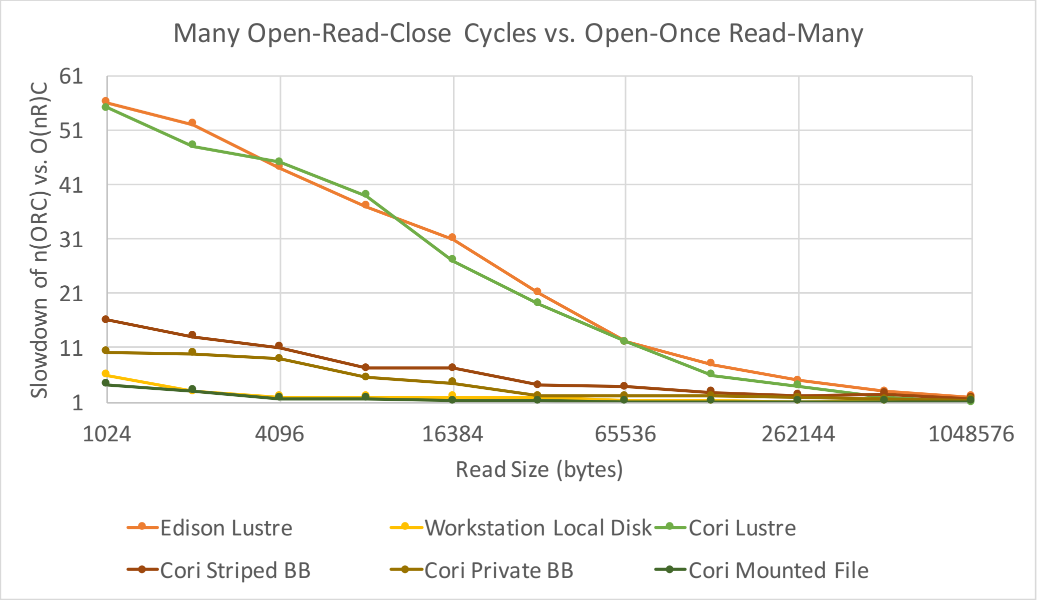


Figure 49. Performance improvements from amortizing the cost of file opens. We compare one read per open with 100,000 reads per open.

within an Executor (node) share a descriptor pool. We redefine `FileInputStream` and `FileOutputStream` to access the pool for open and close operations. Once a file is opened, subsequent close operations are ignored and the descriptor is cached in the pool. For any subsequent opens, if the descriptor is available we simply pass it to the application. To facilitate multiple readers, if a file is requested while being used by another task, we simply reopen it and insert it into the pool.

This descriptor cache is subject to capacity constraints as there are limits on the number of `Inodes` within the node OS image, as well as site-wide Lustre limits on the number of files open for a given job. In the current implementation, each Executor is assigned its proportional number of entries subject to these constraints.

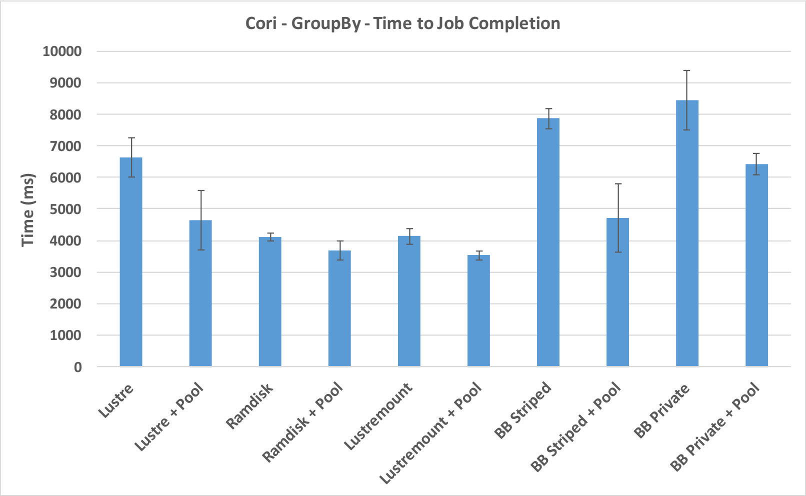
We evaluated a statically sized file pool using two eviction policies to solve capacity conflicts: LIFO and FIFO. For brevity we omit detailed results and note that LIFO provides best performance for the shuffle stage. As results indicate, this simple implementation enables Spark to scale.

Further refinements are certainly possible. Application I/O files can be easily distinguished from intermediate shuffle files and can be allocated from a smaller pool, using FIFO. Within the shuffle, we can tailor the eviction policy based on the shuffle manager behavior, e.g. when a block is dropped from memory the files included in its lineage are likely to be accessed together in time during recomputation.

Running out of `Inodes` aborts execution so in our implementation a task blocks when trying to open a file and the pool descriptor is filled at capacity. As this can lead to livelock, we have audited the Spark implementation and confirmed with traces that the implementation paradigm is to open a single file at a time, so livelock cannot occur.

Impact of Metadata Access Latency on Scalability. In Figure 50 we show the single node performance on Cori in all configurations. As shown, using the back-end Lustre file system is the slowest, by as much as 7× when compared to the best configuration. Both file system configurations improve performance significantly by reducing the overhead of calls to open files: `ramdisk` is up to ≈ 7.7× faster and `lustremount` is ≈ 6.6× faster than Lustre.

`filepool` also improves performance in all cases. It is ≈ 2.2× faster than Lustre, and interestingly enough is speeds up the other two configurations. For example, for *GroupBy* where each task performs $O(partitions^2)$ file opens, adding pooling to the "local" file system (e.g. `ramdisk+filepool`) improves performance by ≈ 15%. The performance improvements are attributed to the lower number of `open` system calls. For *PageRank* and *BigData Benchmark* the improvements are a more modest 1% and 2% respectively. As it never degraded performance, this argues for running in configurations where our `filepool` implementation itself or a user level file system is interposed between Spark and any other "local" file systems used for shuffle data management.



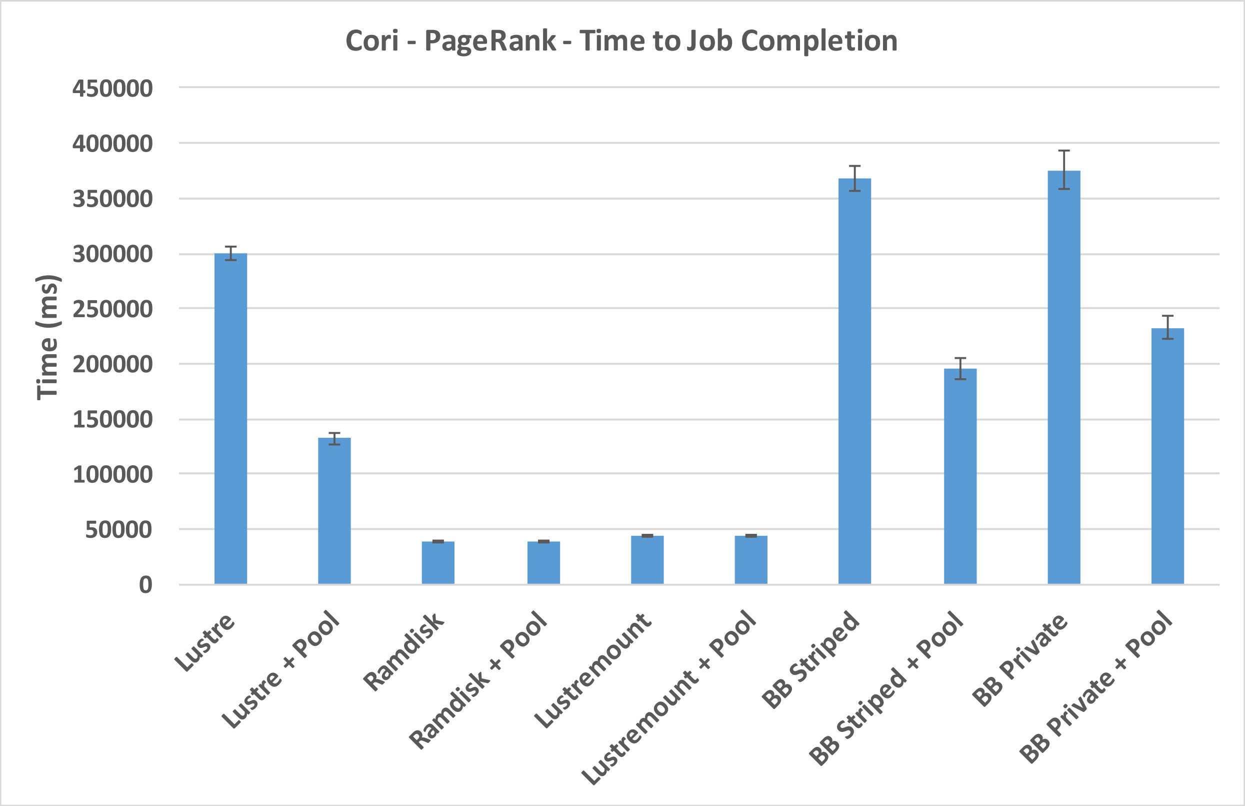


Figure 50. *GroupBy* and *PageRank* performance on a single node of Cori.

For all configurations the performance improvements are proportional to the number of file opens during the shuffle stage: *GroupBy* is quadratic in partitions while in *PageRank* it is a function of the graph structure.

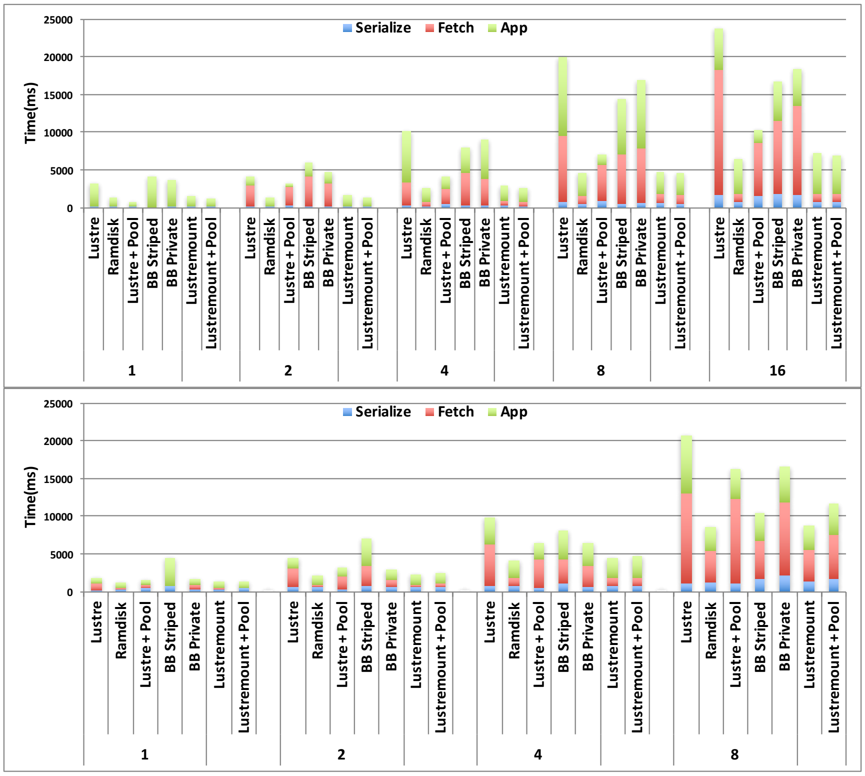


Figure 51. GroupBy weak scaling on Cori up to 8 nodes (256 cores). Top: with YARN. Bottom: with the Spark standalone scheduler.

In Figure 51 we show the scalability of *GroupBy* up to eight nodes (256 cores). We present the average task time and within it, distinguish between time spent in serialization (Serialization), disk access together with network access (Fetch) and application level computation (App). `ramdisk` is fastest, up to 6× when compared to Lustre. `filepool` is slower than `ramdisk`, but still significantly faster than Lustre, up to 4×. The performance differences between `ramdisk` and `filepool` increase with the scale: while system call overhead is constant, metadata latency performance degrades. When combining `filepool` with `lustremount` we observe performance improvements ranging from 17% on one node to 2% on 16 nodes.

In Figure 52 we present scalability for *PageRank* (left) and *BigData Benchmark* (right). As mentioned, the inputs for these benchmarks are not very large and we scale

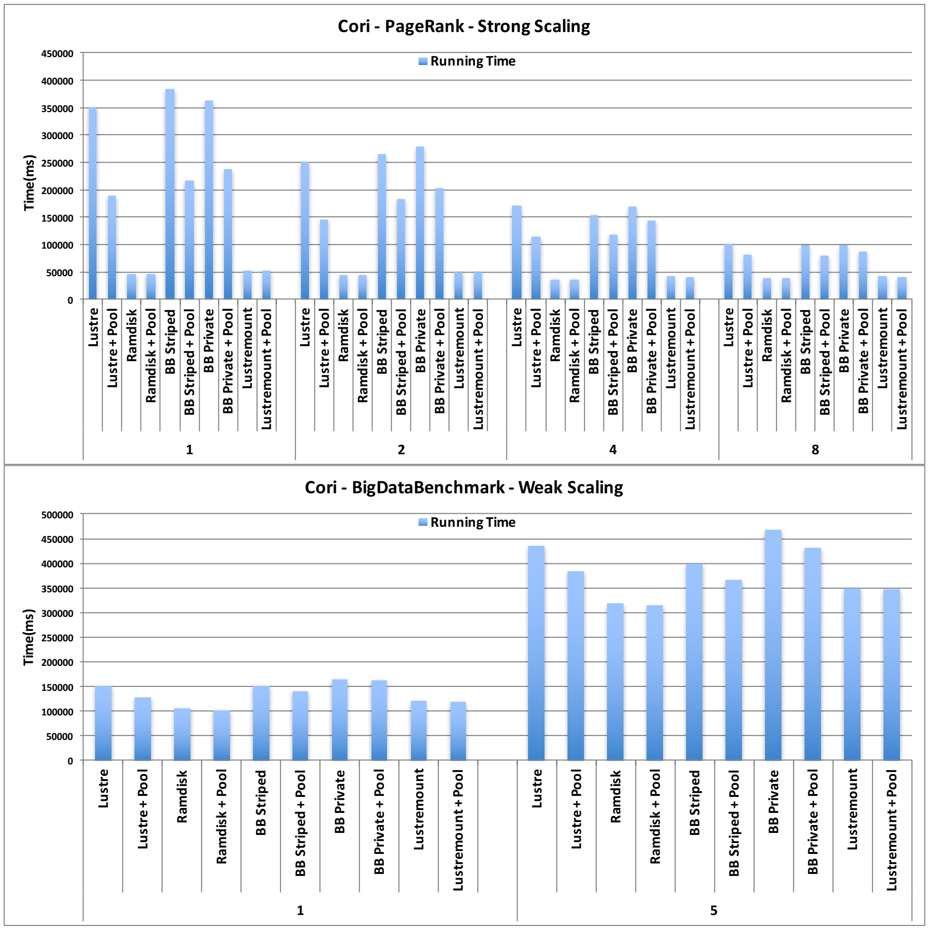


Figure 52. PageRank and *BigData Benchmark* scaling on Cori, up to 8 nodes (256 cores).

up to 8 nodes. The trends for *PageRank* are similar to *GroupBy* and we observe very good performance improvements from `filepool` and `ramdisk`. The improvements from combining pooling with `ramdisk` are up to 3%. In addition, when strong scaling *PageRank* the performance of `ramdisk` improves only slightly with scale (up to 25%), while configurations that touch the file system (Lustre and `BurstBuffer`) improve by as much as 3.5×. The gains are explained by better parallelism in the read/write operations during shuffle.

The performance of *BigData Benchmark* is least affected by any of our optimizations. This is because behavior is dominated by the initial application level

I/O stage, which we did not optimize. This is the case where `ramdisk` helps the least and further performance improvements can be had only by applying the file pooling optimization or `lustremount`. *BigData Benchmark* illustrates the fact that any optimizations have to address in shuffle in conjunction with the application level I/O.

When using the Yarn resource manager we could not effectively scale Spark up to more than 16 nodes on either Edison or Cori. The application runs but executors are very late in joining the job and repeatedly disappear during execution. Thus the execution while reserving the initially requested number of nodes, proceeds on far fewer. After exhausting timeout configuration parameters, we are still investigating the cause.

For larger scale experiments we had to use the Spark standalone scheduler, results presented in Figure 51 right. While Yarn runs one executor (process) per node, the Spark manager runs one executor per core. The Lustre configuration stops scaling at 512 cores. The standalone scheduler limits the the performance impact of our file pooling technique: with Yarn we provide a per node cache while with the standalone scheduler we provide a per core cache. This is reflected in the results: while with YARN `filepool` scales similarly to `ramdisk`, it now scales similarly to Lustre and we observe speedup only as high as 30%. Note that `filepool` can be reimplemented for the standalone scheduler, in which case we expect it to behave again like `ramdisk`.

As illustrated in Figure 53 we successfully (weak) scaled `ramdisk` up to 10,240 cores. Lustre does not scale past 20 nodes, where we start observing failures and job timeouts. When running on the `BurstBuffer` we observe scalability up 80 nodes (2,560 cores), after which jobs abort. Note that `BurstBuffer` performance is highly variable at scale and we report the best performance observed across all experiments.

Figure 54 compares Lustre, `ramdisk` and `lustremount`. To use `lustremount` on more than one node, we run Spark inside a Shifter user-defined image. With Shifter, each node mounts a single image containing JVM and Spark installations in read-only mode and a per-node read/write loopback file system. Because the JVM and Spark are stored on a file-backed filesystem in Shifter, file opens required to load shared libraries, Java class files, and Spark configuration files are also offloaded from the metadata server, improving performance over configurations where Spark is installed on the Lustre filesystem. Identically configured GroupBy benchmarks running on `ramdisk` with Spark running in Shifter is up to 16% faster than than with Spark itself installed on Lustre. In addition, since the mount is private to a single node, the kernel buffer cache and directory entry cache can safely cache metadata blocks and directory entries. This can significantly reduce the number of metadata operations and improves performance for small I/O operations. For the `lustremount` implementation in Shifter initializes a sparse file in the Lustre file system for each node in the Spark cluster. These files are then formatted as XFS file systems and mounted as a loop back mount during job launch. Unlike using `ramdisk`, the `lustremount` approach is not limited to the memory size of the node and it doesn't take away memory resources from the application. Using `lustremount` we can scale up to 10,240 cores, with time to completion only 13% slower than `ramdisk` at 10,240 cores.

Impact of `BurstBuffer` on Scalability. The `BurstBuffer` hardware provides two operating modes, private where files are stored on a single blade (device) and striped where files are stored across multiple blades.

In Figure 46 we present the metadata latency and read operations latency for a single node run of *BigData Benchmark*. As illustrated, the mean time per operation when using the `BurstBuffer` is higher than the back-end Lustre in both striped

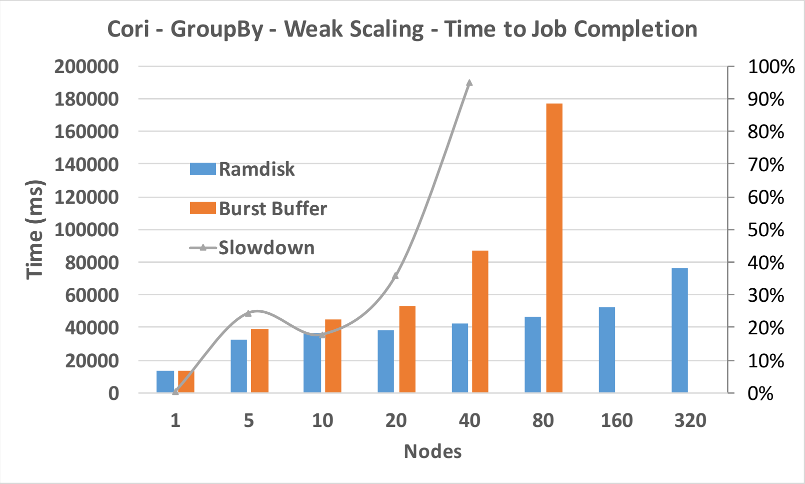


Figure 53. *GroupBy* at large scale on Cori, up to 320 nodes (10,240 cores). Standalone scheduler. Series "Slowdown" shows the slowdown of `BurstBuffer` against `ramdisk`, plotted using the secondary right axis.

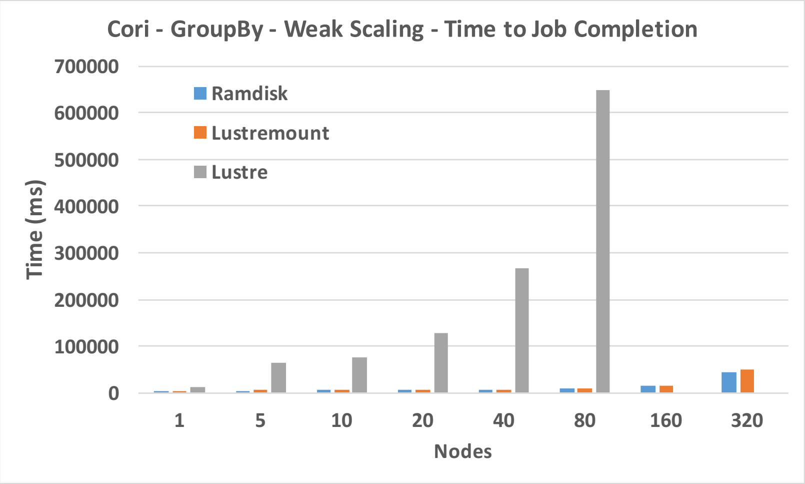


Figure 54. *GroupBy* at large scale on Cori, up to 320 nodes (10,240 cores). Standalone scheduler. Lustre, `ramdisk`, and `lustremount`.

and private mode. This is expected as interposing the `BurstBuffer` layer between processors and Lustre can only increase latency. On the other hand the variance is reduced 5× compared to Lustre. When comparing striped mode with the private mode for *BigData Benchmark* striped exhibits 15% lower variance than private.

Higher latency per operation affects performance at small scale and Spark single node performance with `BurstBuffer` is slightly worse than going directly to Lustre. On the other hand, lower variability translates directly in better scaling as illustrated in Figures 48 and 51. Up to 40 nodes (1,280 cores) `BurstBuffer` provides performance comparable to running in memory with `ramdisk`. As expected, the configuration with lower variability (striped) exhibits better scalability than private mode. This is a direct illustration of the need to optimize for the tail latency at scale.

5.7 Improving Shuffle Scalability With Better Block Management

Even when running using a good configuration available, e.g. `filepool+ramdisk`, some algorithms may not scale due to the memory management within the shuffle manager, which introduces excessive vertical data movement. The behavior of the PageRank algorithm illustrates this.

In Figure 55 left we show the evolution of the algorithm for a problem that fits entirely in main memory on one node of Edison. We plot both memory usage and the duration of an iteration over the execution. As shown, execution proceeds at a steady rate in both memory and time. On the right hand side of the figure, we plot the evolution of the algorithm when the working set does not fit in the main memory. As illustrated, each iteration becomes progressively slower and each iteration takes double the amount of its predecessor. The same behavior is observed on the workstation, albeit less severe.

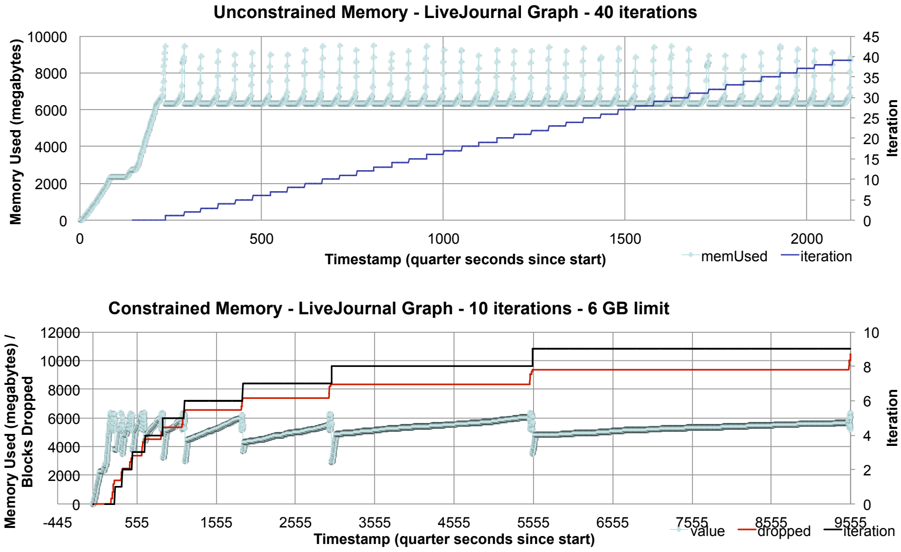


Figure 55. PageRank performance on a single node of Edison. The amount of memory used during execution is plotted against the right hand side axis. The time taken by each iteration is plotted against the left hand side axis. Execution under constrained memory resources slows down with the number of iterations.

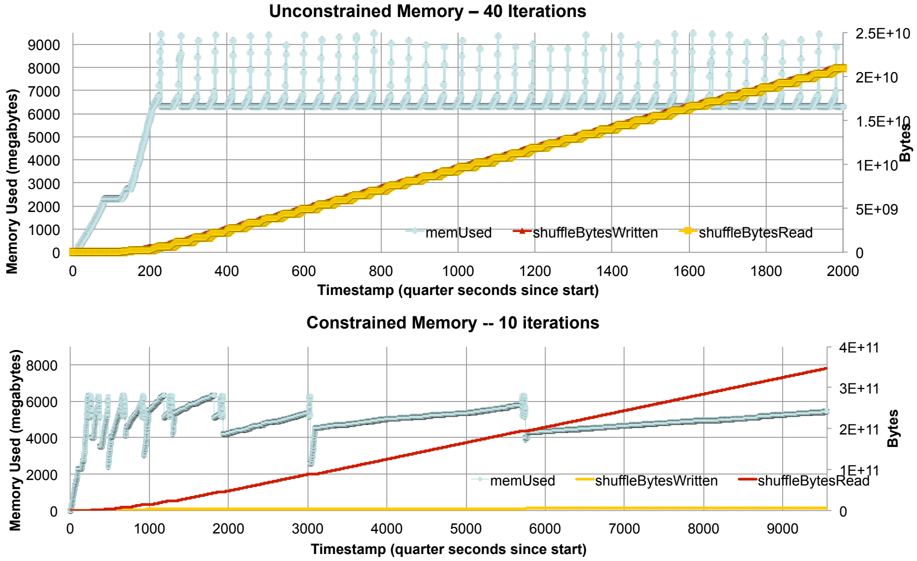


Figure 56. PageRank I/O behavior on a single node of Edison. The amount of memory used during execution is plotted against the right hand side axis. The amount of bytes read and written from disk is plotted against the left hand side axis. While memory usage stays constant, the amount of bytes read explodes under constrained memory resources.

After investigation using the instrumentation framework already developed, we observed that during constrained execution the amount of data read from disk grows at a rate two orders of magnitude higher than during unconstrained execution. After further investigation, we attributed the root cause of the problem to the shuffle block manager. Whenever running out of memory, the block manager evicts the least recently used block. The first subsequent access to the evicted block triggers recomputation, which evicts another block needed for the partial solution which in turn triggers recomputation and eviction of blocks needed. This results in orders of magnitude increases in vertical data movement, as illustrated in Figure 56.

This behavior affects the scaling of iterative algorithms on all systems and it should be fixed. In the data center it is less pronounced as local disks are better at latency. As shown, it is very pronounced on our HPC systems. One lesson here is that because storage behaves differently, in particular for small requests, there exists incentive to specifically tune the shuffle block manager for HPC.

For the PageRank algorithm we have actually an algorithmic fix which involves marking as persistent the intermediate result RDDs from each iteration. This causes Spark to write them to the back-end storage. Upon eviction, a persistent block is read from storage instead of being recomputed. Figure 57 shows the performance of the fixed PageRank algorithm and we observe performance improvements as high as 11×. Note that all the performance findings in this chapter are reported on this fixed algorithm. The original GraphX implementation does not scale beyond a single node on our systems.

There are two possible generic solutions to this problem. First, we could implement a system which tracks how often shuffle data must be reread from disk and automatically persist partitions that depend on that data when a threshold is exceeded.

Second, we could track the cost of recomputing and rereading the lineage of an RDD and, rather than evicting on a least-recently-used basis, instead evict the block which will have the lowest recompute or reread cost. We have implemented the first of these solutions: if the same shuffle block is read three times, the corresponding map output is marked to be persisted to disk immediately before the third read.

5.8 Spark-Perf Benchmark on Lustre

Previous work on porting Spark to the Cray platform [151] running under Cluster Compatibility Mode revealed that performance of TeraSort and PageRank was up to four times worse on a 43 nodes of a Cray XC system compared to an experimental 43-node Cray Aries-based system with local SSDs, even though the experimental system had fewer cores than the Cray XC (1,032 vs 1,376). To mitigate this problem, the authors redirected shuffle intermediate files to an in-memory filesystem, but noted that this limited the size of problem that could be solved, and that the entire Spark job fails if the in-memory filesystem becomes full. Multiple shuffle storage directories can be specified, one using the in-memory filesystem and one using the Lustre scratch filesystem, but the Spark runtime then uses them in a round-robin manner, so performance is still degraded.

On Cori we compare directly Lustre with in-memory execution performance. On Comet we compare Lustre with SSD storage. To illustrate the main differences we use the GroupBy benchmark which is a worst-case shuffle. GroupBy generates key-value pairs with a limited number of keys across many partitions, and then groups all values associated with a particular key into one partition. This requires all-to-all communication, and thus maximizes the number of shuffle file operations required, as described in Section 5.6, above.

Figure 58 shows the results on Cori. On a single node (32 cores), when shuffle intermediate files are stored on Lustre, time to job completion is 6 times longer than when shuffle intermediate files are stored on an in-memory filesystem. The performance degradation increases as nodes are added: at 80 nodes, performance is 61 times worse on Lustre than the in-memory filesystem. Runs larger than 80 nodes using Lustre fail.

Results on Comet are shown in Figure 59. On one node, shuffle performance is 11 times slower on Lustre than on the SSD; however, the performance penalty does not become worse as we add nodes. Because Comet compute nodes feature local SSDs, there is less contention for the Lustre metadata server, as other jobs running on the system tend to make use of the SSD for intermediate file storage.

Figure 60 shows the performance of the spark-perf benchmarks [197] on SDSC Comet. The *scheduling-throughput* benchmark runs a series of empty tasks without any disk I/O; its performance is unaffected by the choice of shuffle data directory. The *scala-agg-by-key*, *scala-agg-by-key-int* and *scala-agg-by-key-naive* benchmarks perform aggregation by key: they generate key-value pairs and then apply functions to all values associated with the same key throughout the RDD; this requires a shuffle to move data between partitions. The version using floating point values (*scala-agg-by-key*) and the integer version (*scala-agg-by-key-int*) are designed to shuffle the same number of bytes of data, so that the number of values in the integer version is larger than for the floating point version, increasing the number of shuffle intermediate file writes. The *scala-agg-by-key-naive* benchmark first performs a groupByKey, grouping all values for each key into one partition, before performing partition-local reductions, so that shuffles move a larger volume of data than for the non-naive versions, giving larger shuffle writes. The three *scala-agg-by-key* benchmarks have degraded performance when intermediate data is stored on Lustre, which continue to degrade as more nodes are added; at 16 nodes, performance for *scala-agg-by-key-naive* is 12 times worse than on SSD. The remaining benchmarks involve little or no shuffling and so are unaffected by shuffle directory placement.

As described in Section 5.6, shuffle intermediate files are opened once for each read or write. When shuffle intermediate files are stored on Lustre, this causes heavy metadata server load which slows the overall process of reading or writing. Figure 61 shows the slowdown that results from opening a file, reading it, closing it, and repeating this process, as compared to opening a file once and performing multiple reads. For read sizes under one megabyte, Lustre filesystems show a penalty increasing with decreasing read size.

Spark-perf also provides a set of machine learning benchmarks implemented using MLLib [157]. Figure 62 shows the slowdown of using Lustre storage instead of SSD for these benchmarks. Iterative algorithms – those which perform the same stages multiple times, and therefore have multiple rounds of shuffling – show the worst slowdown. The *lda* (Latent Dirichlet allocation), *pic* (power iteration clustering), summary statistics, *spearman* (Spearman rank correlation) and *prefix-span* (Prefix Span sequential pattern mining) benchmarks all show substantial slowdown when shuffle files are stored on Lustre rather than local SSDs. These are all iterative with the exception of the summary statistics benchmark, which has smaller block sizes than the other benchmarks.

These results demonstrate that shuffle performance is a major cause of performance degradation when local disk is not available or not used for shuffle-heavy applications.

5.9 Localizing Metadata Operations with Shifter

To improve the file IO performance, ideally we need to avoid propagating metadata operations to the Lustre filesystem because these files are used solely by individual compute nodes. On Cray XC systems, we do not have access to local disk, and using in-memory filesystems limits the problem sizes. Our file pooling technique, described above, maintains a pool of open file handles during shuffling to avoid repeated opens of the same file. However, this requires modifications to the Spark runtime, and affects only operations coming from the Spark runtime. Other sources of redundant opens, such as high-level libraries and third-party file format readers, are not addressed. Furthermore, each file must be opened at least once, still placing load on the Lustre metadata server, even though the files are only needed on one node.

To keep metadata operations local, we have previously experimented with mounting a per-node loopback filesystem, each backed by a file stored on Lustre. This enables storage larger than available through an in-memory filesystem while still keeping file opens of intermediate files local; only a single open operation per node must be sent to the Lustre metadata server, to open the backing file. This approach was not feasible, however, for ordinary use, as mounting a loopback filesystem requires root privileges.

Shifter [114] is a lightweight container infrastructure for the Cray environment that provides Docker-like functionality. With Shifter, the user can, when scheduling an interactive or batch job, specify a Docker image, which will be made available on each of the compute nodes. In order to do this, Shifter provides a mechanism for mounting the image, stored on Lustre, as a read-only loopback filesystem on each compute node within the job. Motivated by our work, Shifter was recently extended to optionally allow a per-compute-node image to be mounted as a read-write loopback filesystem.

Using mounted files eliminates the penalty for per-read opens, as shown in Figure 61. When we run the GroupBy benchmark on Cori with data stored in a per-node loopback filesystem, we vastly improve scaling behavior, and performance at 10,240 cores is only 1.6× slower than in-memory filesystem, as shown in Figure 64. Unlike with the in-memory filesystem, we can select the size of the per-node filesystem to be larger than the available memory, preventing job failure with large shuffles.

We have run the spark-perf benchmarks used in Section 5.8 to compare performance between Lustre and Lustre-backed loopback file systems. Results for the Spark Core benchmarks are shown in Figure 65. Using per-node loopback filesystems improves performance at larger core counts for the *scala-agg-by-key* and *scala-agg-by-key-int* benchmarks, particularly for the latter which performs a larger number of opens. Results for the MLLib benchmarks are shown in Figure 66. The *lda*, *pic*, *spearman*, *chi-sq-feature* and *prefix-span* benchmarks show substantial improvement from the use of per-node loopback filesystems. Furthermore, they exhibit better scaling behavior on Cori than on Comet with local disk. Figure 67 shows weak-scaling performance with those benchmarks on Cori and Comet. Cori nodes provide more cores (32) than Comet nodes (24), although Comet nodes run at a higher clock speed (2.5GHz) than Cori nodes (2.3GHz).

5.10 Input Disk I/O versus Shuffle Disk I/O

The spark-perf benchmarks generate input data as tasks, with the input data therefore being either resident in memory or stored to disk as shuffle intermediate data prior to the start of the computation phase of the benchmark. The benchmarks also involve a large number of small transfers. Consider the Power Iteration Clustering (*pic*) benchmark. At the end of each iteration, the results for each partition are reduced to a single number, which is then used in an all-to-all reduction. This results in

Table 2. Messages sent in MLLib *pic* benchmark.

Nodes	# Msgs	Avg. Msg. Size (bytes)	Total KBytes
1	84	53.38	4
2	43,095	40.20	1,692
4	250,796	41.15	10,079
8	1,134,897	41.66	46,177
16	4,745,730	41.92	194,298
32	19,356,636	42.72	807,578
64	78,051,387	43.41	3,309,025

relatively minimal disk activity, with runtime dominated by a large number of very small inter-node communications. Table 2 shows the scaling behavior of the number of messages as the number of nodes and problem size grows: message sizes remain around 40-50 bytes, while the number of messages grows from 84 to 78 million. Since these benchmarks primarily stress horizontal (between-node) movement, the lack of degradation from using a Lustre-backed mounted file instead of a SSD or RAM disk is not strong evidence of their equivalence for other Spark workloads.

Therefore, we also analyzed the Mini TeraSort benchmark. This benchmark sorts 1 TB of key-value pairs per node, with each key and value being 8 bytes of randomly generated data. The data is pre-generated and written out to disk, and the sorted results are also written to disk, so that, for each input partition, disk I/O is required at the beginning and end of each stage, in addition to shuffle intermediate data. Figure 68 shows weak scaling results for Mini Terasort on Cori, with 50 GB of input data per node, so that a node's input data will fit in the RAM disk with sufficient space remaining for Spark. RAM disk and Lustre-backed mounted files provide equivalent performance, while storing input data on Lustre results in large increases in metadata access time. The usefulness of Lustre-backed mounted files thus remains for input-heavy workloads.

TeraSort uses a sequential data access pattern, reading the entirety of each partition file in order at the beginning of each benchmark. The SQL workloads in BigDataBenchmark provide a random data access pattern, and also allow us to examine the effects of native code generation optimizations present in Spark 2.0. Under Spark 1.6 and earlier, SQL queries are used to generate Scala code, which is executed by Spark tasks. In Spark 2.0 and later, an option is provided to instead generate native code, reducing time in computation. Figure 69 shows weak scaling results for BigDataBenchmark on 1, 5, and 20 nodes of Cori for Spark 1.6, Spark 2.0 with traditional Scala code generation, and Spark 2.0 with native code generation, and Spark 2.0 with both native code generation and file pooling, with input data stored on Lustre, RAM disk, or Lustre-backed mounted file. Additionally, performance is shown with input data stored on SSDs on Comet for the combination of Spark 2.0, native code generation, and file pooling. I/O time is comparable between the P configuration on Cori Lustre-mount and Comet SSD.

5.11 Localizing JVM and Spark Runtime Metadata Accesses

Merely running Spark from a Shifter container improves performance; for example, the GroupBy benchmark shows an improvement of 16% in total execution time on 10,000 cores, even without using the Lustre-backed mounted file capability which is the initial reason for our use of Shifter. This improvement occurs because the JVM, Spark class files, and the shared libraries used by them are themselves stored on a mounted disk image when Shifter is used. This localizes any metadata operations on those files which otherwise would be handled by the Lustre metadata server. Comparisons between runs on Cori using Shifter and runs on other systems, such as Comet, which do not have Shifter installed, may therefore be confounded by the effects of using a container at all.

While it is not possible to compare these systems with the same container infrastructure in place, as Shifter is specific to the Cray exeuction environment, Comet provides a different container solution, Singularity [130]. While Singularity does not provide an equivalent to Shifter's Lustre-backed mounted files, it does allow us to localize metadata accesses to executable code. Figures 70, 71, and 72 show results for the Power Iteration Clustering, Spearman Correlation, and Pearson Correlation benchmarks from spark-perf MLLib, respectively, on Comet and Cori, both with and without the corresponding container technology used. Shuffle intermediate data is stored in RAM disk. On 64 nodes, the use of Shifter on Cori reduces total execution time by 9.9%. The use of Singularity on Comet reduces total execution time by 9.5%.

5.12 Xeon Phi and the Effect of Straggler Tasks

The Cori system has recently been expanded with nodes containing socketed Intel Xeon Phi "Knights Landing" processors. Unlike the previous generation "Knights Corner" coprocessors, the Knights Landing is capable of running unmodified x86-64 executables. Therefore, it is possible to take the same Shifter image used in experiments on the Haswell compute nodes and use it, unmodified, on the Xeon Phi partition of Cori. As Spark workloads are not ordinarily floating-point-oriented, and provide little opportunity for vectorization, we would not expect good performance from most Spark workloads. Figure 73 shows weak scaling results for the GroupBy benchmark on Haswell with 32 workers per node (one per core), on KNL with 32 workers per node (same as Haswell), and on KNL with 68 workers per node (one per core).

Execution time on 32 cores of KNL per node is, on average, 3.8× that of 32 cores of Haswell per node, and 64 cores of KNL per node is, on average, 3.2× that of 32 cores of Haswell per node. The slowdown is more pronounced for the compute-dominated map phase (5.3×) than for the communication-dominated reduce phase (2.1×). This is attributed to a combination of increased scheduler delay and task deserialization time as seen in the execution traces shown in Figure 76 and increased time spent in JVM garbage collection, as shown in Figure 75. The increased latency to memory caused by the Knights Landing's use of MCDRAM is not an issue, as disabling MCDRAM by configuring the KNL in Flat memory mode instead of the default Cache memory mode does not effect performance, as shown in Figure 74.

Part of the slowdown is due to an increase in straggler tasks. Straggler tasks are tasks with take an unusually long time to complete compared to other tasks within the same stage, resulting in workers being left idle. To evaluate the effect of straggler tasks, we predict the performance of the benchmark if no straggler tasks were present. To do this, we replace tasks during which more than half the available workers are idle with tasks that take the median task time to execute for their stage and simulate their execution, repeating the process until no stragglers are identified. Figure 77 shows actual execution times compared to simulated execution times with no stragglers. On Haswell, the runtimes are nearly identical, as very few stragglers were present. On Knights Landing, the simulated execution times without stragglers are an average of 73% of the actual execution times.

5.13 Network Latency

As mentioned above and shown in Table 2, many data analytics applications involve a large number of small communications. As a result, past work has shown a significant benefit from optimizing for message latency through the use of RDMA techniques, such as through InfiniBand `ibverbs` [137]. As the Comet system is equipped with an InfiniBand network, we have evaluated the spark-perf MLLib benchmark on traditional Spark and RDMA-Spark; this is shown in Figures 78, 79, and 80. RDMA-Spark is not fully compatible with all benchmarks in the suite, resulting in some failed benchmarks at larger scales. However, where it functions, RDMA-Spark provides better scaling behavior than traditional Spark running on `IPoIB`.

Figure 81 shows results of a bandwidth microbenchmark on Comet and Cori for UDP packet injection and native RDMA packet injection for various packet sizes. Figure 82 shows CPU overhead of injection (that is, the time an application program is blocked on injection before it can handle another injection call). Figure 83 shows the end-to-end latency. While the difference in bandwidth between UDP and RDMA is minimal for small message sizes, end-to-end latency is 6× greater for UDP than RDMA for small messages on Comet and 5× greater for UDP than RDMA for small messages on Cori. Adapting RMDA-Spark for RDMA on the Cray Aries interconnect would therefore be expected to be beneficial.

5.14 Discussion

Metadata latency and its relative lack of scalability is a problem common to other [4, 35] parallel file systems used in HPC installations. The shuffle stage is at worst quadratic with cores in file open operations, thus metadata latency can dominate Spark performance. We believe our findings to be of interest to more than Cray with Lustre HPC users and operators. While Spark requires file I/O only for the shuffle phase, Hadoop requires file I/O for both map and reduce phases and also suffers from poor performance when run without local storage [198]. Our techniques may therefore also be applicable to Hadoop on HPC systems.

The hardware roadmap points towards improved performance and scalability. Better MDS hardware improves baseline performance (per operation latency), as illustrated by the differences between Edison and Cori. Multiple MDSes will improve scalability. The current usage of `BurstBuffer` I/O acceleration on Cori, while it degrades baseline node performance, it improves scalability up to thousands of cores.

Better performance from it can be expected shortly, as the next stage on the Cori software roadmap provides a caching mode for `BurstBuffer` which may alleviate some of the current latency problems. It may be the case that the `BurstBuffer` is too far from the main memory, or that it is shared by too many nodes for scales beyond $O(10^3)$. The HPC node hardware evolution points towards large NVRAM deployed inside the nodes, which should provide scalability with no capacity constraints.

As our evaluation has shown, software approaches can definitely improve performance and scalability. Besides ours, there are several other efforts with direct bearing. Deploying Spark on Tachyon [135] with support for hierarchical storage will eliminate metadata operations. In fact, we have considered this option ourselves but at the time of the writing the current release of Tachyon, 0.8.2, does not fully support hierarchical storage (missing append). We expect its performance to fall in between that of our configuration with a local file system backed by Lustre and `ramdisk+filepool`. Note also that our findings in Section 5.7 about the necessity of improving block management during the shuffle stage for iterative algorithms are directly applicable to Tachyon.

The Lustre roadmap also contains a shift to object based storage with local metadata. Meanwhile, developers [92, 198] have already started writing and tuning HDFS emulators for Lustre. The initial results are not encouraging and Lustre is faster than the HDFS emulator. We believe that the `lustremount` is the proper configuration for scalability.

The performance improvements due to `filepool` when using “local” file systems surprised us. This may come from the different kernel on the Cray compute nodes, or it may be a common trait when running in data center settings. As HPC workloads are not system call intensive, the compute node kernels such as Cray CNL may not be fully optimized for them. Running commercial data analytics workloads on HPC hardware may force the community to revisit this decision. It is definitely worth investigating system calls overhead and plugging in user level services (e.g. file systems) on commercial clouds.

Luu et al [140] discuss the performance of HPC applications based on six years of logs obtained from three supercomputing centers, including on Edison. Their evaluation indicates that there is commonality with the Spark behavior: HPC applications tend to spend 40% of their I/O time in metadata operations than in data access and they tend to use small data blocks. The magnitude of these operations in data analytics workloads should provide even more incentive to system developers to mitigate this overhead.

We are interested in extending this study with a comparison with Amazon EC2 to gain more quantitative insights into the performance differences between systems with node attached storage and network attached storage. Without the optimizations suggested in this chapter, the comparison would have favored data center architectures: low disk latency provides better node performance and masks the deficiencies in support for iterative algorithms. With our optimizations(`filepool+lustremount`), single node HPC performance becomes comparable and we can set to answer the question of the influence of system design and software configuration on scalability. We believe that we may have reached close to the point where horizontal data movement dominates in the HPC installations as well. Such a comparison can guide both system and software designers whether throughput optimizations in large installations need to be supplemented with latency optimization in order to support data analytics frameworks.

5.15 Related Work

Optimizing data movement in Map-Reduce frameworks has been the subject of numerous recent studies [226, 111, 137, 57]. Hadoop introduced an interface for pluggable custom shuffle [91, 226] for system specific optimizations. InfiniBand has been the target of most studies, due to its prevalence in both data centers and HPC systems. HDFS emulation layers have been developed for parallel filesystems such as PLFS [51] and PVFS [205]. These translate HDFS calls into corresponding parallel filesystem operations, managing read-ahead buffering and the distribution (striping) of data across servers. In Spark, only input and output data is handled through the HDFS interface, while the intermediate shuffle data is handled through the ordinary Java file API. Our work primarily optimizes intermediate shuffle data storage.

Optimizing the communication between compute nodes (horizontal data movement) has been tackled through RDMA-based mechanisms [226, 111, 137]. In these studies, optimized RDMA shows its best benefit when the data is resident in memory. Therefore, only the last stage of the transfer is carried out using accelerated hardware support. The client-server programming model is still employed to service requests because data are not guaranteed to be in memory. Performance is optimized through the use of bounded thread pool SEDA-based mechanism (to avoid overloading compute resources) [111], or through the use of one server thread per connection [226] when enough cores are available.

As we use network-attached storage, the bottleneck shifts to the vertical data movement. A recent study by Cray on its XC30 system shows that an improved inter-node communication support for horizontal movement may not yield significant performance improvement [198]. Note that this study for Hadoop also recommends using memory based file systems for temporary storage.

Optimizing vertical movement, which is one of the main motivation for the introduction of Spark, has been addressed by the file consolidation optimization [57] and by optimizations to persist objects in memory whenever possible. Our experiments have been performed with consolidation. We have analyzed the benefits of extending the optimization from per-core consolidation to per-node consolidation. As this will reduce only the number of file creates and not the number of file opens, we have decided against it.

5.16 Conclusion

We ported and evaluated Spark on Cray XC systems developed in production at a large supercomputing center. Unlike data centers, where network performance dominates, the global file system metadata overhead in `fopen` dominates in the default configuration and limits scalability to O(100) cores. Configuring Spark to use "local" file systems for the shuffle stage eliminates this problem and improves scalability to O(10,000) cores. As local file systems pose restrictions, we develop a user level file pooling layer that caches open files. This layer improves scalability in a similar manner to the local file systems. When combined with the local file system, the layer improves performance up to 15% by eliminating open system calls.

We also evaluate a configuration with SSDs attached closer to compute nodes for I/O acceleration. This degrades single node performance but improves out-of-the-box scalability from O(100) to O(1,000) cores. Since this is the first appearance of such system and its software is still evolving, it remains to be seen if orthogonal optimizations still need to be deployed with it.

Throughout our evaluation we have uncovered several problems that affect scaling on HPC systems. Fixing the YARN resource manager and improving the block management in the shuffle block manager will benefit performance.

Overall, we feel optimistic about the performance of data analytics frameworks in HPC environments. Our results are directly translatable to others, e.g. Hadoop. We scaled Spark up to O(10,000) cores and since, our NERSC colleagues have adopted the Shifter `lustremount` implementation and demonstrated runs up to 50,000 cores. Engineering work to address the problems we identified can only improve its performance. All that remains to be seen is if the initial performance and productivity advantages of Spark are enough to overcome the psychological HPC barrier of expecting bare-metal performance from any software library whatsoever.

5.17 Bridge

This chapter has described a tool-runtime integration with Spark, showing how such an integration can enable mapping performance results between the runtime level and user code level, and how this can enable users to understand the performance consequences of storage and work paritioning decisions. The next chapter describes a tool-runtime integration with OCR, a runtime which, unlike the other runtimes described thus far, is made *explicitly* aware of dependencies between tasks. The integration takes advantage of runtime dependency awareness to automatically diagnose causes of idleness and attribute those causes back to application code.

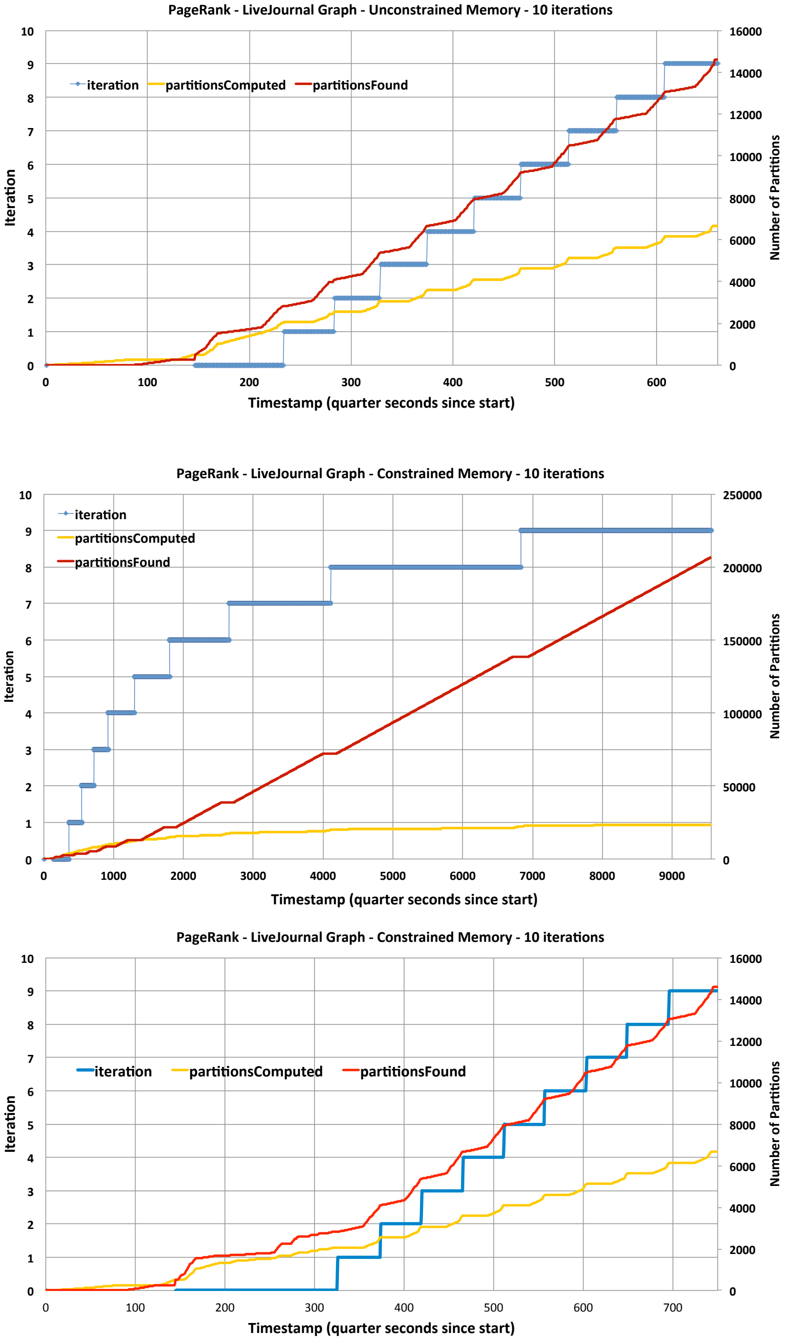


Figure 57. Number of partitions read during the shuffle stage for *PageRank*: Top: execution with unconstrained memory. Middle: when memory is constrained the number of partitions read from disk is one order of magnitude larger. Bottom: persisting intermediate results fixes the performance problems and we see a reduction by a order of magnitude in partitions read from disk.

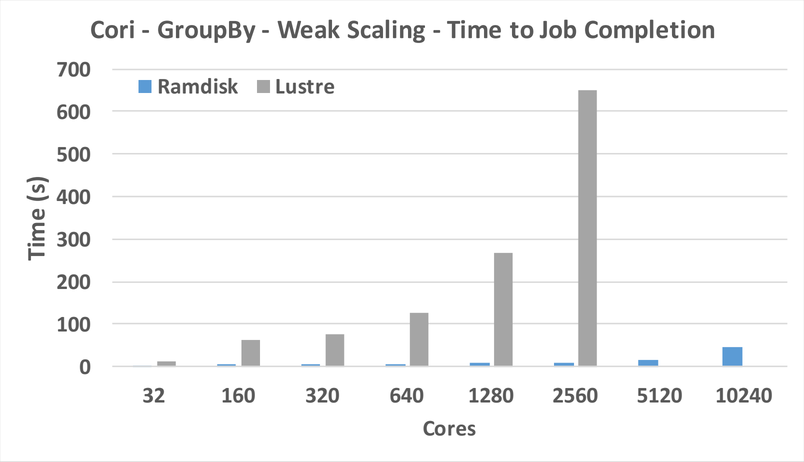


Figure 58. GroupBy benchmark performance (worst-case shuffle) on NERSC Cori, with shuffle intermediate files stored on Lustre or RAMdisk. Number of partitions in each case is 4 × *cores*

.

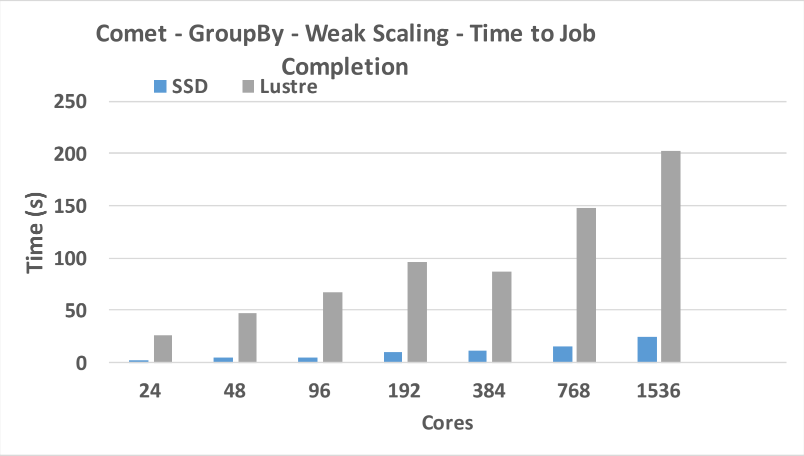


Figure 59. GroupBy benchmark performance (worst-case shuffle) on SDSC Comet, with shuffle intermediate files stored on Lustre or local SSD. Number of partitions in each case is 4 × *cores*

.

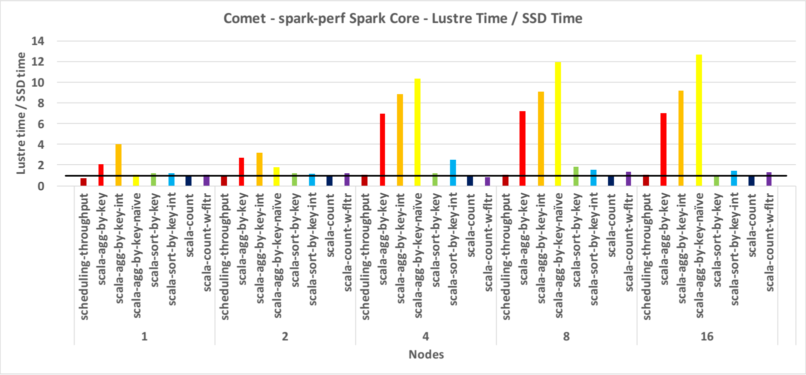


Figure 60. Slowdown of spark-perf Spark Core benchmarks on Comet with shuffle intermediate data stored on the Lustre filesystem instead of local SSDs.

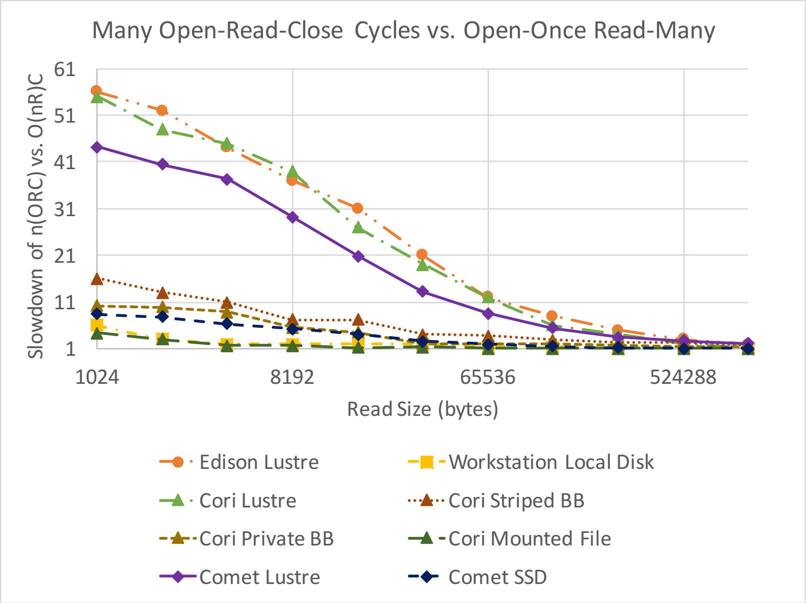


Figure 61. Slowdown from performing open-per-read rather than single-open many-reads for reads of different sizes on various filesystems on Edison, Cori, Comet, and a workstation with local disk. The penalty is highest for the Lustre filesystems.

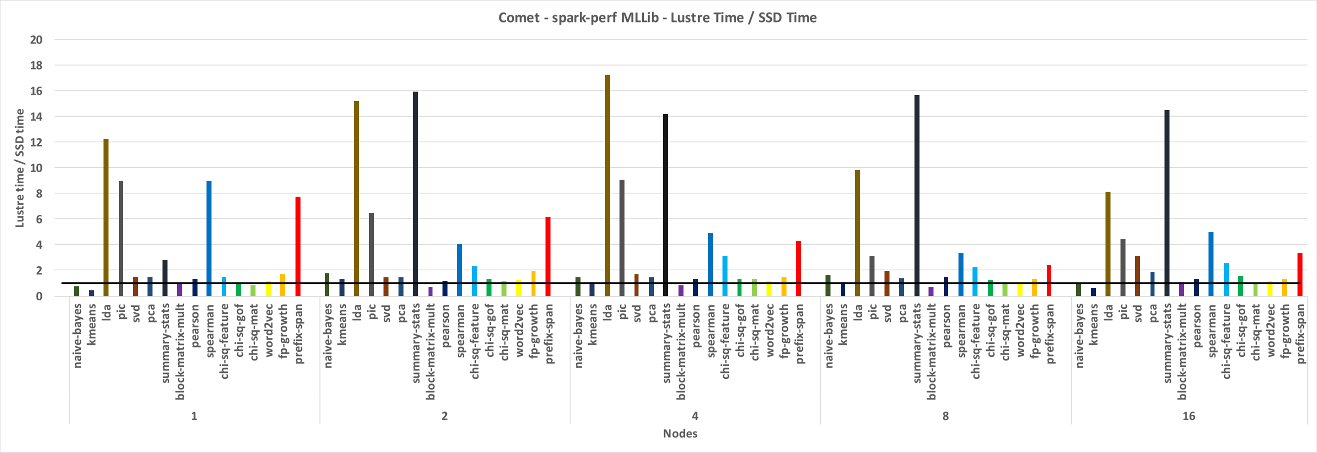


Figure 62. Slowdown of spark-perf MLLib benchmarks on Comet with shuffle intermediate data stored on the Lustre filesystem instead of local SSDs.

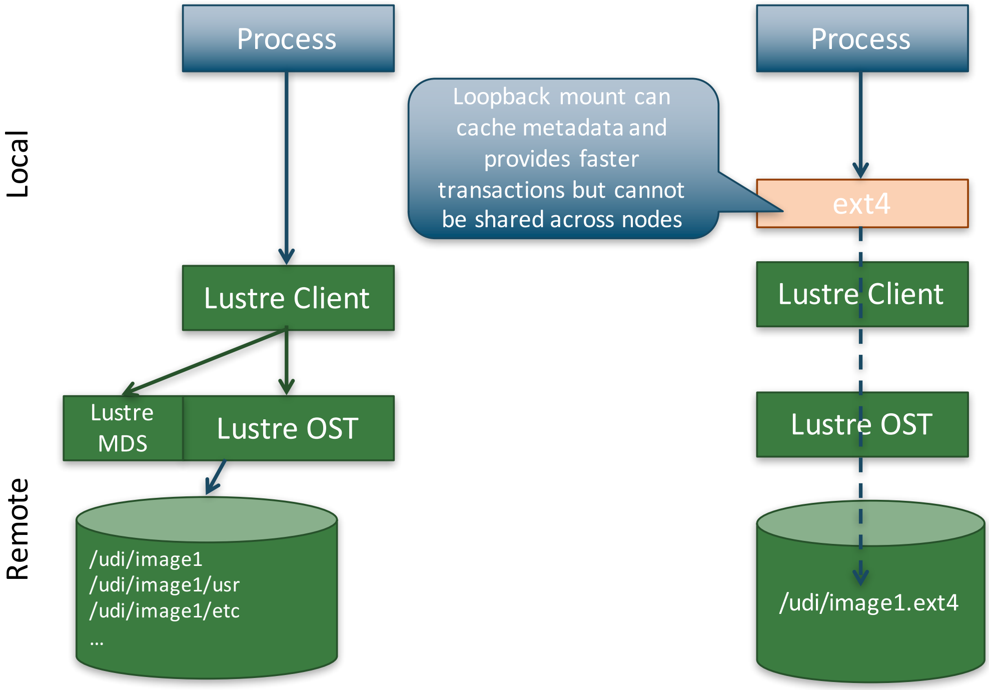


Figure 63. Architecture of Shifter. Shifter can mount node-local filesystems, keeping metadata operations local but preventing cross-node access.

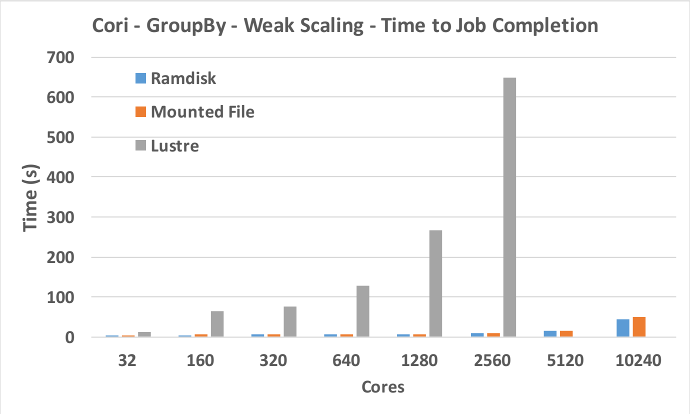


Figure 64. GroupBy benchmark performance (worst-case shuffle) on NERSC Cori, with shuffle intermediate files stored on Lustre, RAMdisk, or per-node loopback filesystems backed by Lustre files. Number of partitions in each case is 4 × *cores*

.

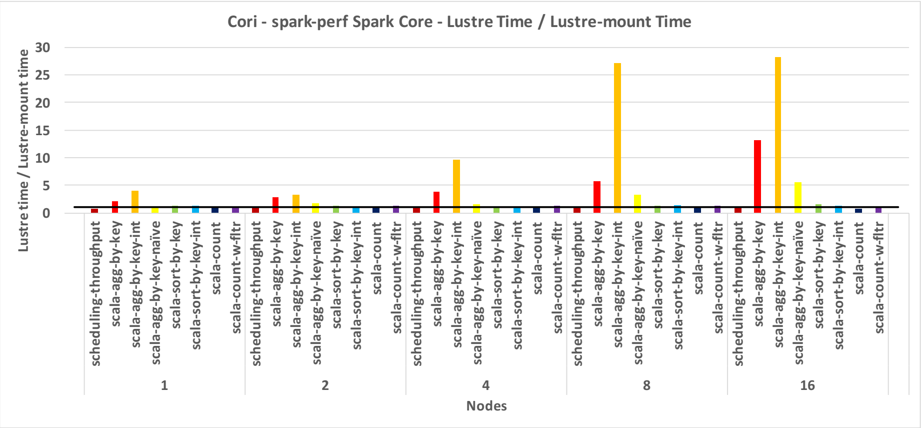


Figure 65. Slowdown of spark-perf Spark Core benchmarks on Cori with shuffle intermediate data stored on the Lustre filesystem instead of Lustre-backed loopback filesystems.

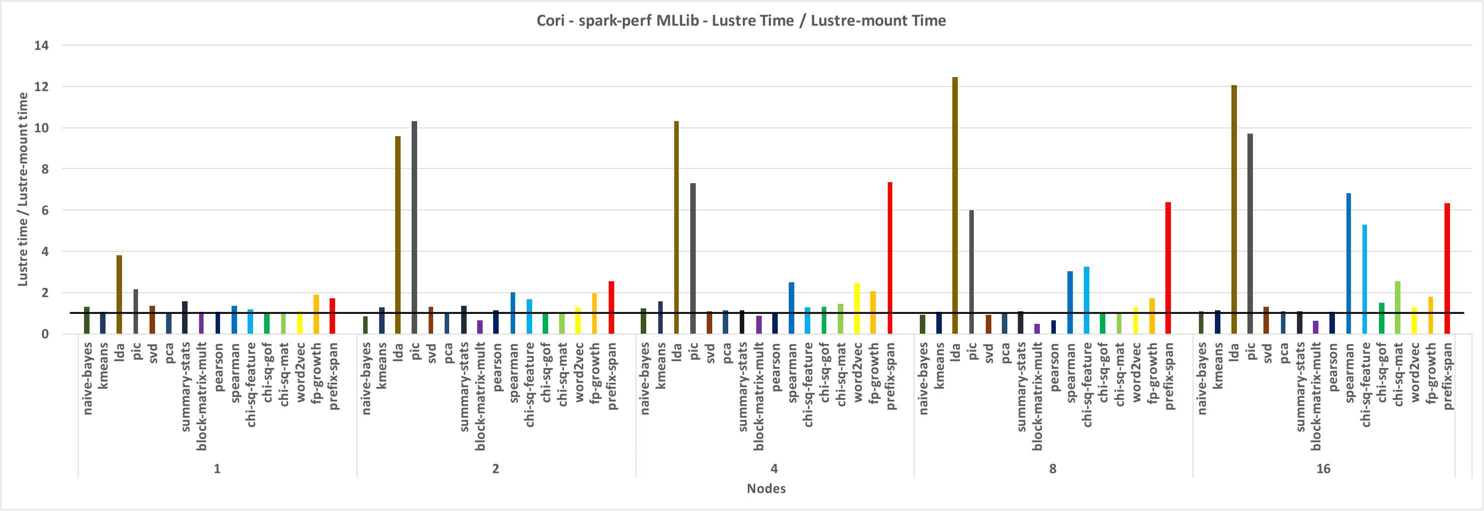


Figure 66. Slowdown of spark-perf MLLib benchmarks on Cori with shuffle intermediate data stored on the Lustre filesystem instead of Lustre-backed loopback filesystems.

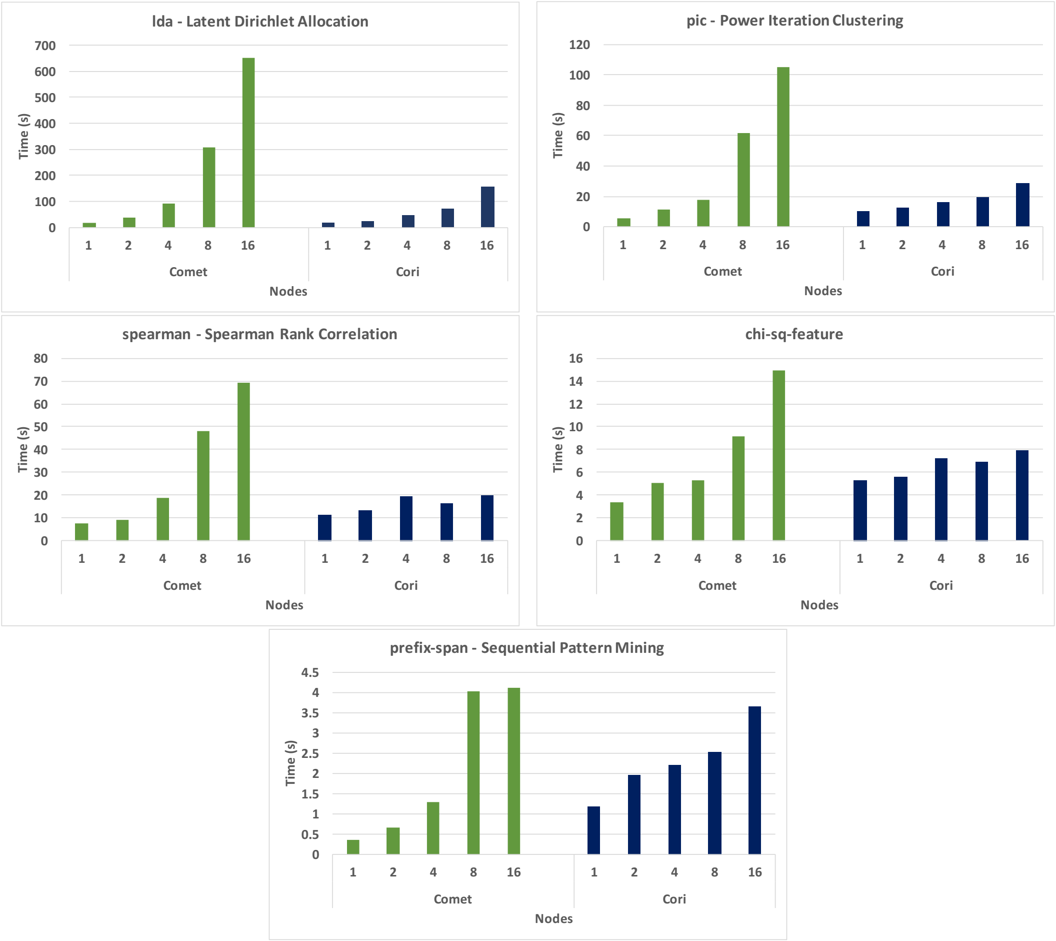


Figure 67. Weak scaling for the MLLib benchmarks most sensitive to shuffle performance on Cori with per-node loopback filesystems and on Comet with local SSDs.

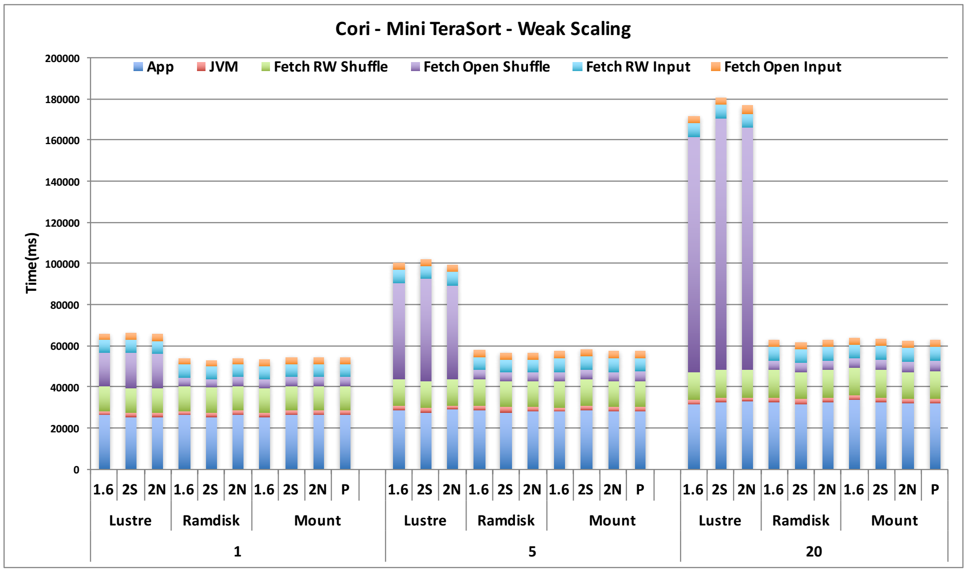


Figure 68. Weak Scaling results for Mini TeraSort on Cori, for 1, 5 and 20 nodes, with storage on Lustre, Ramdisk, or mounted file backed by Lustre. 1.6 indicates Spark 1.6. 2S indicates Spark 2.0 with the default Scala engine. 2N indicates Spark 2.0 with native code generation enabled, which is restricted to SQL queries. P indicates 2N with the addition of file pooling.

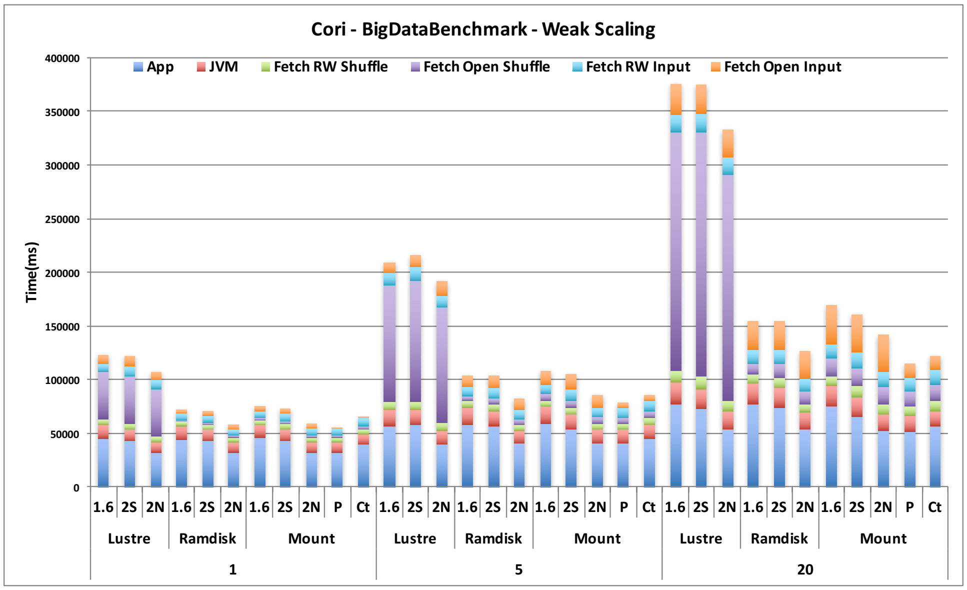


Figure 69. Weak Scaling results for BigDataBenchmark on Cori, for 1, 5 and 20 nodes, with storage on Lustre, Ramdisk, or mounted file backed by Lustre. 1.6 indicates Spark 1.6. 2S indicates Spark 2.0 with the default Scala engine. 2N indicates Spark 2.0 with native code generation enabled, which is restricted to SQL queries. P indicates 2N with the addition of file pooling. Ct indicates P on the Comet system rather than Cori.

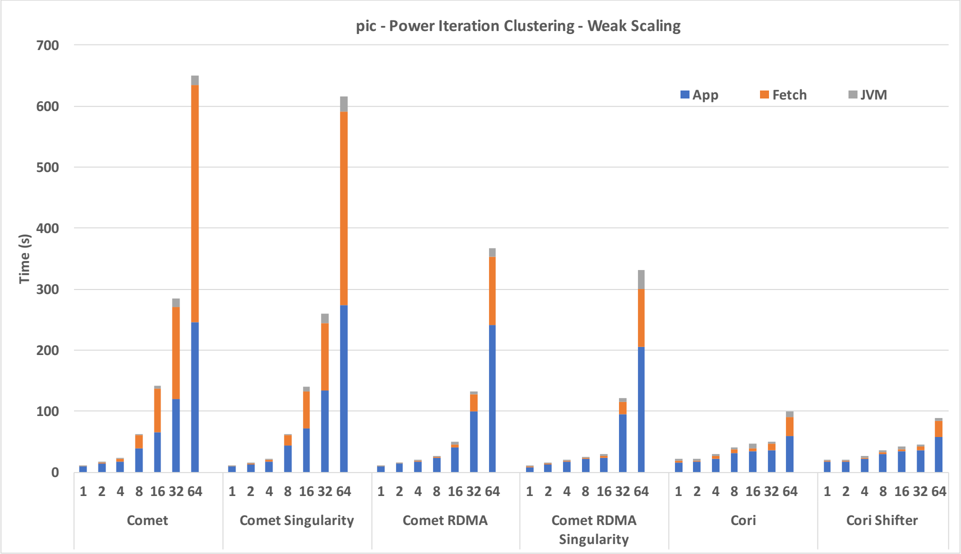


Figure 70. Weak Scaling results for Power Interation Clustering on Comet and Cori, with and without Singularity on Comet and Shifter on Cori.

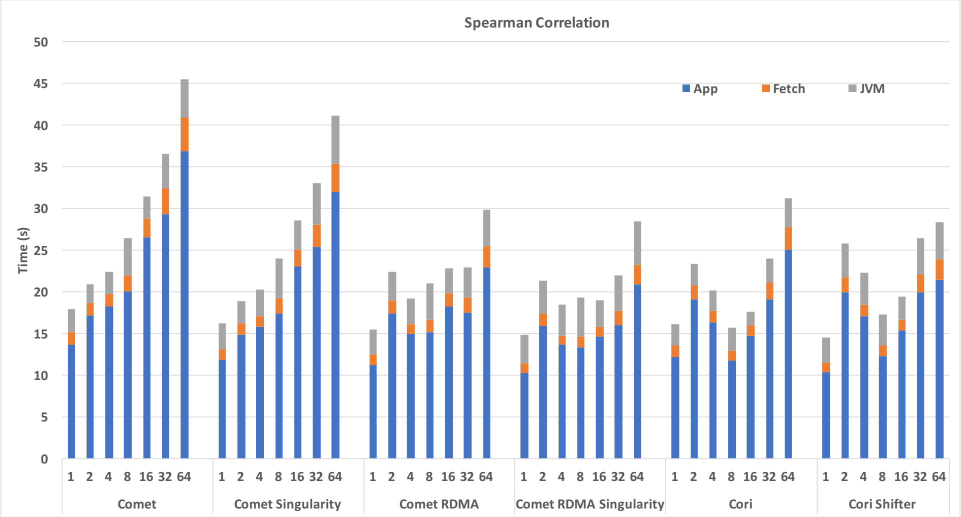


Figure 71. Weak Scaling results for Spearman Correlation on Comet and Cori, with and without Singularity on Comet and Shifter on Cori.

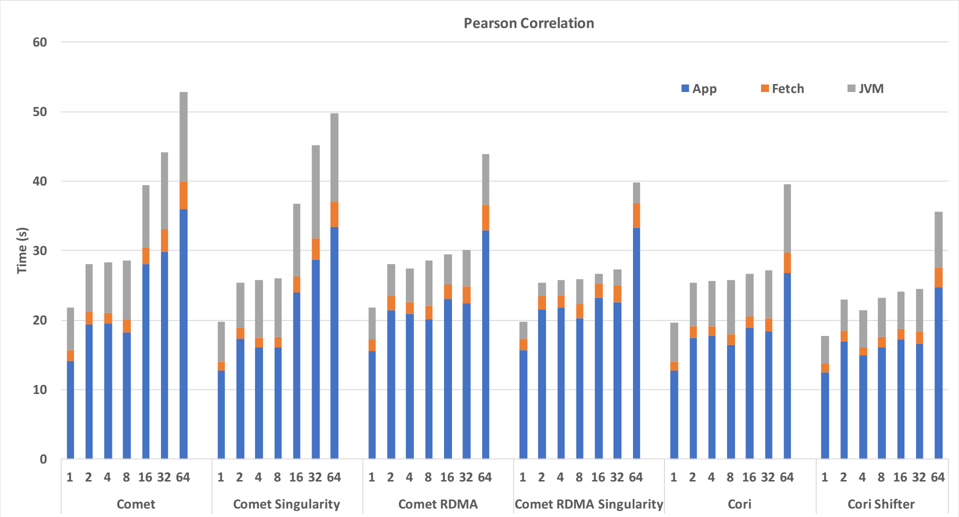


Figure 72. Weak Scaling results for Pearson Correlation on Comet and Cori, with and without Singularity on Comet and Shifter on Cori.

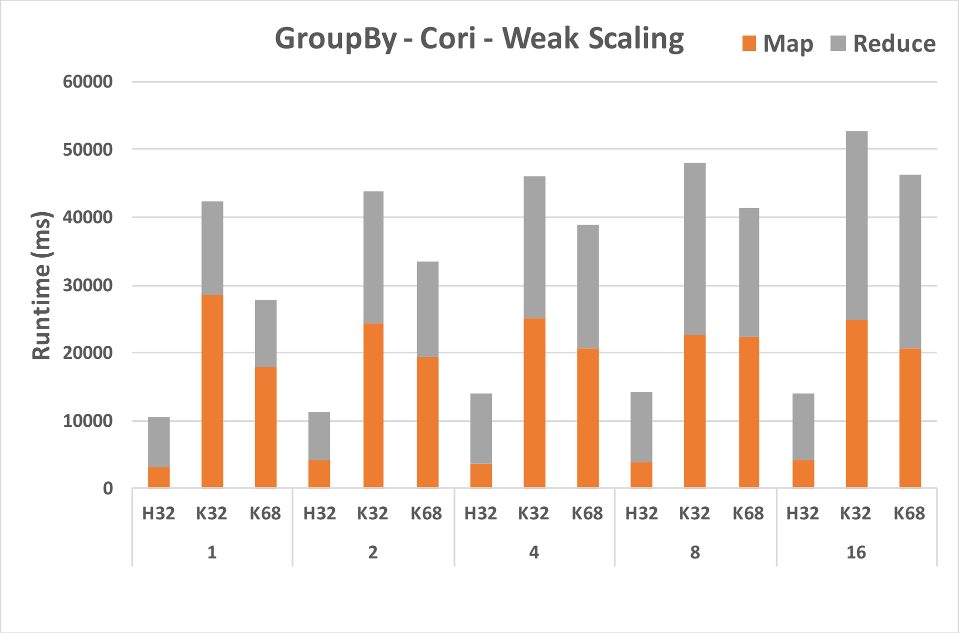


Figure 73. Weak Scaling results for GroupBy on Xeon Phi and Haswell nodes of Cori.

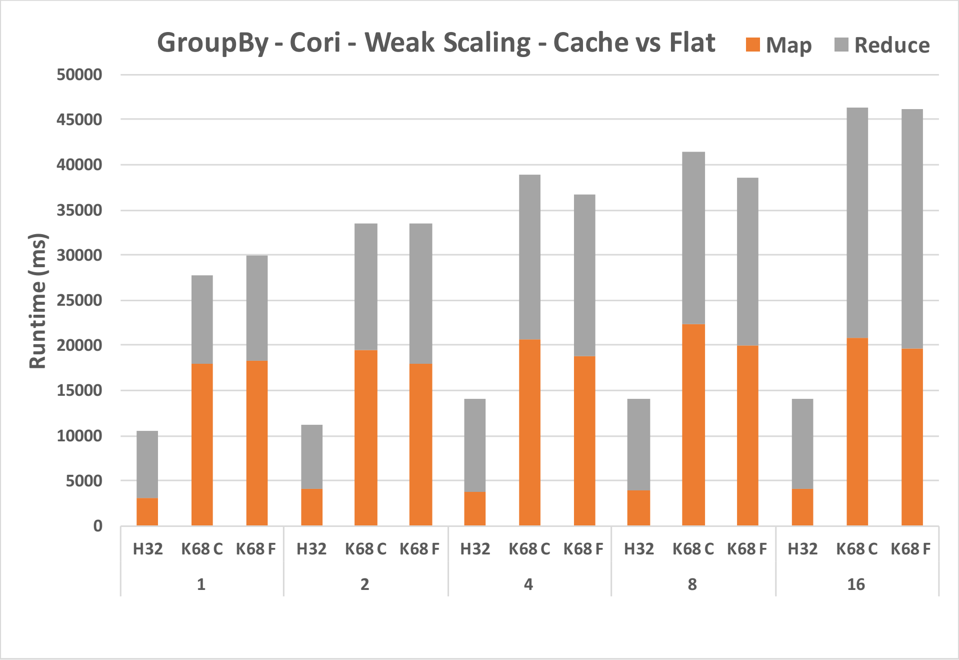


Figure 74. Weak Scaling results for GroupBy on Xeon Phi in Cache vs Flat MCDRAM mode and Haswell nodes of Cori.

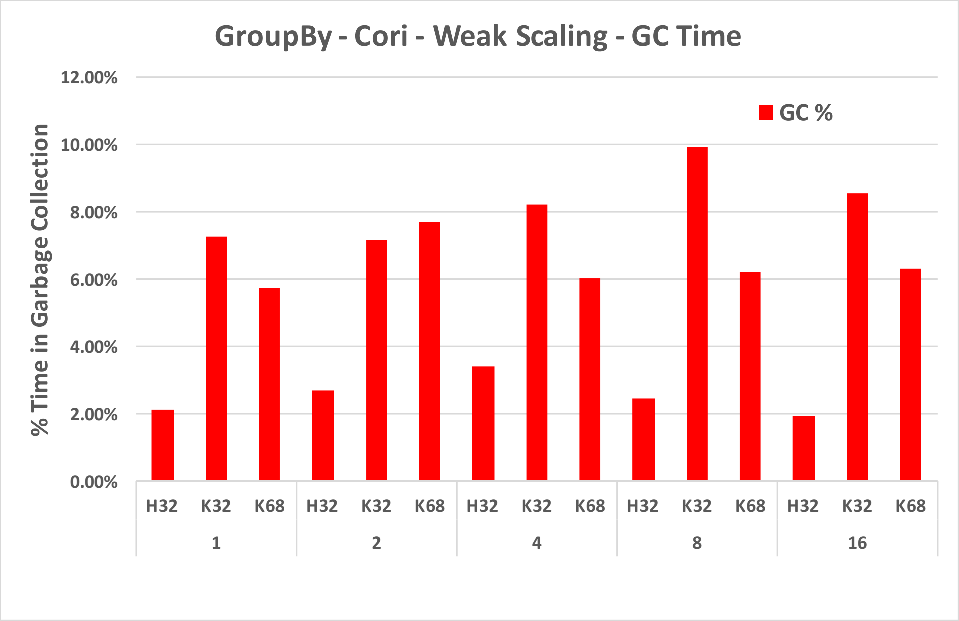


Figure 75. Time spent in garbage collection for GroupBy on Xeon Phi and Haswell nodes of Cori.

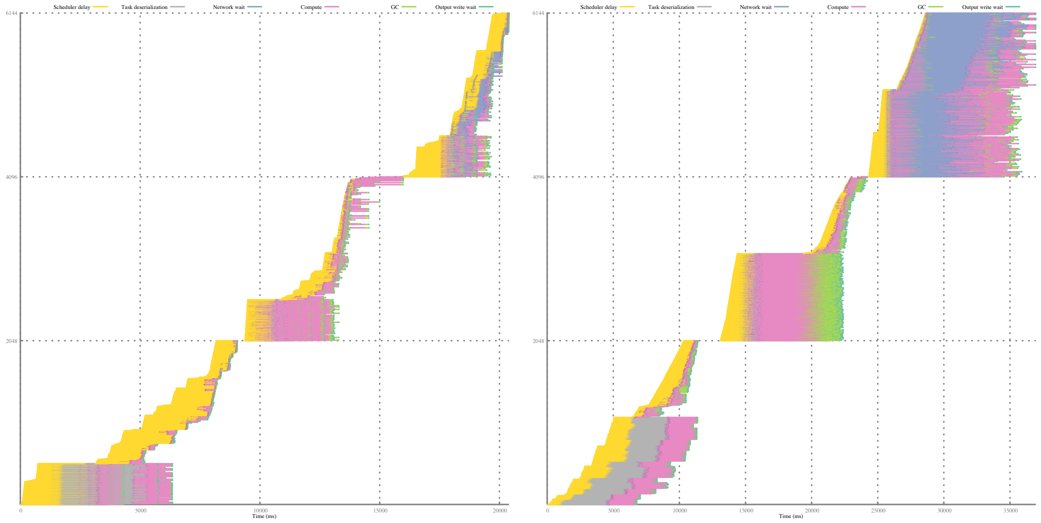


Figure 76. Execution traces of GroupBy on Haswell (left) and Xeon Phi (right) nodes of Cori.

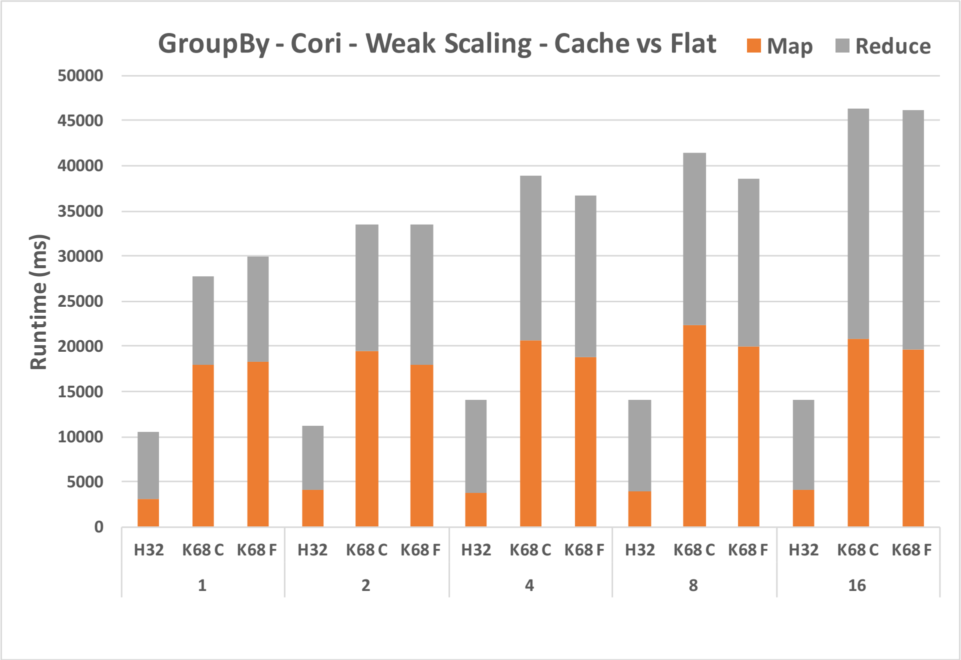


Figure 77. Hypothetical execution time of GroupBy after removing stragglers. Straggler tasks are replaced with median execution time.

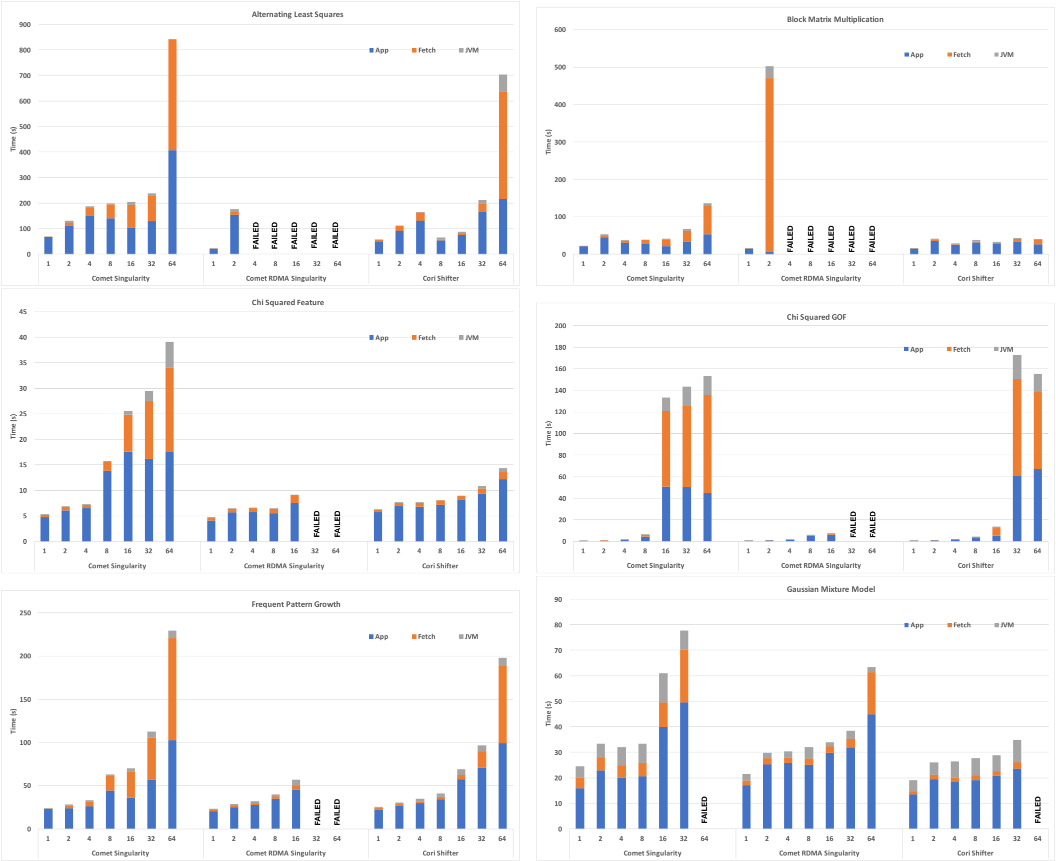


Figure 78. Weak scaling for the first block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori.

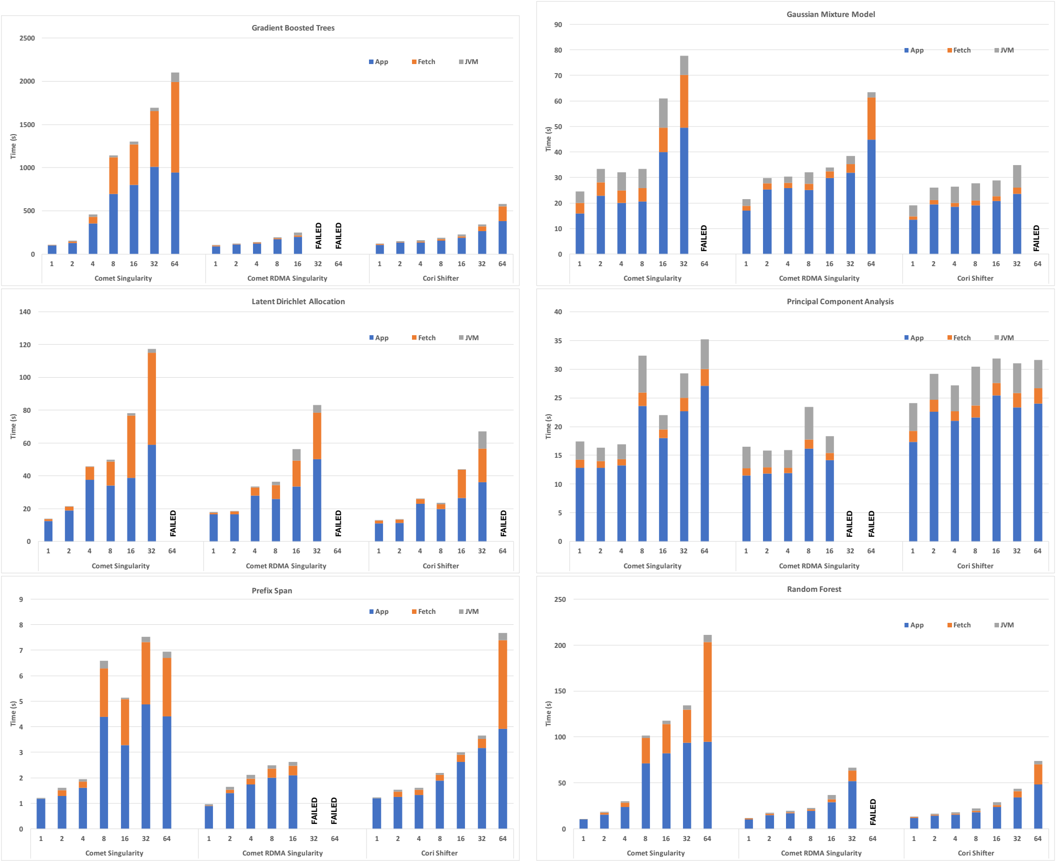


Figure 79. Weak scaling for the second block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori.

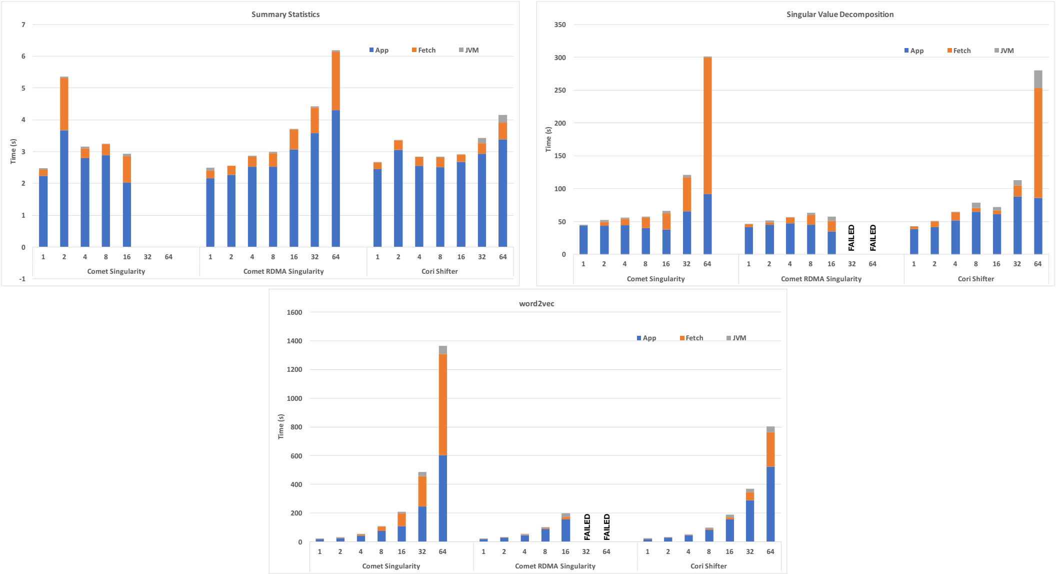


Figure 80. Weak scaling for the third block MLLib benchmarks for Comet IPoIB, Comet RDMA, and Cori.

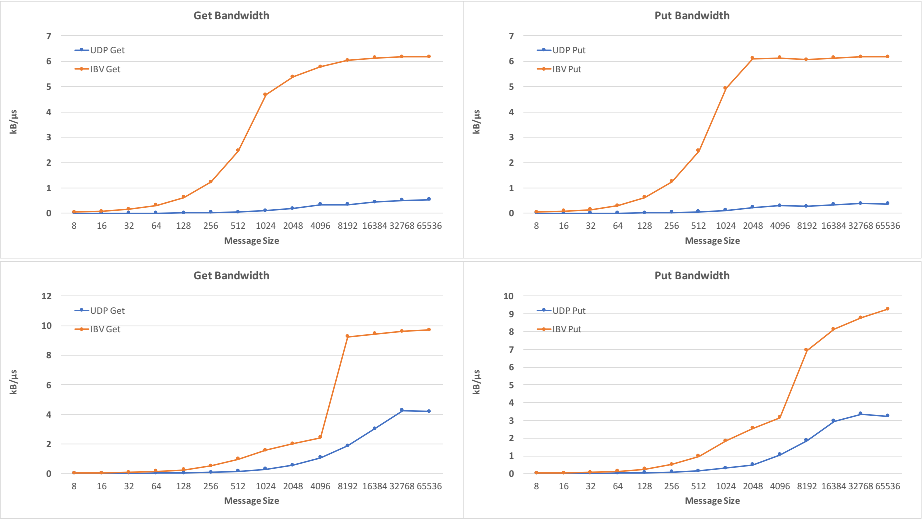


Figure 81. Bandwidth by message size for UDP and native RDMA on Comet and Cori.

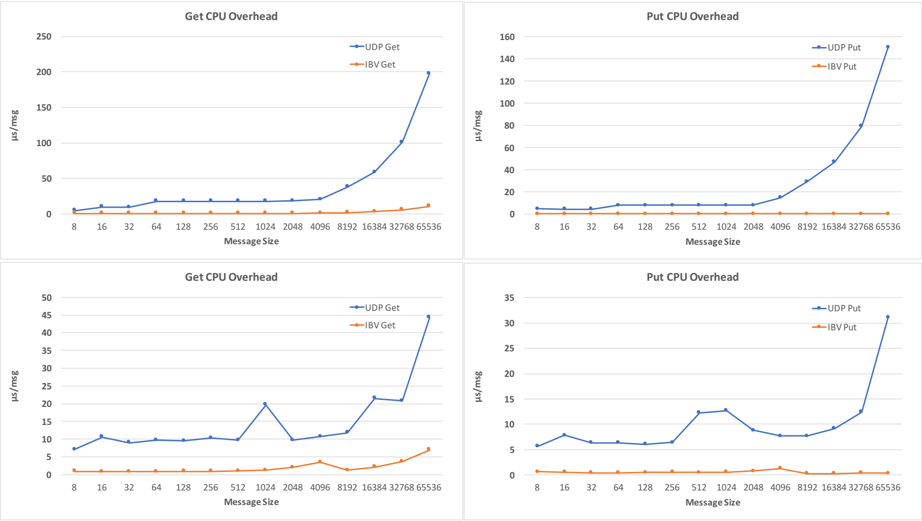


Figure 82. Injection CPU overhead by message size for UDP and native RDMA on Comet and Cori.

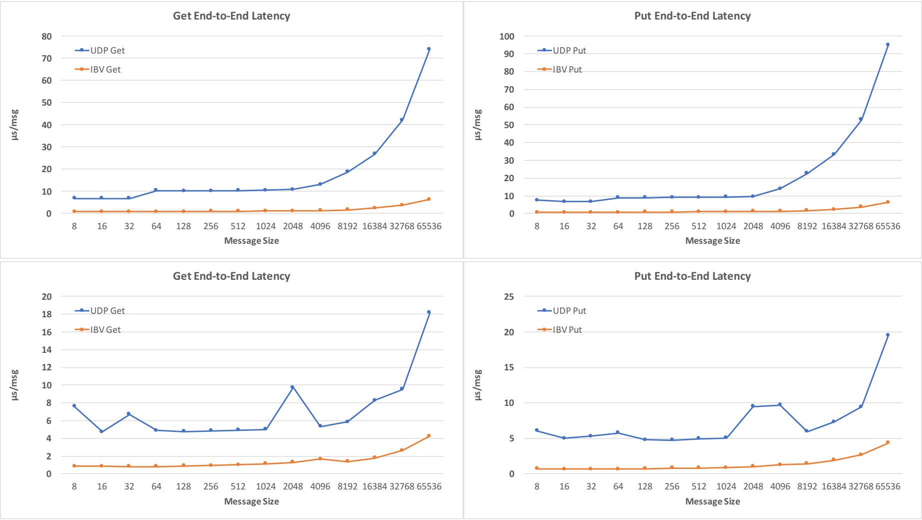


Figure 83. End-to-end latency by message size for UDP and native RDMA on Comet and Cori.

CHAPTER VI

STRAGGLER ANALYSIS IN OCR

6.1 Introduction

The Open Community Runtime (OCR) [152] is a distributed, task-based runtime which provides abstractions for both work and data, and which represents both explicitly. The work abstraction is called an *event-driven task*, or EDT, and consists of a set of code with explicitly defined inputs and a single output, with the restriction that no synchronization or communication can occur within the task. All synchronization and communication is instead handled by the runtime, which is made aware of task dependencies, and does not schedule an EDT until all its dependencies have been computed and are available on the node on which the EDT resides. The runtime handles data through the *datablock* abstraction, which is a fixed-size block of data which can serve as input to or output from an EDT. EDTs acquire datablocks with a certain level of access privilege specifying whether the EDT requires exclusive access to the block of memory. Once an EDT begins executing, it runs to completion; unlike in systems such as HPX, a task never yields during execution. The runtime also provides *events*, which can be viewed as empty, or control-only, tasks, for synchronization purposes. An OCR program is implemented as a set of library calls to the OCR runtime, which create events, EDTs and datablocks; which connect the input and output slots of the objects; and which satisfy dependencies.

It is possible to use traditional performance monitoring tools with OCR applications, and correct results will be obtained. The results will not be insightful, however. Capturing function-level profiles will provide a runtime-centric view of application execution, which shows *what* tasks are running, but not *why*. It captures nothing about what tasks *exist*, but are *not* running, which is essential to diagnosing performance problems. It captures nothing about dependencies between tasks. Also complicating performance analysis is OCR's use of *deferred API calls*, in which an API call within an EDT may returns as soon as possible, deferring processing by the runtime until the end of the EDT. As such, when we observe functions executing within the runtime, the call stack does not necessarily provide the true context of the call, making it difficult to attribute the library time back to an application level EDT.

Tool-runtime integration with OCR can give us actionable data: as I will show in this chapter, we can automatically diagnose several causes of idleness during program execution, providing feedback to the user on the number of workers needed for optimal execution and on task granularity, and providing online feedback to the runtime for load balancing.

6.2 APEX-OCR Integration

To provide insightful and actionable performance information, we integrated OCR with the APEX system described in Chapter IV. The basic architecture of the integration is shown in Figure 84. The OCR runtime provides a built-in tracing interface which records a trace of all API calls and runtime events. These are implemented in OCR as a set of weakly-defined functions called by the runtime, for which the default implementations write to a trace file. APEX provides strongly-defined versions of this function, so that whenever an OCR application is linked against APEX, APEX's versions of the functions will override the versions built into OCR. The APEX versions forward event data to the APEX handlers.

As APEX was originally designed for HPX, and OCR has concepts absent from HPX, some additional event types were added to APEX. These are represent task creation and destruction, a task becoming runnable, datablock creation and destruction, a task or event acquiring or releasing a datablock, and dependencies being added or

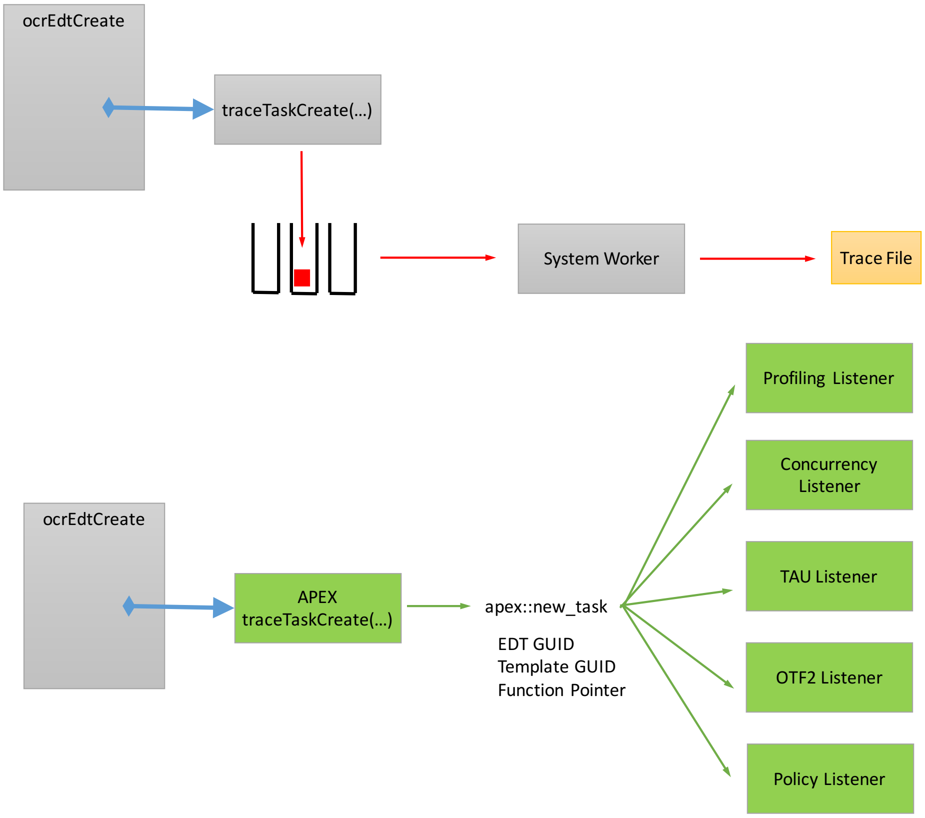


Figure 84. Design of the OCR-APEX interface. On top, the default configuration of OCR, using its built-in tracing support. On bottom, APEX replacing the default tracing handler and dispatching events to its various listeners.

satisfied. Each of these event types can occur in an API context (which represents the site at which the application code requested that something occur) and in a runtime context (which represents the runtime's processing of a deferred call). The API and runtime versions of an APEX event may occur on different nodes: for example, if an EDT satisfies a dependency for an OCR event or EDT which resides on a different node than the one executing the EDT making the call, the API event will occur on the local node and the runtime event will occurs some time later on the remote node. APEX's task identifiers were extended to carry globally unique identifiers (GUIDs), in addition to a name or address, in order to correlate API and runtime events and to identify the objects in the dependency graph. These events allow APEX to create visualizations showing the relationships between tasks. Figure 85 shows the task creation graph for the High Performance Conjugate Gradient (HPCG) miniapp [61].

APEX's concurrency visualization, as described in Chapter IV, is particularly useful, as one can see at a glance whether the application is making use of all the computational resources (workers/cores) available to it, as well as identify application phases. Figure 86 shows a concurrency visualization of the Stencil2D benchmark running on four nodes of the X-Stack cluster installed at Intel. There is low concurrency during the initialization phase of the application, and it thereafter uses 45-50 of the 64 available worker cores. The worker cores are predominately executing application tasks, with the FNC_timestep and FNC_update tasks dominating the computation. In contrast, the concurrency visualization for the HPCG miniapp, shown in Figure 87, shows that very few workers are executing application code, with the vast majority of workers instead executing the processRequestEdt task. This is a task internal to the OCR runtime which processes remote invocations.

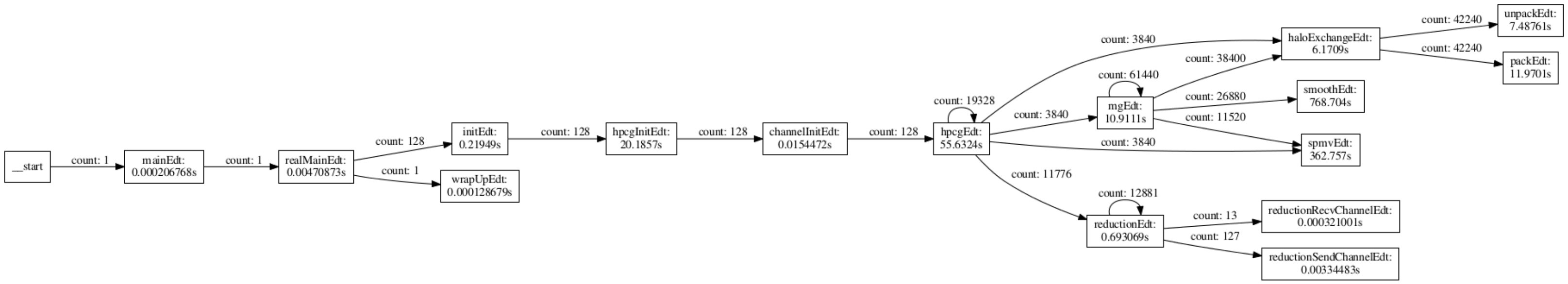


Figure 85. Task creation graph of the HPCG miniapp.

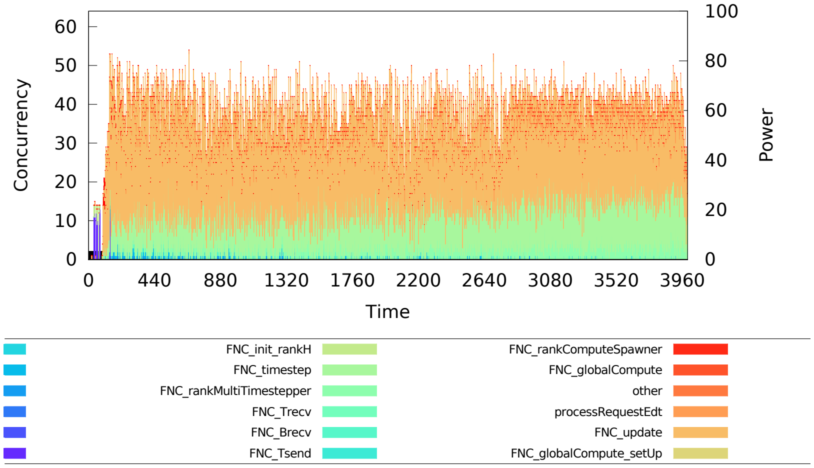


Figure 86. Concurrency visualization of the Stencil2D benchmark.

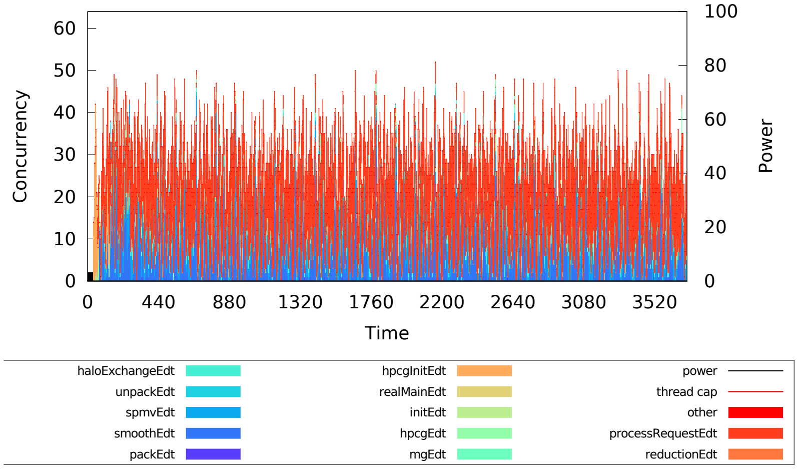


Figure 87. Concurrency visualization of the HPCG miniapp.

6.3 Task Eligibility

To identify and understand the causes of the low user-EDT utilization in the HPCG benchmark, we extended the concurrency visualization to show *actual concurrency* (the tasks which are currently executing), as it already did, and also *available concurrency* (tasks which are available for execution, but which are not executing, usually due to a lack of resources) and *unavailable concurrency* (tasks which have been created, but which are not executing, and cannot currently execute because not all of their dependencies have been satisfied). In HPX, there is not a concept of unavailable concurrency, as a task could execute until it requires data not yet available, at which point it would suspend. Figure 88 shows a concurrency eligibility visualization for a single node of the HPCG benchmark, with the bars extending above the X axis indicating actual concurrency and the bars extending below the X axis indicating available concurrency (dark blue) and unavailable concurrency (light blue). This visualization omits processRequestEdt, as it is internal to the runtime and does not represent work from the application developer's perspective.

The zoomed-in region in Figure 88 helps show the phase structure of the application. The application alternates between periods of full occupancy of the workers and periods of less than full occupancy, dropping as low as one active worker. The visualization shows that, during these regions of low occupancy, there is no available concurrency – all the EDTs which have been created and not yet executed are not ready to execute. The application is experiencing repeated bottlenecks.

6.4 Tracing OCR Applications

We would like to understand the *causes* of bottlenecks in OCR applications. To do this, we have developed a tracing infrastructure for task-based applications and a trace visualizer and automatic performance diagnosis tool. The tracing infrastructure

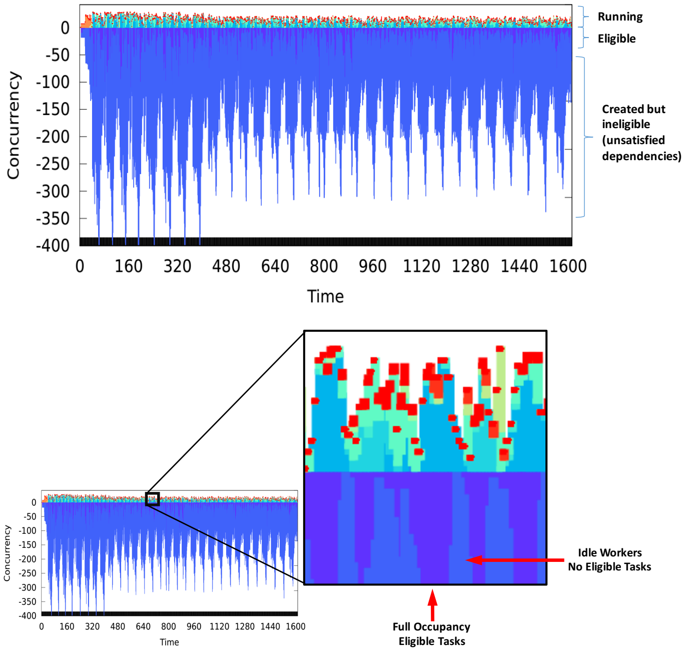
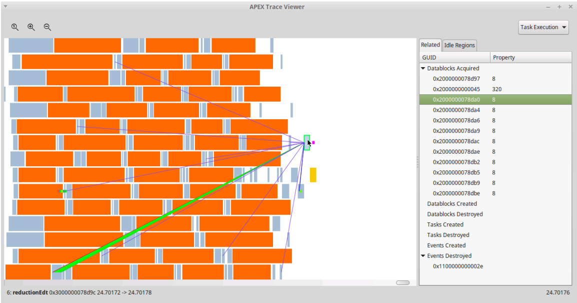


Figure 88. Task eligibility visualization of the HPCG miniapp. Light blue tasks, shown below the X-axis, are created but ineligible due to unsatisfied dependencies. Dark blue tasks are created, and eligible to run, but have not yet begun execution. Zoomed region shows idle regions with no eligible tasks available for scheduling.

is an extension of Open Trace Format 2 (OTF2) [70] with additional event record types for task creation and destruction, datablock creation and destruction, and dependency specification and satisfaction. For task execution, the existing `Enter` and `Leave` records are used, so that the resultant traces are compatible with existing OTF2 visualizers such as Vampir, albeit without support for task-specific event record types. GUIDs are stored in the otherwise unused Description field of OTF2 regions, which are referenced by `Enter` and `Leave` event records. The existing `ThreadTaskCreate`, `ThreadTaskSwitch`, and `ThreadTaskComplete` event record types are not used, as the semantics of these types are strongly based on the OpenMP tasking model and do not support, among other things, a distinction between a task's creation and start of execution, or between a task's completion and destruction.



GUID	Property
Datablocks Acquired	
0x2000000078d97	8
0x2000000000045	320
0x2000000078da0	8
0x2000000078da4	8
0x2000000078da6	8
0x2000000078da9	8
0x2000000078dac	8
0x2000000078dae	8
0x2000000078db2	8
0x2000000078db5	8
0x2000000078db9	8
0x2000000078dbe	8
Datablocks Created	
Datablocks Destroyed	
Tasks Created	
Tasks Destroyed	
Events Created	
Events Destroyed	
0x110000000002e	

6: **reductionEdt** 0x3000000078d9c 24.70172 -> 24.70178

24.70176

Figure 89. User interface of the APEX Trace Viewer.

We designed a prototype trace visualizer, the APEX Trace Viewer, shown in Figure 89. The Trace Viewer shows a timeline view with each EDT's execution represented by a colored rectangle, where the color of the rectangle is determined by the EDT template used in constructing the EDT. Selecting an EDT in the Viewer will also show all API and runtime events involving the EDT, and lines are drawn connecting the runtime events associated with EDT creation, dependency addition, and dependency satisfaction.

6.5 Blame Analysis

With the ability to collect traces, we designed a tool to automatically identify and diagnose idle regions such as those described in HPCG earlier in this chapter. This occurs in the following steps:

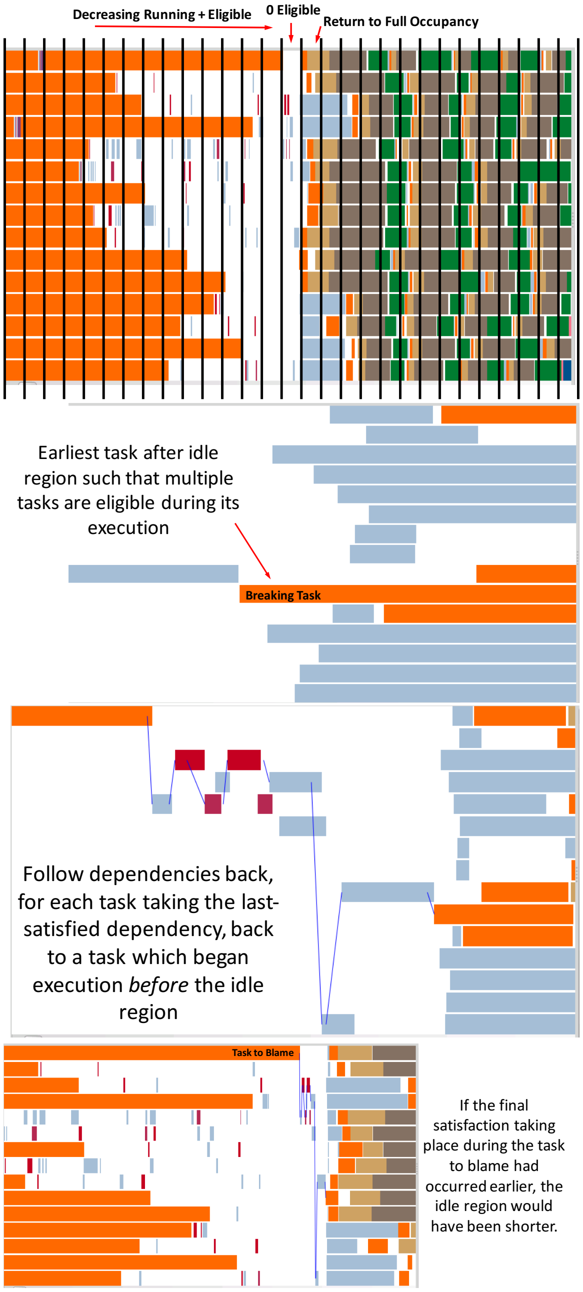
Idle Region Detection. The trace is sampled at intervals. Idle regions are identified as a contiguous set of samples such that the first sample in the idle region has full occupancy (that is, every worker is running some task), subsequent contiguous samples show decreasing occupancy down to an occupancy of one or zero, after which occupancy returns to full.

Breaking Task Identification. Next, the *breaking task* is identified. This is the task which, if it had completed earlier, would have caused the return to full occupancy to occur earlier. It is identified by locating the first-started task at the end of the idle region such that more than one task is running, and then following its dependencies backwards in time to identify the task which performed the final satisfaction necessary for it to run. The task performing the final satisfaction is the breaking task.

Initiating Task Identification. Next, the chain of last satisfactions is followed backwards from the breaking task to the latest-occurring task in the dependency chain which has a start time prior to the beginning of the idle region. This is the initiating task, and represents the task which, if it had completed earlier, would have shortened the duration of the idle region.

These stages are shown using examples in the Trace Viewer in Figure 90.

Once we have identified an initiating task, there are several possible suggestions we can make for fixing the issue, depending on the circumstances surrounding the start



Earliest task after idle region such that multiple tasks are eligible during its execution

Follow dependencies back, for each task taking the last-satisfied dependency, back to a task which began execution *before* the idle region

If the final satisfaction taking place during the task to blame had occurred earlier, the idle region would have been shorter.

Figure 90. Process for assigning blame for idle regions. First, idle regions are identified. For each idle region, a *breaking task* is identified, being the earliest-occurring task extending outside of the idle region. The chain of dependencies is then followed back from the breaking task to the latest-occurring ancestor at least partly outside the idle region.

of the initiating task. If the initiating task could have started earlier (that is, it became eligible earlier than it actually ran), but did not start earlier because no workers were available, then the problem can be diagnosed as a lack of computation resources: the idle region can be shortened by adding a worker. If the initiating task could not have started earlier than it did, we can look at other causes. If the initiating task is longer running than other tasks which are predecessors of the breaking task, then the task is a straggler. In this case, work is not equally divided among tasks, and repartitioning of work or increasing task granularity is suggested.

In the case of HPCG, the blame analysis tool identifies 80 idle regions, one for each reduction phase in the application. These are natural bottlenecks, because data from each input task is used as input to a series of reduction tasks culminating in a single, base-level reduction task. However, the idle regions could be shortened. Initially, a lack of resources is identified as the issue. Providing enough resources shortens the idle region and results in the identification of straggler tasks. These straggler tasks, in turn, are caused by variability in the time to acquire a GUID within tasks.

6.6 Load Balancing

Another possibility for diagnosing idle regions is poor load balancing, and this is amenable to an online policy to correct the issue. We diagnose load balance as an issue when a task could have started earlier than it actually did, workers were not available on the node on which the task was resident, but workers *were* available on some other node. OCR places EDTs at EDT creation time, but provides an opportunity to move an EDT once all dependencies have been satisfied.

We developed a load-balancing policy as an APEX triggered policy which runs when the `TaskRunnable` event occurs. To avoid the unacceptable overhead of having a centralized scheduler, each node has its own instance of the policy, and each policy instance maintains a local approximation of the overall load on workers, which is updated based on a gossip protocol [158]. Load data is distributed in two ways: first, after a configurable number of messages sent by the runtime from one node to another the runtime sends, APEX sends, along with the runtime message, its view of system load, which is represented by the number of created-but-ineligible, eligible, and running tasks for each node. Each local view of system node is versioned, with a version associated with each node's load. When node B received load from node A, it updates its local view of node A's load, as well as the views of nodes other than A and B with node A's loads, so long as it does not already have more recent load data from the other node. Additionally, when load information is sent, a random other node is selected and also receives load data. This prevents underutilized nodes from being unaware of load elsewhere on the system.

When a task becomes runnable, the task is randomly assigned to another node with a probability inversely proportional to the load on the system, which is defined as the number of running and eligible nodes. In naturally unbalanced applications such as MiniAMR, this results in a substantial improvement in performance. Figure 91 shows traces of executions of MiniAMR with tasks being scheduled as assigned at creation time (bottom) and dynamically reassigned when eligible by APEX (top). Idle regions are shortened by use of the APEX policy, and overall execution time is 21% faster with the policy than without.

6.7 Conclusion

In this chapter, we have shown how the APEX performance monitoring system can be integrated into the Open Community Runtime, producing performance profiles, concurrency visualizations, and traces which incorporate runtime-specific

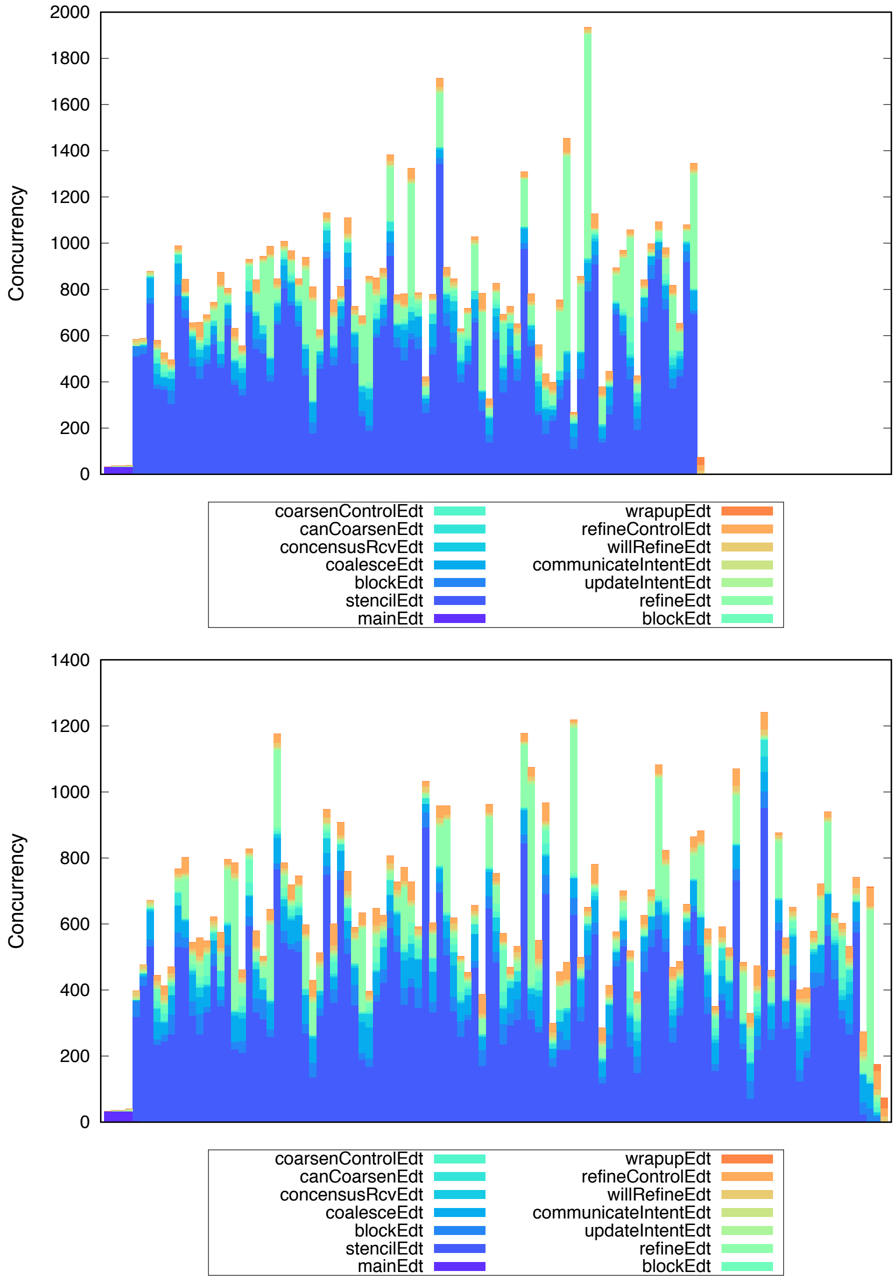


Figure 91. OCR MiniAMR with (top) and without (bottom) load balancing policy.

information. Through this integration, we are able to provide automatic diagnosis of causes of idleness in OCR applications, as well as to provide for a load balancing policy which can mitigate idleness caused by improper load balancing.

6.8 Bridge

This chapter has described a tool-runtime integration with OCR, showing how such an integration can enable dynamic load balancing and automatic performance issue diagnosis. The next chapter describes a tool-runtime integration unlike the others described thus far: it is with a traditional runtime, OpenMP, rather than a many-task runtime. This integration is used for online adaptation of thread scheduling under varying power constraints, and demonstrates that tool-runtime integration is useful both inside and outside the domain of many-task runtimes.

CHAPTER VII

OPTIMIZING SCHEDULING IN OPENMP

This chapter includes co-authored material previously published in the IEEE International Conference on Cluster Computing (CLUSTER 2016) [191]. That paper was a collaboration with Md. Abdullah Shahneous Bari, Abid M Malik, Kevin Huck, Barbara Chapman, Allen D. Malony and Osman Sarood. I wrote the APEX tuning policy and ran and analyzed experiments on x86 with NAS BT and LULESH. Md. Abdullah Shahneous Bari and Abid M Malik ran and analyzed experiments on POWER 8 and modified the NAS Parallel Benchmarks to be compatible with runtime schedule selection. Kevin Huck is the lead developer of APEX and developed the OMPT interface for APEX. Barbara Chapman, Allen D. Malony, and Osman Sarood edited the paper.

7.1 Introduction

OpenMP is an extremely commonly used standard for specifying intra-node parallelism, which allows for otherwise serial code to be annotated with directive to indicate how it is to be parallelized. Certain directives have parameters which indicate how work is to be scheduled onto resources, which can have considerable performance and power implications. Traditional performance tools – that is, those without deep integration with the runtime – can collect correct and insightful performance performance data. However, actionability is constrained not by the output of the tool but by the limited ability for traditional tools to communicate results back to the runtime.

Directive parameters such as number of worker threads, scheduling policy, and chunk size are either specified at compile time – in which case they are unchangeable without recompiling the program – or are deferred to runtime, in which case they are set by the application itself or by environment variables. Setting environment variables to configure these parameters sets them for all parallel regions in the entire application, even though the parameter values that yield the best performance may vary between parallel regions within an application. Table 3 shows the parameter settings which produces the best performance for the listed parallel region. Note that the best settings vary considerably between regions within each application.

In this chapter, we show how APEX can be integrated with the OpenMP runtime, and design an APEX policy which performs online adaptation of work sharing region parameters on a per-region basis. We show that this can improve application performance, reduce power usage, or both.

The major contributions of this chapter are:

- We present an APEX framework, ARCS, that selects the best OpenMP runtime configurations for OpenMP regions to optimize HPC applications under a power constraint.
- To the best of our knowledge, ARCS is the first fully automatic framework that chooses OpenMP runtime configurations with no involvement of the application programmer.
- ARCS chooses and adapts OpenMP runtime configurations dynamically based on OpenMP region and underlying architecture characteristics, resulting in efficient execution on a number of applications under a power constraint across different architectures.

7.2 Motivation

OpenMP programming model is an integral part of many important HPC legacy codes in the form of hybrid programming models (e.g., - MPI + OpenMP).

Table 3. Per-parallel-region parameter settings with best performance for NAS OpenMP benchmarks.

Kernels	No. of Threads	Scheduling Policy	Chunk Size
BT_add_1	2	DYNAMIC	1
BT_compute_rhs_1	16	DYNAMIC	32
BT_error_norm_1	16	GUIDED	8
BT_exact_rhs_1	24	GUIDED	8
BT_initialize_1	16	DYNAMIC	1
BT_rhs_norm_1	16	STATIC	8
BT_x_solve_1	16	DYNAMIC	1
BT_y_solve_1	32	GUIDED	1
BT_z_solve_1	16	GUIDED	1
CG_conj_grad_1	8	GUIDED	8
CG_main_1	16	GUIDED	32
CG_main_2	16	STATIC	256
CG_main_3	24	STATIC	64
CG_main_4	8	STATIC	64
CG_main_5	16	STATIC	64
CG_main_6	16	STATIC	512
EP_main_1	2	DYNAMIC	512
EP_main_2	2	DYNAMIC	32
EP_main_3	32	GUIDED	1
FT_cffts1_1	16	DYNAMIC	1
FT_cffts2_1	16	DYNAMIC	1
FT_cffts3_1	16	STATIC	8
FT_checksum_1	4	STATIC	32
FT_compute_indexmap_1	16	STATIC	1
FT_compute_initial_conditions_1	24	DYNAMIC	1
FT_evolve_1	2	GUIDED	1
FT_init_ui_1	4	DYNAMIC	1
LU_erhs_1	32	GUIDED	8
LU_error_1	8	STATIC	8
LU_l2norm_1	32	GUIDED	32
LU_pintgr_1	8	STATIC	8
LU_rhs_1	16	DYNAMIC	1
LU_setbv_1	8	GUIDED	8
LU_setiv_1	24	GUIDED	1
LU_ssor_1	16	GUIDED	64
SP_add_1	2	GUIDED	32
SP_compute_rhs_1	16	GUIDED	32
SP_error_norm_1	16	DYNAMIC	8
SP_exact_rhs_1	16	GUIDED	1
SP_initialize_1	16	GUIDED	1
SP_ninvr_1	16	STATIC	32
SP_pinvr_1	16	STATIC	32
SP_rhs_norm_1	16	STATIC	8
SP_txinvr_1	32	GUIDED	32
SP_tzetar_1	24	GUIDED	32
SP_x_solve_1	8	GUIDED	1
SP_y_solve_1	16	GUIDED	8
SP_z_solve_1	8	GUIDED	1

Therefore, tuning an OpenMP code to get a better per node performance for a given power budget is an important research problem. In this section, we motivate a reader about the need of ARCS for power-constrained OpenMP applications. The need for ARCS like framework depends on the following questions:

- *Does the best configuration for a given OpenMP region remain same across different power levels and workloads?*
- *Does the performance gain due to the best configuration persist across all power caps?*

We took an OpenMP region from the SP benchmark application and ran it with different power levels or power caps[1] using different number of threads, scheduling policies, and chunk sizes (150 different configurations). The region belongs to the `compute_rhs` function, and has 11 different parallel loops, i.e., **`#pragma omp for`** OpenMP directives.

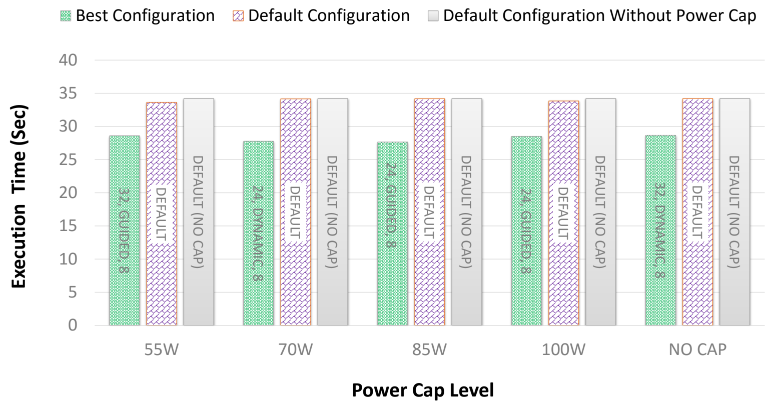


Figure 92. Execution time comparison for the **compute_rhs** region of SP using different OpenMP runtime configurations at different power levels. Smaller value is better. The function was run on Intel Sandy Bridge.

---

[1] We use the two words synonymously in this paper.

Figure 92 shows the comparison of execution time using the optimal configuration[2] and the default configuration at different power levels. The default configuration uses maximum number of available threads, static scheduling, and chunk sizes calculated dynamically by dividing total number of loop iterations by number of threads. The figure clearly shows that the optimal configuration is different from the default configuration at all the power levels. It also shows that the optimal configuration improves the execution time of the region up to 19% compared to the default configuration at the same power level. Also, we can see that the optimal configuration at a lower power level gives better execution time performance than the default configuration with maximum power level prescribed by the manufacturer or Thermal Design Power (TDP). For example, the optimal configuration at 70W power cap improves execution time by 15% as compared to the default configuration at TDP (115W in our case).

We also experimented with OpenMP regions from other NAS Parallel benchmark applications using different runtime configurations. We observed that a significant number of the OpenMP regions showed similar behavior. We observed these OpenMP regions to have poor load balancing and cache behavior with the default configuration. We also saw that these poor behaviors persist across different power levels and workloads for these kernels with the default configuration. As a result, irrespective of power level or workload size an optimal configuration always shows consistent performance improvement compared to the default configuration for these kernels. However, we observed that the optimal configurations for these kernels change across different power levels and workloads.

---

[2] The configuration that gives the best execution time.

In future HPC facilities, the load of applications may change dynamically. If the facility is working under a power constraint, the resource manager may add/remove number of nodes and adjust their power level dynamically. To get the best per node performance at each power level, the runtime configurations need to be changed dynamically. Our ARCS framework can do this efficiently.

7.3 Framework

The ARCS runtime is composed of two key software components. The first component is a modified OpenMP runtime. The second component is the APEX instrumentation and adaptation library. APEX integrates the Active Harmony search engine, integrated as part of the APEX library.

OpenMP runtime with OMPT. A broad group of interested parties has been working on extending the OpenMP specification to include a formal performance and debugging tool interface [67]. In order to provide support for both instrumentation (event-based) and sampling based tools, OMPT includes both events and states. The OMPT draft specification is complete and is available as a Proposed Draft Technical Report at the OpenMP Forum website [68]. The key OMPT design objectives are to provide low overhead observation of OpenMP applications and the runtime in order to collect performance measurements, provide stack frame support for sampling tools and incur minimal (near zero) overhead when not in use. OMPT specifies support for a large set of events and states, covering the OpenMP 4.0 standard. In addition, OMPT specifies additional insight into the OpenMP runtime in the form of data structures populated by the runtime itself. These data structures include the parallel region and task identifiers, wait identifiers and stack frame data. OMPT has been integrated into performance tools such as TAU [103] for providing detailed insight into OpenMP runtime behavior. From a tool developer perspective, the broad support and large set of events and states makes OMPT an attractive approach to access the OpenMP runtime performance state.

APEX. We have implemented a measurement and runtime adaptation library for asynchronous multitasking runtimes called *Autonomic Performance Environment for eXascale (APEX)* [107, 106]. The APEX environment supports both introspection and policy-driven adaptation for performance and power optimization objectives. APEX aims to enable autonomic behavior in software by providing the means for applications, runtimes, and operating systems to observe and control performance. Autonomic behavior requires both performance awareness (introspection), and performance control/adaptation. APEX can provide introspection from timers, counters, node- or machine-wide resource utilization data, energy consumption, and system health, all accessed in real-time. The introspection results are analyzed in order to provide the feedback control mechanism.

The most distinguishing component in APEX is the *policy engine*. The policy engine provides controls to an application, library, runtime, and/or operating system using the aforementioned introspection measurements. Policies are rules that decide on outcomes based on the observed state captured by APEX. The rules are encoded as callback functions that are periodic or triggered by events. The policy rules access the APEX state in order to request profile values from any measurement collected by APEX. The rules can change runtime behavior by whatever means available, such as throttling threads, changing algorithms, changing task granularity, or triggering data movement.

APEX was designed for use with runtimes based on the ParalleX [116] programming model, such as HPX [120] or HPX-5 [8]. However, the APEX design has proven to be flexible enough to be broadly applied to other thread-concurrent runtimes such as OpenMP.

APEX integrates the auto-tuning and optimization search framework *Active Harmony* [206]. In APEX, Active Harmony is directly integrated into the library to receive APEX performance measurements and suggest new parametric options in order to converge on an optimal configuration. Active Harmony implements several search methods, including exhaustive search, *Parallel Rank Order* and *Nelder-Mead*. In this work, we used the exhaustive and *Nelder-Mead* search algorithms. In our experiments, the *ARCS-Offline* method uses an exhaustive search to find the best configuration during one execution, then executes again with that optimal configuration. Only the second execution with the optimal configuration is measured. The *ARCS-Online* method uses the *Nelder-Mead* search algorithm to search for and use an optimal configuration in the same execution.

Prior to running the examples with the framework, the NPB 3.3-OMP-C OpenMP benchmarks were exhaustively parameterized to explore the full search space for the OpenMP environment variables `OMP_NUM_THREADS` and `OMP_SCHEDULE` (schedule type and chunk size). From that initial dataset, the search space was manually reduced. Unlike the initial parameter search, ARCS can tune the settings for each OpenMP parallel region independently. The reduced set of search parameters was used to limit the search space that had to be explored at runtime. The final ranges explored by ARCS are listed in Table 4.

Using the policy engine, we designed a policy to tune OpenMP thread count, schedule, and chunk size based upon the reduced search space described above. At program initialization, the policy registers itself with the APEX policy engine, and receives callbacks whenever an APEX timer is started or stopped. The

Table 4. Set of ARCS search parameters for OpenMP parallel regions.

Parameter	**Set of values**
Number of threads (Crill)	2, 4, 8, 16, 24, 32, default
Number of threads (Minotaur)	10, 20, 40, 80, 120, 160, default
Schedule Type	dynamic, static, guided, default
Chunk Size	1, 8, 16, 32, 64, 128, 256, 512, default

OMPT interface starts a timer upon entry to an OpenMP parallel region and stops that timer upon exit. When a timer is started for a parallel region which has not been previously encountered, the policy starts an Active Harmony tuning session for that parallel region. When a timer is stopped, the policy reports the time to complete the parallel region. When a timer is started for a parallel region which has been previously encountered, the policy sets the number of threads, schedule, and chunk size to the next value requested by the tuning session, or, if tuning has converged, to the converged values. When the program completes, the policy saves the best parameters found during the search. When the same program is run again in the same configuration in the future, the saved values can be used instead of repeating the search process.

Overhead. The main overhead of ARCS can be characterized into three different types.

- **Configuration changing overhead**: ARCS changes the runtime configuration each time a region is executed. To change these configurations, ARCS uses the OpenMP runtime library routine `omp_set_num_threads()` and `omp_set_schedule()`. Time consumed during these routine calls adds some extra overhead. We call this overhead *Configuration Changing overhead*. This overhead is present in both Online and Offline strategies. In Crill, we calculated this overhead to be about 0.0008 sec in each region call. If a region is large enough, this overhead becomes insignificant. However, if the the region time is not large enough this overhead can become a significant factor.

- **APEX instrumentation overhead**: Overhead incurred due to APEX runtime instrumentation. Just like *Configuration changing overhead*, the impact of this overhead is also dependent on the region execution time. It is present in both Online and Offline strategies.
- **Search overhead**: In the online search strategy, finding the optimal configuration requires ARCS to test several runtime configurations before converging. Many of these configurations are not optimal, and as a result these sub-optimal configurations incur extra execution time. This additional execution time can be termed as *Search overhead*. This overhead is only present in the Online strategy. It is not present in *Offline strategy*, because in *Offline strategy* ARCS does not search for the the optimal configuration, it reads it from the history file only once during the whole application lifetime. We observed this overhead to vary across regions based on how fast they converge to the optimal configuration. During our experimentation, we observed this overhead to reach as high as 10% of the total execution time.

7.4 Experiment Design

Test System. We evaluated our framework on two different systems, Crill and Minotaur. These systems differ in architecture, number of cores, memory size and power consumption.

Crill is a dual socket machine with two 2.4 GHz quad-core Intel® Xeon® E5-2665 processors (based on the Intel Sandy Bridge architecture).

It has a total of 16 cores (32 hyper-threaded threads) and 16 GB of memory. It runs on OpenSUSE 13.1 and has a TDP limit of 115W. Crill is from the University of Houston.

Our second test machine, *Minotaur* is hosted at the University of Oregon. It is an IBM® S822LC system equipped with two 10-core IBM POWER8® processors that operate at 2.92 GHz. It has support for 160 hardware threads (8 per core) and 256 GB of memory. It is running Ubuntu Linux, version 15.04.

Compiler & Libraries. We used GCC compiler version 4.9.2 with Intel OpenMP runtime for our experimentation. We also used libmsr[188], a library that facilitates access to MSRs via RAPL interface for energy measurement and power capping.

Benchmarks. We used three proxy applications, LULESH 2.0, BT and SP to evaluate ARCS. We selected these benchmarks because they exhibit performance and load balancing behavior typical for a broad range of HPC applications.

LULESH 2.0[123] is a shock hydrodynamics computational kernel from Lawrence Livermore National Laboratory. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It is built on the concept of an unstructured hex mesh. It is one of the most used proxy applications in the HPC area, and it shows excellent load balancing and cache behavior. We used mesh sizes of 45 and 60 for our experimentation.

BT is a simulated CFD computational kernel that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5 × 5 blocks and are solved sequentially along each dimension. This application shows good load balancing behavior. We used data set sizes B (102 × 102 × 102) and C (164 × 164 × 164) with custom 1000 time steps.

SP is a simulated CFD computational kernel that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension. It shows good load balancing behavior but poor cache behavior. For SP, we also used data set sizes B (102 × 102 × 102) and C (164 × 164 × 164) with custom 1000 time steps.

Both *BT* and *SP* are from from NAS parallel benchmark suite[196], version 3.3-OMP-C.

Experimental Details. We carried out extensive experiments to evaluate the impact of ARCS. We considered both the execution time and energy consumption during the evaluation. An optimal OpenMP runtime configuration for a region is dependent on the region's characteristics, power cap level, workload size, and architecture. For that reason, we designed our experiments in such a way that they cover all these scenarios. We tested ARCS on five different power levels, two different workloads, and two distinct architectures (Intel Sandy Bridge and IBM POWER8).

As mentioned before, our primary experimental resource Crill is equipped with Sandy Bridge processors, and our secondary resource Minotaur with POWER8 architecture. In Crill, we had power capping privilege and access to the energy counters. For that reason we were able to evaluate the impact of ARCS at different power levels. We experimented on 55W, 70W, 85W, 100W and 115W (TDP for this processor) power level. We only limited the processor power (package power).

A package consists of cores, caches and other internal circuitry. We used maximum power for other components (DRAM, Network card, etc.), because we did not have capping capability on these subsystems. We used RAPL for power capping and collecting energy information. We tried to tackle known issues of RAPL such as counter update frequency and the warm up period after enforcing a power cap during the experimentation to get reliable energy readings.

As Minotaur is a relatively new resource, we did not have energy counter access nor power capping privilege. Therefore all the experiments conducted on this machine were using the default (TDP) power level of this machine. Also, all the evaluation done on this machine is based on execution time only. We evaluated both Online and Offline ARCS strategies in the above-mentioned environments.

In this section we present our experimental results. Through these results we show the impact of ARCS on different types of OpenMP applications. As mentioned previously, we evaluated ARCS on three different OpenMP applications. These applications vary in scalability, load balancing, and cache behavior. *LULESH* is a well-balanced application with good cache behavior. *BT* is also fairly well balanced with good cache behavior. *SP* is well balanced but shows poor cache behavior. We mainly concentrated on scalability, load balancing and caching because these are the behaviors that impact OpenMP performance the most.

In an OpenMP application with loop level parallelism, these behaviors can be controlled by the number of threads, scheduling policy and chunk sizes. The number of threads has a significant impact on scalability while scheduling policy and chunk sizes are very important for good load balancing and cache behavior. These behaviors not only affect the execution time performance, but they also impact energy consumption. Load balancing and cache behavior of an application are two of the main factors that define an application's energy profile.

Applications with bad cache behavior tend to consume more energy[201]. If there is a cache miss, the system has to do the extra work of fetching the data from the next level of cache or memory and in the process use I/O path which leads to extra energy consumption.

On the other hand, load balancing affects the energy consumption in a different way. Poor load balancing of an application leads the cores to wait in idle states in the synchronization points (barriers). Lightly loaded threads wait for highly loaded threads to finish their work. Even though current processors do a decent job at saving energy by entering the sleep state while waiting, entering and exiting sleep states incurs non-trivial overheads and can cause negative savings if the idle duration is short[10]. In OpenMP regions, the waiting time is usually short. Therefore, improving the load balancing behavior is crucial to improving the energy profile of an OpenMP application. Not only that but also these behaviors impact an OpenMP application's power profile, as power is the ratio of the energy consumption and execution time.

Moreover, cores and caches are the main power consuming components of a processor[189]. The total power of a processor is divided between these two components. So when a power cap is imposed on a processor, it not only affects the performance of the cores but also impacts the cache performance. As a result, the load balancing and cache behavior also change with the change of the power cap.

Furthermore, these behaviors vary across different regions of an application. Therefore, choosing an optimal configuration (number of threads, scheduling policy, and chunk sizes) for each regions separately is no trivial task. But we show through extensive analysis that ARCS is able to do this job very proficiently.

In the following discussion, we analyze each application separately. We show that ARCS can potentially improve performance across different types of applications. We also demonstrate the effect of ARCS strategies at both application and region level using detailed analysis of dynamic features. We show the performance behavior across different power caps and different workload sizes. Finally, we show the ARCS performance across different architectures.

We compare the performance of ARCS strategies with the default configuration. The default configuration uses maximum number of available threads, static scheduling, and chunk sizes calculated dynamically by dividing total number of loop iterations by number of threads. We concentrate on both online and offline strategies for ARCS. Results shown here is based on Crill, unless mentioned otherwise. The same applies for the power cap; if nothing is mentioned, that means we are using the highest power cap (TDP).

SP. *SP* is an application which shows a good load balancing behavior and poor cache behavior with the default configuration. SP consists of 13 loop based OpenMP regions. However, almost 75% of it's execution time is spent on four regions (`compute_rhs`, `x_solve`, `y_solve and z_solve`). Among them, `compute_rhs` has a poor load balancing and cache behavior, `x_solve, y_solve` and `z_solve` regions have good load balancing behavior but show poor cache behavior. To improve these regions' performance, their load balancing and cache behavior has to be improved. Therefore, we need to find configurations that improve the load balancing and cache behavior of these regions. To find such configurations we applied ARCS on this application. Table 5 shows the optimal configuration chosen by ARCS-Offline strategy for these regions at TDP power.

Table 5. Optimal configuration chosen by ARCS-Offline strategy for SP regions.

Region	**Optimal Configuration (Thread, Schedule, Chunk)**
`compute_rhs`	16, guided, 8
`x_solve`	16,guided, 1
`y_solve`	8, static, default
`z_solve`	4, static, 32

In Figure 93 we show the feature comparison between the default configuration and the configurations chosen by *ARCS-Offline*, the best ARCS strategy. We compare the L1 cache miss rate in Figure 93a, L2 cache miss rate in Figure 93b, L3 cache miss rate in Figure 93c and OpenMP barrier (`OMP_BARRIER`) time in Figure 93d. The L1, L2 and L3 cache miss rates show the cache behavior of these regions. The OMP_BARRIER time shows the load balancing behavior; greater OMP_BARRIER time is a symptom of poor load balancing. For all of these metrics, lower values indicate better performance.

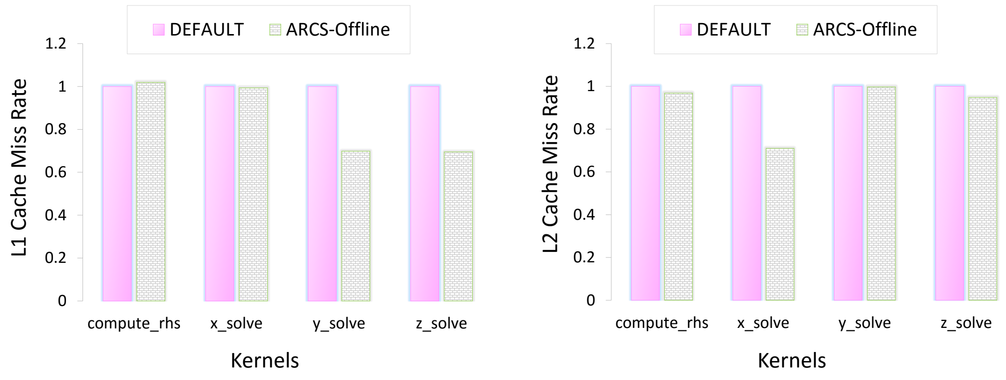
From these figures, we observe that all four regions show better cache and load balancing behavior with the ARCS strategy. Using the configuration chosen by ARCS, the OMP_BARRIER time is decreased by more than 50% in all four regions compared to the default configuration, shown in Figure 93d. The best improvement, which is more than 80% is achieved in the `z_solve` region while a relatively smaller improvement (around 50%) is achieved in `compute_rhs`.

We also observed L1, L2 and L3 cache miss rate improvement. Although L1 and L2 cache behaviors show good improvement, the biggest improvement (up to 90%) is visible in L3 cache behavior. This is important for performance because L3 cache misses have the highest cache miss penalty. The improvement also shows that these configurations enabled different cores to maximize their use of the shared L3 cache.

The above analysis shows that ARCS strategies can improve the cache behavior and load balancing of SP regions. This leads to the question: how much do these improvements affect the overall application's execution time and energy consumption? In Figure 94 we show the execution time and energy consumption comparisons between the default and ARCS strategies (*ARCS-Online* and *ARCS-Offline*). We show the results for five different power levels. We compare both execution time (in Figure 94a) and energy consumption (in Figure 94b). In Figure 94a we see that all the strategies in all five power levels outperform the default configuration by a large margin. The improvement varies between 26-40%. We observe similar behavior in energy consumption, shown in Figure 94b with the highest improvement touching 40% limit.

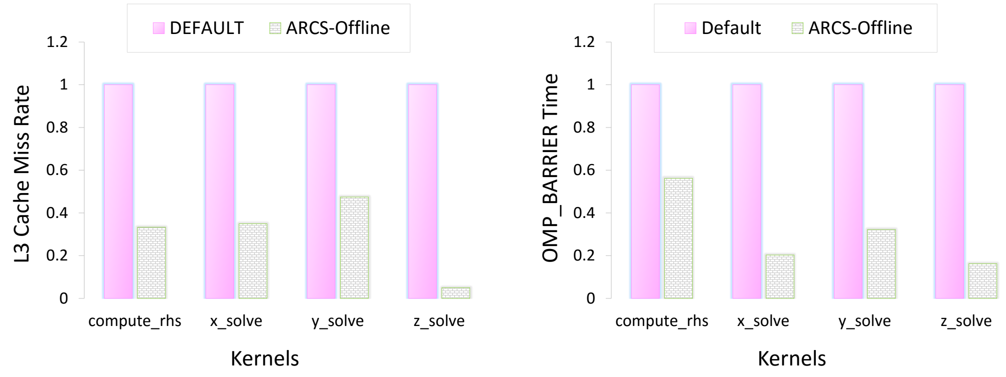
We were able to achieve so much improvement using ARCS because most of these time-consuming regions have a slight load imbalance and poor cache behavior. However there are applications which may have a very good load balance and cache behavior. In those kind of applications, the improvement will likely not be that significant, because there is very little room for ARCS to work on. In the later part of this section, we will look into such applications as well.

We discussed in Section 7.2 that the behavior of a region changes across different workloads. To see how efficient ARCS in choosing optimal configurations across workloads, we used ARCS on data set C of SP. Dataset C is four times larger than data set B. Figure 95 shows the execution time and energy consumption improvement at TDP (highest power cap). Even in this workload, we achieve execution time improvement of up to 40% and energy consumption improvement of



(a) L1 cache miss rate

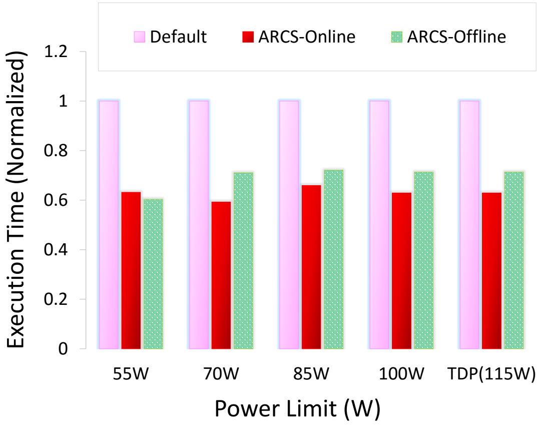
(b) L2 cache miss rate



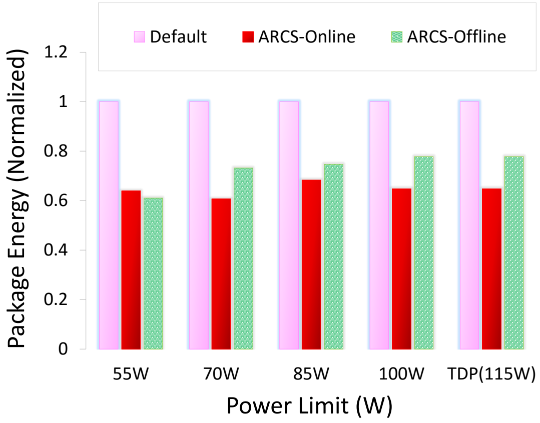
(c) L3 cache miss rate

(d) OMP_BARRIER time

Figure 93. Feature comparison between the default and *ARCS-Offline* strategy at TDP power level. Comparison is done on four of the most time consuming regions of SP. Y-axis shows the normalized feature value. Smaller value is better.



(a) Execution Time



(b) Package energy

Figure 94. Application level execution time and package energy comparison among the default and ARCS strategies in SP at data set B. Comparison is done on five different power levels. Smaller value is better.

up to 42% using ARCS strategies. It shows that ARCS can find optimal configurations across different workloads. We also observed that the configurations of the regions from SP differed across workloads which also proves the claim we made in Section 7.2. To validate ARCS's consistency across different architectures, we used ARCS on a new architecture, IBM POWER8 (Minotaur). Minotaur differs significantly compared to Crill. Even so, when we ran SP with data set B in Minotaur, we observed 37% execution time improvement compared to the default strategy. This result demonstrates ARCS's versatility across architectures.

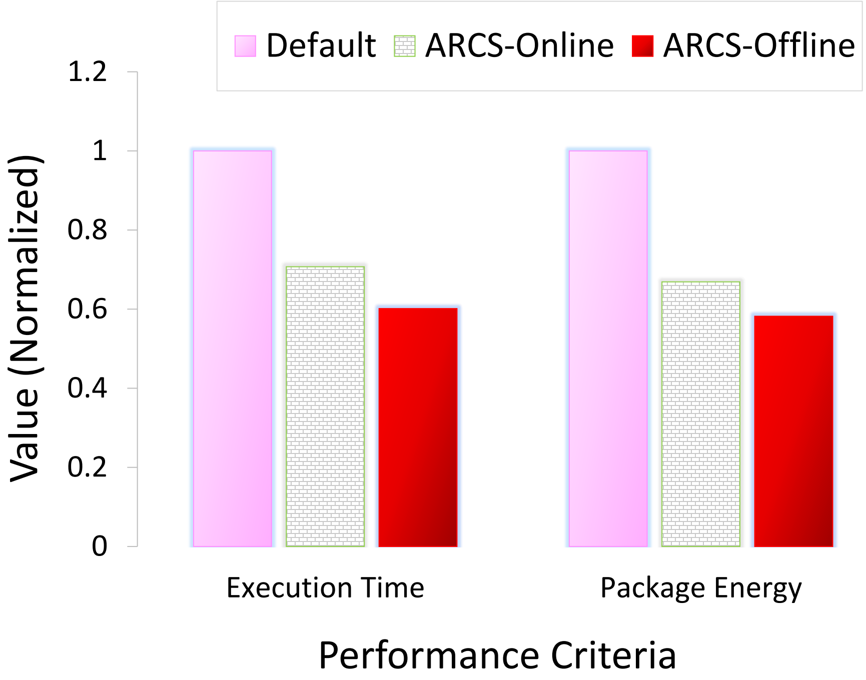


Figure 95. Execution time and energy consumption comparison of ARCS strategies and the default strategy in data set C of SP. Smaller value is better.

BT. *BT* is an application with good load balancing and cache behavior. BT is very similar to SP in structure although the approximate factorization is different. Like SP, majority of its execution time is also dependent on four regions (`compute_rhs`, `x_solve`, `y_solve` and `z_solve`). However, the behavior of these regions is slightly different. Three of these regions (`x_solve, y_solve` and `z_solve`) show very good load balancing and cache behavior in the default configuration. Only `compute_rhs` shows poor scaling, load balancing, and cache behavior. As a result, ARCS has a limited opportunity to improve the performance of this application. `compute_rhs` is the only region where ARCS strategies can have a significant effect, as all other regions already perform very well with the default strategy. In addition, `compute_rhs` is algorithmically hard to optimize due to its long stride memory access. Specifically, the second-order stencil operation in `rhsz` uses the $K \pm 2$, $K \pm 1$ and K elements of the solution array to compute RHS for the z direction:

$$\begin{aligned} RHS(I,J,K) = {} & A * U(I,J,K-2) + \\ & B * U(I,J,K-1) + C * U(I,J,K) + \\ & D * U(I,J,K+1) + E * U(I,J,K+2) \end{aligned}$$

Such memory accesses are not cache friendly, so finding an optimal configuration for such a region is not trivial. However, ARCS does a very good job in finding an optimal configuration (24, guided, 1) for `compute_rhs` that improves the `OMP_BARRIER` and cache behavior of the region. The comparison between the *ARCS-Offline* and default strategy is shown in Figure 96. We compare the cache (L1, L2 and L3 cache miss rate) and load balancing (OMP_BARRIER time) behavior. We are only showing the result for `compute_rhs` region, because in other regions the improvement is negligible. For `compute_rhs`, the ARCS configuration shows a significant load balancing behavior improvement which is demonstrated by 80% OMP_BARRIER time improvement. It also shows good L3 cache miss rate improvement indicating better cache utilization among different cores.

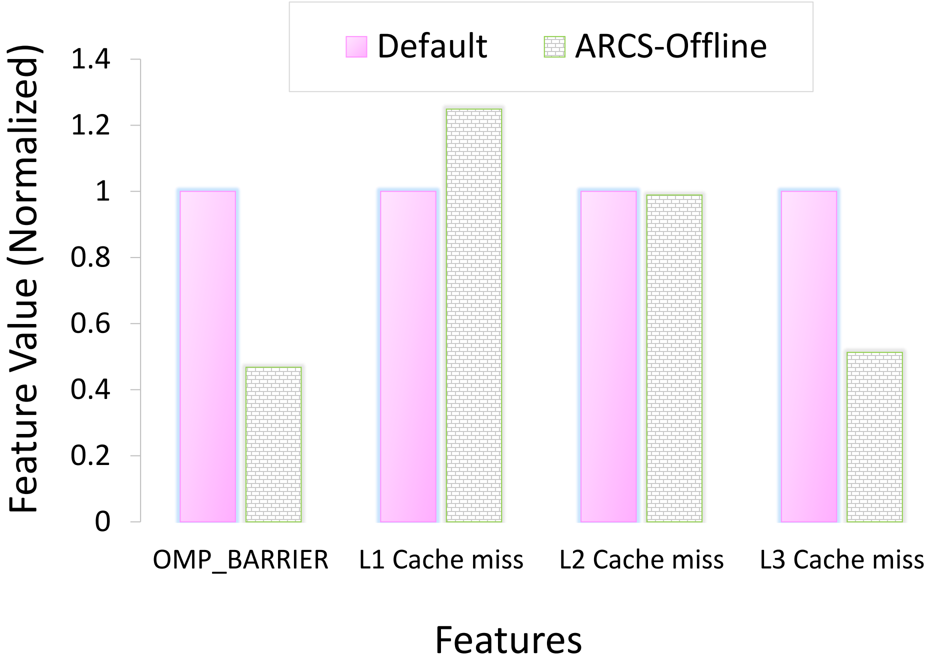


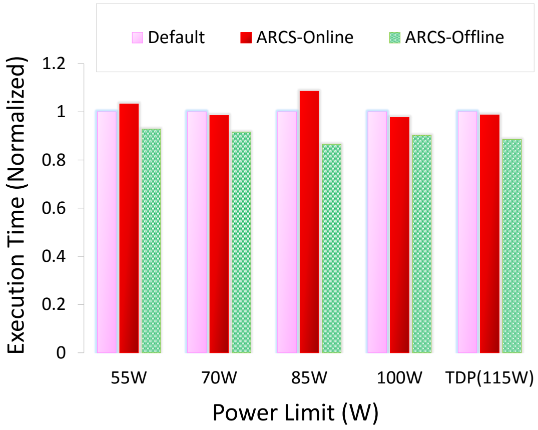
Figure 96. Feature comparison between the default and *ARCS-Offline* strategy at TDP power level for `compute_rhs` region of BT. Smaller value is better.

The impact of these behaviors is also visible in the overall application level execution time and energy consumption comparison in Figure 97. Here, we compare the execution time(97a) and energy consumption(97b) among the default and ARCS strategies(*ARCS-Online and ARCS-Offline*). We show the results for all five power levels. We observe that the execution time improvement is small across all power levels, with the highest improvement recorded is 13% at 85W power cap with *ARCS-Offline* strategy. In some cases ARCS actually performs worse than the default strategy (e.g., ARCS-Online at 85W). This is because in those cases small improvement achieved by ARCS is offset by the overhead. Similar behavior is visible for package energy in Figure 97b.

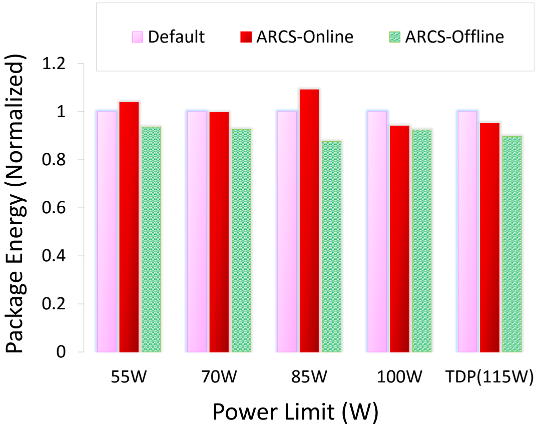
We also observed similar trend at Power8 architecture. Only the *ARCS-Offline* strategy was able to achieve an application level improvement of 18%.

LULESH 2.0. In Figure 98 we show the comparison of execution time and energy consumption comparison between the default strategy and *ARCS-Online* and *ARCS-Offline* strategies on both Crill and Minotaur. In Minotaur, We achieved a 40% execution time improvement using the *ARCS-Offline* strategy, while with *ARCS-Online* we achieved around a 4% improvement.

However, in Crill, the improvement is not evident. With *ARCS-Offline* strategy, we achieved about 3% execution time improvement in the smallest (55W) and the highest (115W) power levels. However, we lost performance on other three power levels. We achieved energy consumption improvement in all five power levels with maximum of 26% coming in 85W power level. As for *ARCS-Online* strategy, we observed a degradation in both execution time and energy consumption for every power levels as compared to default.



(a) Execution Time



(b) Package energy

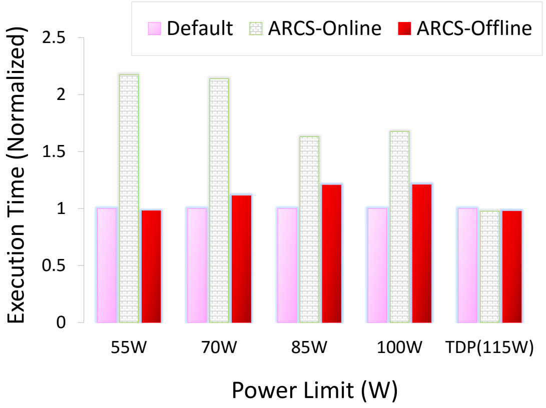
Figure 97. Application level execution time and package energy comparison among the default and ARCS strategies in BT with data set B. Comparison is done on five different power levels. Smaller value is better.

To understand why ARCS is performing poorly with LULESH on Crill, we did an extensive analysis. We used TAU [192] for our analysis. We profiled LULESH running with the default configuration at the highest power cap. In Figure 99 we show the top five regions based on total time (inclusive time). Through three OMPT events we show how these regions spent their time. These OMPT events are,

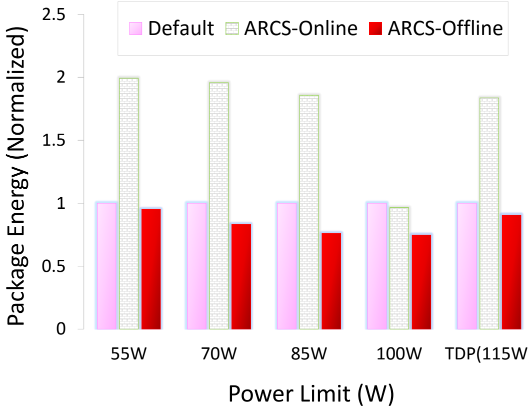
- `OpenMP_IMPLICIT_TASK`, it reports the total time spent by an implicit task, in other words it shows the overall execution time of the region.
- `OpenMP_LOOP` reports the execution time that is spent only on the loop body.
- `OpenMP_BARRIER/OMP_BARRIER` reports the time spent on the implicit and explicit barriers.

We observe from Figure 99 that in terms of `OpenMP_IMPLICIT_TASK` the most time consuming region is `EvalEOSForElems_1`. But most of its time is spent on `OpenMP_BARRIER`. Only a small portion of time is spent on real computation which can be attributed by `OpenMP_LOOP` time. The same applies for the `CalcPressureForElems_1` region. Both of these regions have a very small execution time per region call, `EvalEOSForElems_1` with 0.000828 sec and `CalcPressureForElems_1` with 0.000139 sec. And as we explained in the Overhead section, for each region run ARCS has a **Configuration changing overhead** of around 0.0008 sec. For these regions this overhead becomes a huge issue. In fact the overhead becomes almost 100% and 600%. Combined with APEX instrumentation overhead, ARCS loses a significant amount of performance in these tiny regions and in the process adds a fair amount of extra execution time.

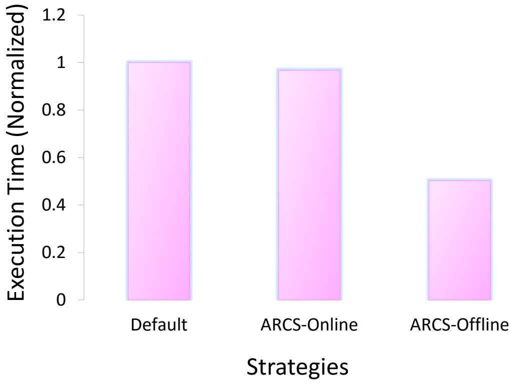
As for other three regions in Figure 99, although they have reasonable region time (execution time per region call), `CalcKinematicsForElems_1` and



(a) Execution time(Crill)



(b) Package energy (Crill)



(c) Execution time (Minotaur)

Figure 98. Application level execution time and package energy comparison among the default and ARCS strategies in LULESH, for mesh size 45. It shows results in both architectures. Smaller value is better.

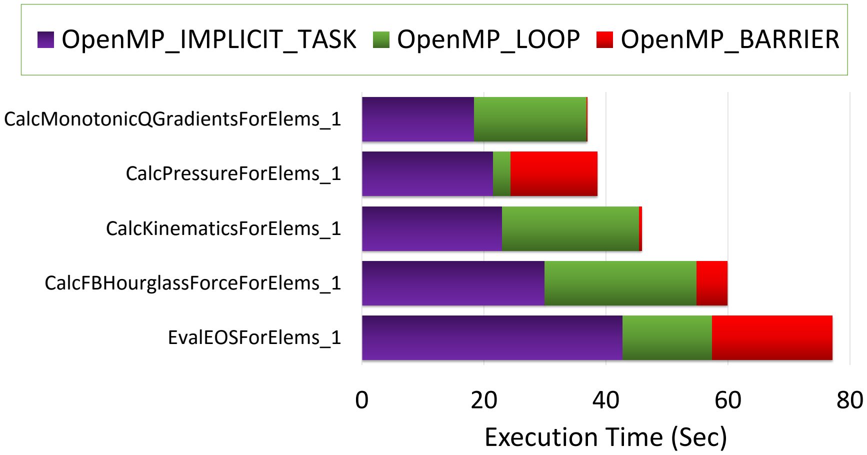


Figure 99. OpenMP events data for top 5 time consuming regions from LULESH.

`CalcMonotonicQGradientsForElems_1` show near perfect load balancing behavior with only 1.8% and 0.26% of their total execution time spent in `OpenMP_BARRIER`. So there is not much ARCS can do to improve these regions' performance. However, the `CalcFBHourglassForceForElems_1` region shows slightly worse load balancing behavior with 16% of its total execution time spent in `OpenMP_BARRIER`, so ARCS can have some impact on its performance. ARCS was able to do so, which is evident in Figure 100. The figure shows OpenMP_BARRIER, L1, L2 and L3 cache miss rate comparison between the default and *ARCS-Offline* strategy on CalcFBHourglassForceForElems_1 region. From the figure we can see that the configuration (4, guided, 32) chosen by the *ARCS-Offline* strategy is able make the OpenMP_BARRIER time almost zero. It also shows that the configuration also improved the L1 and L3 cache miss rate significantly.

But execution time improvement from just this region was not enough to offset the overhead incurred by those tiny regions in Crill. However, these overheads are not energy hungry computation, that's why we still achieved overall energy improvement in all power levels.

As for Minotaur, we achieved execution time improvement for the following reason: Minotaur can support up to 160 threads without oversubscribing, which causes a bit more load imbalance in larger regions. As a result, ARCS improvement in those regions overcomes the overhead incurred by the smaller ones, which in turn results in overall application level improvement.

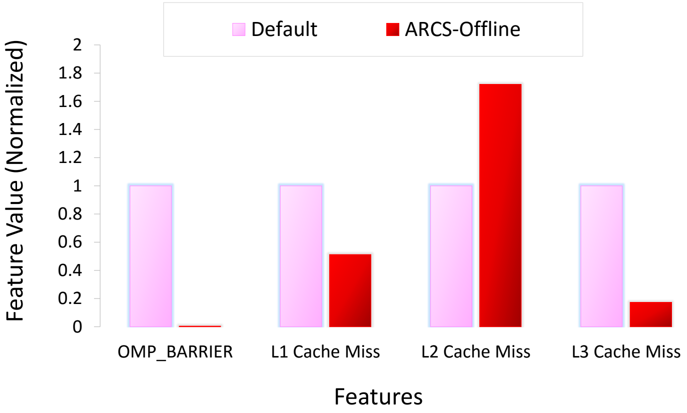


Figure 100. Feature comparison among default and ARCS strategies on `CalcFBHourglassForceForElems_1` region.

7.5 Related Work

The paper by Bull et al. [30] is one of the first which provides an insight into the choices of the number of threads, scheduling policy and synchronization on an OpenMP application's performance. They show that selecting the best number of runtime parameters is not a trivial task as different applications behave differently. Suleman et al. [202] proposed a framework to dynamically control the number of threads using run-time information. It uses Feedback-Driven Threading (FDT) to implement Synchronization Aware Threading (SAT), which predicts the optimal number of threads using the amount of data synchronization. However, neither of these works consider power or energy consumption in their analysis, only execution time.

As the number of threads and processor frequency have a significant impact on performance and energy consumption of a given OpenMP application, many researchers have studied energy efficient performance prediction models for parallel applications. The work by Curtis-Maury et al. [53, 54] falls under this category. They employ dynamic voltage and frequency scaling (DVFS), dynamic concurrency throttling (DCT) and simultaneous multithreading (SMT) to implement various online and offline configuration selection strategies for OpenMP applications. Their main goal was to decrease energy consumption without losing execution time. However, the work does not consider power budget. Peter Baily et al. [15] implemented an adaptive configuration selection scheme for both homogeneous and heterogeneous power constrained systems. It considers only two parameters – number of threads and processor frequency. Although the system selects these parameters for a given power budget, more than 10% of the time it violates the given power budget. The approach is not useful for a system working under a strict power budget. Dong Li et al. [134, 133] used DVFS and DCT to select energy efficient configurations for threads and operating frequency for MPI/OpenMP hybrid applications. They also did not consider a power budget. Their main target was to save energy without losing execution time. The work by Wei et al. [225] shows the impact of optimal operating frequency on energy consumption improvement for parallel loops. It uses different operating frequency across different loops using frequency modulation techniques. In contrast to these works, ours concentrates on a complete set of runtime parameters on a strict power constrained system.

Power has become a limiting factor for large scale HPC centers. As a result, research on over-provisioned systems with a strict power budget is gaining popularity in the HPC community. Work by Rountree et al. [185] is one of the first to explore the impact of power capping. They investigate how different power levels impact the performance of different types of applications. Work by Patki et al. [173] explores the impact of hardware over-provisioning on a system level performance. The main contribution of their work was to select the number of nodes, number of cores per node, and power cap per node. Work by Aniruddha et al. [149] and Bailey et al. [16] consider only two parameters, DVFS and number of threads, as configuration options. They focus on overall system level performance on a MPI/OpenMP hybrid application. Compared to these works, our work concentrates on single node OpenMP performance given a power budget to that node.

7.6 Conclusions

Application power budgeting with over-provisioned systems is becoming an attractive solution to handle the power challenge in future HPC platforms. Previous work in this area only looks at distributed programming models. However, intra-node performance at different power levels is also important. OpenMP API is mostly used to exploit parallelism for shared memory processors. In this paper, we presented the ARCS framework that selects the best run-time configurations under imposed power constraints for OpenMP applications. Our framework handles a larger configuration search space as compared to prior work. We show that our framework is practical with varying data sets as well as architectures. We tested ARCS using three proxy applications, SP, BT and LULESH. We show that for a given power level, efficient OpenMP runtime parameter selection can improve the execution time and energy consumption of an application up to 40% and 42% respectively.

In future work, we plan to improve ARCS to enable selective tuning for OpenMP regions to avoid overheads on the smaller regions. We also intend to account for memory power in addition to processor power. Currently, we are not looking into the DVFS (Dynamic Voltage Frequency Scaling) strategy. We plan to include this policy in the future. We also aim to extend the power management policy of the framework for heterogeneous nodes.

7.7 Bridge

This chapter has described a tool-runtime integration with OpenMP, showing how such an integration can enable online tuning of thread scheduling parameters under varying power constraints. In the final chapter, we will review the tool-runtime integrations performed in this work and the differences between the runtimes, as well as describe potential future integrations and integration use cases.

CHAPTER VIII

CONCLUSIONS

This document has described the integration of performance monitoring tools with several runtimes, and demonstrated how this allows access to performance-relevant information which would not be available absent the integration. For UPC, I showed how a tool, THOR, can capture network flow data and use this to dynamically adjust communication policies to account for different application-level communication patterns. For HPX, I showed how its integration with APEX allows for user-definable policies which dynamically adjust task granularity and/or keep power below a bound. For Apache Spark, I showed how runtime-tool integration allows understanding of the storage implications of application-level directives, thereby allowing the diagnosis of a severe performance problem with Spark on HPC systems, and allowing user-definable policies to handle storage decisions. For OCR, I showed how its integration with APEX allows for automatic diagnosis of causes of worker idleness, providing actionable suggestions to the application developer, and how user-definable policies can be used to mitigate load imbalance in an inherently load-imbalanced application. Finally, for OpenMP, I showed that even in traditional runtimes, integration between the runtime and a performance tool can provide benefits, by developing a policy for online tuning of OpenMP parallel region scheduling parameters on a per-region basis.

While I have shown that tool-runtime integration is an extremely promising technique for understanding the performance of emerging many-task runtimes, work in this area is still early, and there is more to be done.

An important limitation of the work described in this document is that it is entirely performed on microbenchmarks and mini-applications. Because full applications have traditionally been designed almost exclusively with MPI, there are very few full application codes available for many-task runtimes. HPX has only very recently had a full application developed for it (OctoTiger), while OCR currently has none. Apache Spark has many real applications, but these are not designed to scale to large numbers of cores. Once scalable, full applications become available for many-task runtimes, the results described in this dissertation will need to be revalidated on those applications. Early results with OctoTiger suggest that APEX scales well in the context of that application.

The many-task runtime community is still in a very experimental mode, and it is not yet clear what combinations of features will exist in runtimes which will be in actual use on future exascale systems. Future systems may require modifications to the tools described in this dissertation in order to accommodate features of those runtimes. An important example of this is Legion [21], a library and programming model which provides a much higher degree of data abstraction than any of the runtimes evaluated here. While we believe that APEX can be extended to support all features of Legion, this is yet to be done.

Finally, the runtimes described in this dissertation are primarily focused on CPUs and CPU-like accelerators such as the Xeon Phi. They do not provide special features for task executions on GPUs, as GPU architectures which have existed up until now have been very poorly suited to many-task runtimes, as tasks by their very nature will not run in lockstep. Newly announced GPU architectures from NVIDIA relax the requirement for GPU threads to run in lockstep, making tasking models much more feasible for GPU codes. If this occurs, new techniques will need to be evaluated for incorporating accelerators into the APEX model.

REFERENCES CITED

[1] Bilge Acun et al. “Parallel Programming with Migratable Objects: Charm++ in Practice”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, Louisana: IEEE Press, 2014, pp. 647–658. ISBN: 978-1-4799-5500-8. DOI: `10.1109/SC.2014.58`. URL: `http://dx.doi.org/10.1109/SC.2014.58`.

[2] Laksono Adhianto et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[3] Inc. Advanced Micro Devices. *AMD GPU Performance API User Guide*. 2015. URL: `http://developer.amd.com/wordpress/media/2013/12/GPUPerfAPI-UserGuide-2-15.pdf`.

[4] Sadaf R. Alam et al. “Parallel I/O and the metadata wall”. In: *Proceedings of the sixth workshop on Parallel Data Storage*. ACM, 2011, pp. 13–18. URL: `http://dl.acm.org/citation.cfm?id=2159356` (visited on 11/11/2015).

[5] Allan Porterfield et al. *Adaptive Scheduling Using Performance Introspection*. TR-12-02. RENCI, 2012. URL: `http://www.renci.org/technical-reports/tr-12-02/` (visited on 05/01/2014).

[6] Saman Amarasinghe et al. “Exascale programming challenges”. In: *Report of the 2011 Workshop on Exascale Programming Challenges, Marina del Rey*. 2011.

[7] Vinay C Amatya. “Parallel Processes in HPX: Designing an Infrastructure for Adaptive Resource Management”. PhD thesis. Louisiana State University,

2014. URL: `http://etd.lsu.edu/docs/available/etd-11172014-122205/unrestricted/amatya_dissertation_1.pdf`.

[8] Matthew Anderson et al. “A dynamic execution model applied to distributed collision detection”. In: *Supercomputing*. Springer. 2014, pp. 470–477.

[9] Michael Armbrust et al. “Spark SQL: Relational data processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394. URL: `http://dl.acm.org/citation.cfm?id=2742797` (visited on 06/17/2015).

[10] Manish Arora et al. “Understanding idle behavior and power gating mechanisms in the context of modern benchmarks on CPU-GPU Integrated systems”. In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 366–377.

[11] Cédric Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. English. In: *Euro-Par 2009 Parallel Processing*. Ed. by Henk Sips, Dick Epema, and Hai-Xiang Lin. Vol. 5704. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 863–874. ISBN: 978-3-642-03868-6. DOI: `10.1007/978-3-642-03869-3_80`. URL: `http://dx.doi.org/10.1007/978-3-642-03869-3_80`.

[12] E. Ayguade et al. “The Design of OpenMP Tasks”. In: *Parallel and Distributed Systems, IEEE Transactions on* 20.3 (Mar. 2009), pp. 404–418. ISSN: 1045-9219. DOI: `10.1109/TPDS.2008.105`.

[13] Shivnath Babu and Lance Co Ting Keh. “Better Visibility into Spark Execution for Faster Application Development”. In: *Spark Summit*. 2015.

[14] D. H. Bailey et al. "The NAS Parallel Benchmarks – Summary and Preliminary Results". In: *Supercomputing*. Albuquerque, New Mexico, USA: ACM, 1991, pp. 158–165. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125925. URL: `http://doi.acm.org/10.1145/125826.125925`.

[15] Peter E Bailey et al. "Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems". In: *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE. 2014, pp. 371–380.

[16] Peter E Bailey et al. "Finding the limits of power-constrained application performance". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 79.

[17] P. Balaji et al. "MPI on Millions of Cores". In: *Parallel Processing Letters (PPL)* 21.1 (Mar. 2011), pp. 45–60.

[18] Prasanna Balaprakash, R. Gramacy, and S. Wild. *Active-Learning-Based Surrogate Models for Empirical Performance Tuning*. 2013.

[19] Richard F Barrett, Courtenay T Vaughan, and Michael A Heroux. *MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing*. Tech. rep. SAND2012-10431. 2011. URL: `http://prod.sandia.gov/techlib/access-control.cgi/2012/122437.pdf`.

[20] Protonu Basu et al. "Towards making autotuning mainstream". In: *International Journal of High Performance Computing Applications* 27.4 (Nov. 1, 2013), pp. 379–393. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342013493644. URL: `http://hpc.sagepub.com/content/27/4/379` (visited on 05/01/2014).

[21] Michael Bauer et al. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11. ISBN: 978-1-4673-0804-5. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389086`.

[22] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. "Periscope: An online-based distributed performance analysis tool". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 1–16. URL: `http://link.springer.com/chapter/10.1007/978-3-642-11261-4_1` (visited on 08/05/2015).

[23] Katharina Benkert and Edgar Gabriel. "Empirical Optimization of Collective Communications with ADCL". In: *High Performance Computing on Vector Systems 2010*. Ed. by Michael Resch et al. Springer Berlin Heidelberg, 2010, pp. 37–49. ISBN: 978-3-642-11850-0, 978-3-642-11851-7. URL: `http://link.springer.com/chapter/10.1007/978-3-642-11851-7_3` (visited on 05/01/2014).

[24] Filip Blagojević et al. "Hybrid PGAS Runtime Support for Multicore Nodes". In: *Conference on Partitioned Global Address Space Programming Model*. PGAS '10. 2010.

[25] D. Bohme, F. Wolf, and M. Geimer. "Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications". In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. Parallel and Distributed Processing Symposium

Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International. May 2012, pp. 2538–2541. DOI: `10.1109/IPDPSW.2012.321`.

[26] Dan Bonachea. *GASNet Specification, v1.1*. Tech. rep. CSD-02-1207. University of California at Berkeley, Oct. 2002.

[27] Peter J. Braam et al. *The Lustre storage architecture*. 2004. URL: `http://idning-paper.googlecode.com/svn/trunk/reference/Luster/The_Lustre_Storage_Architecture.pdf` (visited on 11/11/2015).

[28] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2014. URL: `http://tools.ietf.org/html/rfc7159.html` (visited on 06/26/2015).

[29] Zoran Budimlić et al. "Concurrent collections". In: *Scientific Programming* 18.3-4 (2010), pp. 203–217.

[30] J Mark Bull. "Measuring synchronisation and scheduling overheads in OpenMP". In: *Proceedings of First European Workshop on OpenMP*. Vol. 8. 1999, p. 49.

[31] *Berkeley UPC*. `http://upc.lbl.gov`.

[32] Jong-Ho Byun et al. "Autotuning sparse matrix-vector multiplication for multicore". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-215* (2012). URL: `http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2012-215.pdf` (visited on 07/10/2015).

[33] Rosario Cammarota et al. "Optimizing Program Performance via Similarity, Using a Feature-Agnostic Approach". In: *Advanced Parallel Processing Technologies*. Ed. by Chenggang Wu and Albert Cohen. Lecture Notes in Computer Science 8299. Springer Berlin Heidelberg, 2013, pp. 199–213.

ISBN: 978-3-642-45292-5, 978-3-642-45293-2. URL: `http://link.springer.com/chapter/10.1007/978-3-642-45293-2_15` (visited on 05/02/2014).

[34] F. Cappello and D. Etiemble. “MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks”. In: *Supercomputing*. Nov. 2000, pp. 12–12.

[35] P. Carns et al. “Small-file access in parallel file systems”. In: *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*. IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009. May 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161029.

[36] Mohamad Chaarawi et al. “A Tool for Optimizing Runtime Parameters of Open MPI”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. Lecture Notes in Computer Science 5205. Springer Berlin Heidelberg, 2008, pp. 210–217. ISBN: 978-3-540-87474-4, 978-3-540-87475-1. URL: `http://link.springer.com/chapter/10.1007/978-3-540-87475-1_30` (visited on 05/01/2014).

[37] Nicholas Chaimov, Boyana Norris, and Allen Davis Malony. “Toward Multi-target Autotuning for Accelerators”. In: *International Conference on Parallel and Distributed Systems*. 2014.

[38] Nicholas Chaimov et al. “Exploiting communication concurrency on high performance computing systems”. In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2015, pp. 132–143.

[39] Nicholas Chaimov et al. “Performance Evaluation of Apache Spark on Cray XC Systems”. In: *Cray Users Group 2016*. 2016.

[40] Nicholas Chaimov et al. "Reaching bandwidth saturation using transparent injection parallelization". In: *International Journal of High Performance Computing Applications* (2016), p. 1094342016672720.

[41] Nicholas Chaimov et al. "Scaling Spark on HPC Systems". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 2016, pp. 97–110.

[42] Nicholas Chaimov et al. "Scaling Spark on Lustre". In: *International Conference on High Performance Computing*. Springer International Publishing. 2016, pp. 649–659.

[43] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442. eprint: `http://hpc.sagepub.com/content/21/3/291.full.pdf+html`. URL: `http://hpc.sagepub.com/content/21/3/291.abstract`.

[44] Kavitha Chandrasekar et al. "Task characterization-driven scheduling of multiple applications in a task-based runtime". In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM. 2015, pp. 52–55.

[45] Philippe Charles et al. "X10: An Object-oriented Approach to Non-uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN:

1-59593-031-0. DOI: 10.1145/1094811.1094852. URL: http://doi.acm.org/10.1145/1094811.1094852.

[46] P. Charles, C. Donawa, K. Ebcioglu, et al. “X10: An Object-Oriented Approach to Non-Unifrom Cluster Computing”. In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’05)*. Oct. 2005.

[47] Sanjay Chatterjee et al. “Integrating Asynchronous Task Parallelism with MPI”. In: *Parallel and Distributed Processing Symposium, International (IPDPS)* (2013).

[48] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Utah, 2008.

[49] Guojing Cong et al. “A Systematic Approach toward Automated Performance Analysis and Tuning”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.3 (Mar. 2012), pp. 426–435. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.189.

[50] *Cori Phase 1*. https://www.nersc.gov/users/computational-systems/cori/.

[51] Chuck Cranor, Milo Polte, and Garth Gibson. *HPC computation on Hadoop storage with PLFS*. Tech. rep. CMU-PDL-12-115. Carnegie Mellon University, 2012.

[52] Tom Crowe, Nathan Lavender, and Stephen Simms. “Scalability Testing of DNE2 in Lustre 2.7”. In: *Lustre Users Group*. 2015.

[53] Matthew Curtis-Maury et al. "Online strategies for high-performance power-aware thread execution on emerging multiprocessors". In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 2006, 8–pp.

[54] Matthew Curtis-Maury et al. "Prediction models for multi-dimensional power-performance optimization on many cores". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 250–259.

[55] Tomasz S Czajkowski et al. "From OpenCL to high-performance hardware on FPGAs". In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE. 2012, pp. 531–534.

[56] Georges Da Costa et al. "Exascale Machines Require New Programming Paradigms and Runtimes". In: *Supercomputing Frontiers and Innovations* 2 (2015), pp. 6–27.

[57] Aaron Davidson and Andrew Or. *Optimizing Shuffle Performance in Spark*. UC Berkeley Tech. Report.

[58] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782. DOI: `10.1145/1327452.1327492`. URL: `http://doi.acm.org/10.1145/1327452.1327492` (visited on 06/17/2015).

[59] James Dinan et al. "Enabling MPI Interoperability Through Flexible Communication Endpoints". In: *Proceedings of the 20th European MPI Users' Group Meeting*. EuroMPI '13. Madrid, Spain, 2013, pp. 13–18. ISBN: 978-1-4503-1903-4.

[60] Lamia Djoudi et al. “Exploring application performance: a new tool for a static/dynamic approach”. In: *Proceedings of the 6th LACSI Symposium*. 2005. URL: `http://www.researchgate.net/profile/William_Jalby/publication/239735488_Exploring_Application_Performance_a_New_Tool_For_a_StaticDynamic_Approach/links/00b4952d6f80d5e051000000.pdf` (visited on 09/14/2015).

[61] Jack Dongarra and Michael A Heroux. “Toward a new metric for ranking high performance computing systems”. In: *Sandia Report, SAND2013-4744* 312 (2013).

[62] Isaac Dooley. “Intelligent Runtime Tuning of Parallel Applications With Control Points”. http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml. PhD thesis. Dept. of Computer Science, University of Illinois, 2010.

[63] Gábor Dózsa et al. “Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems”. In: *European MPI Users’ Group Meeting Conference on Recent Advances in the Message Passing Interface*. EuroMPI’10. 2010.

[64] Peng Du et al. “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming”. In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[65] Alejandro Duran et al. “Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp”. In: *Parallel Processing, 2009. ICPP’09. International Conference on*. IEEE. 2009, pp. 124–131.

[66] Juan Durillo and Thomas Fahringer. “From single-to multi-objective auto-tuning of programs: Advantages and implications”. In: *Scientific Programming* 22.4 (2014), pp. 285–297.

[67] Alexandre E. Eichenberger et al. “OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis”. In: *IWOMP 2013*. Vol. 8122. 2013, pp. 171–185. ISBN: 978-3-642-40697-3.

[68] Alexandre Eichenberger et al. “OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging”. (OpenMP 4.0 draft proposal). 2014.

[69] Alexandre Eichenberger et al. *OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging*. Tech. rep. Technical report, 2013.

[70] Dominic Eschweiler et al. “Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries.” In: *PARCO*. Vol. 22. 2011, pp. 481–490.

[71] S. Feki and E. Gabriel. “Incorporating Historic Knowledge into a Communication Library for Self-Optimizing High Performance Computing Applications”. In: *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO ’08*. Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO ’08. Oct. 2008, pp. 265–274. DOI: `10.1109/SASO.2008.47`.

[72] Leonardo Fialho and James Browne. “Framework and modular infrastructure for automation of architectural adaptation and performance optimization for

HPC systems". In: (2014), pp. 261–277. URL: `http://link.springer.com/chapter/10.1007/978-3-319-07518-1_17` (visited on 07/10/2015).

[73] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Sept. 2012. URL: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[74] Matteo Frigo, Steven, and G. Johnson. "The design and implementation of FFTW3". In: *Proceedings of the IEEE*. 2005, pp. 216–231.

[75] Karl Fürlinger and Michael Gerndt. "ompP: A profiling tool for OpenMP". In: *OpenMP Shared Memory Parallel Programming*. Springer, 2008, pp. 15–23.

[76] Grigori Fursin. "Collective Mind: cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning". In: *arXiv:1308.2410 [cs, stat]* (Aug. 11, 2013). URL: `http://arxiv.org/abs/1308.2410` (visited on 05/01/2014).

[77] Grigori Fursin. "Collective Tuning Initiative: automating and accelerating development and optimization of computing systems". In: GCC Developers' Summit. June 8, 2009. URL: `http://hal.inria.fr/inria-00436029` (visited on 05/01/2014).

[78] E. Gabriel and S. Huang. "Runtime Optimization of Application Level Communication Patterns". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. Mar. 2007, pp. 1–8. DOI: `10.1109/IPDPS.2007.370406`.

[79] Edgar Gabriel et al. "Towards Performance Portability Through Runtime Adaptation for High-performance Computing Applications". In: *Concurr.*

Comput. : Pract. Exper. 22.16 (Nov. 2010), pp. 2230–2246. ISSN: 1532-0626. DOI: `10.1002/cpe.v22:16`. URL: `http://dx.doi.org/10.1002/cpe.v22:16` (visited on 05/01/2014).

[80] Markus Geimer et al. "A scalable tool architecture for diagnosing wait states in massively parallel applications". In: *Parallel Computing* 35.7 (2009), pp. 375–388. URL: `http://www.sciencedirect.com/science/article/pii/S0167819109000398` (visited on 08/10/2015).

[81] Markus Geimer et al. "Scalable parallel trace-based performance analysis". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2006, pp. 303–312. URL: `http://link.springer.com/chapter/10.1007/11846802_43` (visited on 08/10/2015).

[82] Michael Gerndt. "Performance analysis tools". In: *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1515–1522.

[83] Joseph E. Gonzalez et al. "Graphx: Graph processing in a distributed dataflow framework". In: *Proceedings of OSDI*. 2014, pp. 599–613. URL: `https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-gonzalez.pdf` (visited on 11/11/2015).

[84] David Goodell et al. "Minimizing MPI Resource Contention in Multithreaded Multicore Environments". In: *IEEE International Conference on Cluster Computing*. CLUSTER '10. 2010.

[85] Georgios Goumas et al. "Adapt or Become Extinct!: The Case for a Unified Framework for Deployment-time Optimization (Position Paper)". In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT '11. New York, NY, USA: ACM,

2011, pp. 46–51. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000422. URL: http://doi.acm.org/10.1145/2000417.2000422 (visited on 05/01/2014).

[86] Martin Griebl, Christian Lengauer, and Sabine Wetzel. "Code Generation in the Polytope Model". In: *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[87] OpenACC Working Group. *The OpenACC Application Programming Interface, Version 2.0*. 2013. URL: http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf.

[88] Patricia Grubel et al. "The Performance Implication of Task Size for Applications on the HPX Runtime System". In: *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE. 2015, pp. 682–689.

[89] Patricia Grubel et al. "Using Intrinsic Performance Counters to Assess Efficiency in Task-Based Parallel Applications". In: *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE. 2016, pp. 1692–1701.

[90] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. "Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage". In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Lecture Notes in Computer Science 8632. Springer International Publishing, Aug. 25, 2014, pp. 87–98. ISBN: 978-3-319-09872-2, 978-3-319-09873-9. URL: http:

//link.springer.com/chapter/10.1007/978-3-319-09873-9_8 (visited on 04/13/2015).

[91] Apache Hadoop. *Pluggable Shuffle and Pluggable Sort*. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html.

[92] J. Mario Gallegos, Zhiqi Tao, and Quy Ta- Dell. *Deploying Hadoop on Lustre Storage: Lessons learned and best practices.* Lustre User Group Meeting. 2015.

[93] Robert J Hall. "Call path profiling". In: *Proceedings of the 14th international conference on Software engineering*. ACM. 1992, pp. 296–306.

[94] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. "Annotation-Based Empirical Performance Tuning Using Orio". In: *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*. Rome, Italy, 2009.

[95] *Habanero-C*. wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[96] Michael Heroux and Richard Barrett. *Mantevo Project*. 2011. URL: https://mantevo.org.

[97] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[98] Jeffrey Hollingsworth and Ananta Tiwari. "End-to-End Auto-Tuning with Active Harmony". In: *Performance Tuning of Scientific Applications*. CRC Press,

June 2010. Chap. 10, pp. 217–238. ISBN: 978-1-4398-1569-4. DOI: `doi:10.1201/b10509-11`. URL: `http://dx.doi.org/10.1201/b10509-11`.

[99] Brandon Holt et al. “Flat Combining Synchronized Global Data Structures”. In: *7th International Conference on PGAS Programming Models*, p. 76.

[100] Chao Huang, Orion Lawlor, and Laxmikant V Kale. “Adaptive MPI”. In: *Languages and Compilers for Parallel Computing*. Springer, 2004, pp. 306–322.

[101] Chao Huang, Gengbin Zheng, and Laxmikant V Kalé. “Supporting adaptivity in MPI for dynamic parallel applications”. In: *Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign* (2007).

[102] Kevin A Huck et al. “An Autonomic Performance Environment for Exascale”. In: *Supercomputing frontiers and innovations* 2.3 (2015), pp. 49–66.

[103] Kevin A Huck et al. “Integrated Measurement for Cross-Platform OpenMP Performance Analysis”. In: *IWOMP 2014: Using and Improving OpenMP for Devices, Tasks, and More*. Springer International Publishing, 2014, pp. 146–160.

[104] Kevin A. Huck et al. “TAUg: Runtime Global Performance Data Access Using MPI”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Bernd Mohr et al. Lecture Notes in Computer Science 4192. Springer Berlin Heidelberg, 2006, pp. 313–321. ISBN: 978-3-540-39110-4, 978-3-540-39112-8. URL: `http://link.springer.com/chapter/10.1007/11846802_44` (visited on 05/20/2015).

[105] Kevin A. Huck et al. “TAUg: Runtime Global Performance Data Access Using MPI”. English. In: *Recent Advances in Parallel Virtual Machine and*

Message Passing Interface. Vol. 4192. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 313–321. ISBN: 978-3-540-39110-4. DOI: 10.1007/11846802_44. URL: `http://dx.doi.org/10.1007/11846802_44`.

[106] Kevin Huck et al. "An Autonomic Performance Environment for Exascale". In: *Supercomputing Frontiers and Innovations* 2.3 (2015). ISSN: 2313-8734. URL: `http://superfri.org/superfri/article/view/64`.

[107] Kevin Huck et al. "An Early Prototype of an Autonomic Performance Environment for Exascale". In: *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '13. Eugene, Oregon: ACM, 2013, 8:1–8:8. ISBN: 978-1-4503-2146-4. DOI: 10.1145/2491661.2481434. URL: `http://doi.acm.org/10.1145/2491661.2481434`.

[108] Khaled Z. Ibrahim and Katherine A. Yelick. "On the conditions for efficient interoperability with threads: an experience with PGAS languages using Cray communication domains". In: *ICS*. 2014.

[109] K.Z. Ibrahim et al. "An Evaluation of One-Sided and Two-Sided Communication Paradigms on Relaxed-Ordering Interconnect". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. May 2014, pp. 1115–1125.

[110] Intel. *Intel SDK for OpenCL Applications - Performance Debugging*. 2013. URL: `https://software.intel.com/en-us/articles/intel-sdk-for-opencl-applications-performance-debugging-intro`.

[111] N. S. Islam et al. "High Performance RDMA-based Design of HDFS over InfiniBand". In: *Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 35:1–35:35. ISBN: 978-1-4673-0804-5. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389044`.

[112] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[113] ISO. *Standard for Programming Language C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.

[114] Douglas M Jacobsen and Richard Shane Canon. "Contain This, Unleashing Docker for HPC". In: *Proceedings of the Cray User Group* (2015).

[115] Changhee Jung et al. "Brainy: Effective Selection of Data Structures". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: ACM, 2011, pp. 86–97. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993509. URL: `http://doi.acm.org/10.1145/1993498.1993509` (visited on 05/02/2014).

[116] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. "Parallex an advanced parallel execution model for scaling-impaired applications". In: *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE. 2009, pp. 394–401.

[117] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications". In: *Proceedings of the 2009 International Conference on Parallel Processing Workshops*. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–

401. ISBN: 978-0-7695-3803-7. DOI: 10.1109/ICPPW.2009.14. URL: http://dx.doi.org/10.1109/ICPPW.2009.14.

[118] Hartmut Kaiser et al. “Higher-level parallelization for local and distributed asynchronous task-based programming”. In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM. 2015, pp. 29–37.

[119] Hartmut Kaiser et al. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. Eugene, OR, USA: ACM, 2014, 6:1–6:11. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883. URL: http://doi.acm.org/10.1145/2676870.2676883.

[120] Hartmut Kaiser et al. “HPX: A task based programming model in a global address space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 6.

[121] Laxmikant V. Kale and Gengbin Zheng. “Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects”. In: *Advanced Computational Infrastructures for Parallel and Distributed Applications*. Ed. by M. Parashar. Wiley-Interscience, 2009, pp. 265–282.

[122] Karen L Karavanic and Barton P Miller. “Improving online performance diagnosis by the use of historical performance data”. In: *Supercomputing, ACM/IEEE 1999 Conference*. IEEE. 1999, pp. 42–42.

[123] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, Aug. 2013, pp. 1–9.

[124] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: 10.1137/S1064827595287997. URL: `http://dx.doi.org/10.1137/S1064827595287997`.

[125] Andreas Knüpfer et al. "The vampir performance analysis tool-set". In: *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[126] Souad Koliai et al. "A Balanced Approach to Application Performance Tuning". In: *Languages and Compilers for Parallel Computing*. Ed. by Guang R. Gao et al. Lecture Notes in Computer Science 5898. Springer Berlin Heidelberg, 2010, pp. 111–125. ISBN: 978-3-642-13373-2, 978-3-642-13374-9. URL: `http://link.springer.com/chapter/10.1007/978-3-642-13374-9_8` (visited on 05/01/2014).

[127] Souad Koliaï et al. "Quantifying Performance Bottleneck Cost Through Differential Analysis". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. New York, NY, USA: ACM, 2013, pp. 263–272. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465440. URL: `http://doi.acm.org/10.1145/2464996.2465440` (visited on 05/01/2014).

[128] Sameer Kumar and Laxmikant V. Kale. "Scaling All-to-All Multicast on Fat-tree Networks". In: *ICPADS'04*. 2004, p. 205. ISBN: 0-7695-2152-5.

[129] Vivek Kumar et al. "HabaneroUPC++: A Compiler-free PGAS Library". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. New York, NY, USA: ACM, 2014,

5:1–5:10. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676879. URL: `http://doi.acm.org/10.1145/2676870.2676879` (visited on 05/20/2015).

[130] GM Kurtzer, V Sochat, and MW Bauer. "Singularity: Scientific containers for mobility of compute." In: *PloS one* 12.5 (2017), e0177459.

[131] Mahendra Kutare et al. "Monalytics: online monitoring and analytics for managing large scale data centers". In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 141–150.

[132] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 81–91. ISBN: 978-0-7695-3576-0. DOI: 10.1109/CGO.2009.21. URL: `http://dx.doi.org/10.1109/CGO.2009.21` (visited on 05/01/2014).

[133] Dong Li et al. "Hybrid MPI/OpenMP power-aware computing". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–12.

[134] Dong Li et al. "Strategies for energy-efficient resource management of hybrid programming models". In: *Parallel and Distributed Systems, IEEE Transactions on* 24.1 (2013), pp. 144–157.

[135] Haoyuan Li et al. "Tachyon: Reliable, memory speed storage for cluster computing frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15. URL: `http://dl.acm.org/citation.cfm?id=2670985` (visited on 11/11/2015).

[136] Chunhua Liao et al. “Early experiences with the openmp accelerator model”. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[137] Xiaoyi Lu et al. “Accelerating Spark with RDMA for Big Data Processing: Early Experiences”. In: *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects (HOTI)*. 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects (HOTI). Aug. 2014, pp. 9–16. DOI: 10.1109/HOTI.2014.15.

[138] Miao Luo et al. “Congestion Avoidance on Manycore High Performance Computing Systems”. In: *ICS*. 2012.

[139] Miao Luo et al. “Initial Study of Multi-endpoint Runtime for MPI+OpenMP Hybrid Programming Model on Multi-core Systems”. In: *SIGPLAN Not.* 49.8 (Feb. 2014).

[140] Huong Luu et al. “A Multiplatform Study of I/O Behavior on Petascale Supercomputers”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’15. 2015.

[141] Kamesh Madduri et al. “Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms”. In: *Parallel Comput.* 37.9 (2011), pp. 501–520. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.02.001. URL: dx.doi.org/10.1016/j.parco.2011.02.001.

[142] Kamesh Madduri et al. “Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. 2011, 23:1–23:12.

[143] Allen D Malony, Sameer Shende, and Alan Morris. “Phase-Based Parallel Performance Profiling.” In: *PARCO*. 2005, pp. 203–210.

[144] Allen D Malony et al. “Parallel performance measurement of heterogeneous parallel systems with GPUs”. In: *Parallel Processing (ICPP), 2011 International Conference on*. IEEE. 2011, pp. 176–185.

[145] Azamat Mametjanov et al. “Autotuning Stencil-Based Computations on GPUs”. In: *Proceedings of IEEE Cluster 2012*. 2012.

[146] Anirban Mandal, Rob Fowler, and Allan Porterfield. “Modeling Memory Concurrency for Multi-Socket Multi-Core Systems”. In: *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS2010)*. IEEE. White Plains, NY, Mar. 2010, pp. 56–75.

[147] Anirban Mandal, Rob Fowler, and Allan Porterfield. “System-wide introspection for accurate attribution of performance bottlenecks”. In: *Second International Workshop on High-perfromance Infrastruture for Scalable Tools*. 2012.

[148] Anirban Mandel, Rob Fowler, and Allan Porterfield. “System-wide introspection for accurate attribution of performance bottlenecks”. In: *Second International Workshop on High-perfromance Infrastruture for Scalable Tools*. 2012.

[149] Aniruddha Marathe et al. “A Run-Time System for Power-Constrained HPC Applications”. In: *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*. Vol. 9137. Springer. 2015, p. 394.

[150] Steven J. Martin and Matthew Kappel. “Cray XC30 Power Monitoring and Management”. In: *Cray User Group Conference Proceedings*. 2014.

[151] Kristyn J Maschhoff and Michael F Ringenburg. “Experiences running and optimizing the berkeley data analytics stack on cray platforms”. In: *Cray Users Group* (2015).

[152] T Mattson et al. *OCR: The Open Community Runtime interface version 1.1. 0*. 2015.

[153] *MPI Solutions for GPUs*. `https://developer.nvidia.com/mpi-solutions-gpus`.

[154] Paul E McKenney. “Is parallel programming hard, and, if so, what can you do about it?” In: *Linux Technology Center, IBM Beaverton* (2011).

[155] Abdul Wahid Memon and Grigori Fursin. “Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing”. In: PARCO mini-symposium on ’Application Autotuning for HPC (Architectures)’. Sept. 12, 2013. URL: `http://hal.inria.fr/hal-00944513` (visited on 05/01/2014).

[156] Jiayuan Meng et al. “GROPHECY: GPU Performance Projection from CPU Code Skeletons”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. New York, NY, USA: ACM, 2011, 14:1–14:11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063402. URL: `http://doi.acm.org/10.1145/2063384.2063402` (visited on 05/30/2014).

[157] Xiangrui Meng et al. “MLlib: Machine Learning in Apache Spark”. In: *arXiv:1505.06807 [cs, stat]* (May 26, 2015). arXiv: 1505.06807. URL: `http://arxiv.org/abs/1505.06807` (visited on 11/11/2015).

[158] Harshitha Menon and Laxmikant Kalé. “A distributed dynamic load balancer for iterative applications”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE. 2013, pp. 1–11.

[159] Renato Miceli et al. “AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications”. In: *Applied Parallel and Scientific Computing*. Ed. by Pekka Manninen and Per Öster. Lecture Notes in Computer Science 7782. Springer Berlin Heidelberg, 2013, pp. 328–342. ISBN: 978-3-642-36802-8, 978-3-642-36803-5. URL: `http://link.springer.com/chapter/10.1007/978-3-642-36803-5_24` (visited on 05/01/2014).

[160] Barton P Miller et al. “The Paradyn parallel performance measurement tool”. In: *Computer* 28.11 (1995), pp. 37–46.

[161] *miniGMG website*. `http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG`.

[162] Bernd Mohr. “Scalable parallel performance measurement and analysis tools-state-of-the-art and future challenges”. In: *Supercomputing frontiers and innovations* 1.2 (2014), pp. 108–123.

[163] *National Energy Research Scientific Computing Center*. `https://www.nersc.gov`.

[164] Jacob Nelson et al. “Latency-tolerant Software Distributed Shared Memory”. In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’15. Santa Clara, CA: USENIX Association, 2015,

pp. 291–305. ISBN: 978-1-931971-225. URL: `http://dl.acm.org/citation.cfm?id=2813767.2813789`.

[165] *NERSC User Survey*. 2013. URL: `https://www.nersc.gov/news-publications/publications-reports/user-surveys/2013/`.

[166] Chris J Newburn et al. "Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1213–1225.

[167] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[168] Jarek Nieplocha and Bryan Carpenter. "ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems". In: *IPPS/SPDP'99 Workshops*. 1999.

[169] NVIDIA. *CUDA Toolkit 7.5 CUPTI API Specification*. 2015. URL: `http://docs.nvidia.com/cuda/cupti/`.

[170] *Oak Ridge Leadership Computing Facility*. https://www.olcf.ornl.gov.

[171] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. Nov. 2015. URL: `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[172] Kay Ousterhout et al. "Making sense of performance in data analytics frameworks". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*. 2015, pp. 293–307.

URL: `https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-ousterhout.pdf` (visited on 11/11/2015).

[173] Tapasya Patki et al. "Exploring hardware overprovisioning in power-constrained, high performance computing". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 173–182.

[174] S. Pellegrini, R. Prodan, and T. Fahringer. "Tuning MPI Runtime Parameter Setting for High Performance Computing". In: *2012 IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS)*. 2012 IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS). Sept. 2012, pp. 213–221. DOI: 10.1109/ClusterW.2012.15.

[175] Chuck Pheatt. "Intel&Reg; Threading Building Blocks". In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), pp. 298–298. ISSN: 1937-4771. URL: `http://dl.acm.org/citation.cfm?id=1352079.1352134`.

[176] Meikel Poess et al. "Tpc-ds, taking decision support benchmarking to the next level". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 582–587. URL: `http://dl.acm.org/citation.cfm?id=564759` (visited on 11/11/2015).

[177] Markus Püschel et al. "SPIRAL: Code generation for DSP transforms". In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[178] Dan Quinlan. "ROSE: Compiler support for object-oriented frameworks". In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226.

[179] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on.* Feb. 2009, pp. 427–436.

[180] Giridhar Ravipati et al. *Toward the deconstruction of Dyninst.* Tech. rep. Technical Report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp. cs. wisc. edu/paradyn/papers/Ravipati07Symta bAPI. pdf), 2007.

[181] James Reinders. "An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors". In: (2012). URL: `https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf`.

[182] Arch D. Robison. "Composable Parallel Patterns with Intel Cilk Plus". In: *Computing in Science and Engineering* 15.2 (2013), pp. 66–71, 87. DOI: `http://dx.doi.org/10.1109/MCSE.2013.21`. URL: `http://scitation.aip.org/content/aip/journal/cise/15/2/10.1109/MCSE.2013.21`.

[183] Roxana-Ioana Roman et al. "Understanding Spark Performance in Hybrid and Multi-Site Clouds". In: *6th International Workshop on Big Data Analytics: Challenges and Opportunities (BDAC-15).* 2015.

[184] Philip C Roth and Barton P Miller. "On-line automated performance diagnosis on thousands of processes". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM. 2006, pp. 69–80.

[185] Barry Rountree et al. “Beyond DVFS: A first look at performance under a hardware-enforced power bound”. In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE. 2012, pp. 947–953.

[186] Gabe Rudy. “CUDA-CHiLL: A Programming Language Interface for GPGPU Optimization and Code Generation”. MA thesis. University of Utah, 2010.

[187] Santosh Sarangkar and Apan Qasem. “MATS: A Model-driven Adaptive Tuning System for Parallel Workloads”. In: *Journal of Parallel and Cloud Computing (JPCC)* 1.2 (2012), pp. 50–64.

[188] Barry Rountreee Scott Walker Kathleen Shoga and Lauren Morita. *libmsr - A Wrapper library for model-specific registers*. `https://github.com/LLNL/libmsr`. 2014.

[189] Rathijit Sen and David A Wood. *Cache Power Budgeting for Performance*. Tech. rep. 2013.

[190] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009.

[191] Md Abdullah Shahneous Bari et al. “ARCS: Adaptive Runtime Configuration Selection for Power-Constrained OpenMP Applications”. In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2016.

[192] Sameer S Shende and Allen D Malony. “The TAU parallel performance system”. In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[193] Jaewook Shin et al. “Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology”. In: *Software Automatic Tuning*. Springer, 2010, pp. 353–370.

[194] Jaewook Shin et al. “Speeding up Nek5000 with autotuning and specialization”. In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM. 2010, pp. 253–262.

[195] Min Si et al. “MT-MPI: Multithreaded MPI for Many-core Environments”. In: *ICS*. Munich, Germany, 2014, pp. 125–134. ISBN: 978-1-4503-2642-1.

[196] Seoul National University Center for Manycore Programming. *Implementation of NAS Parallel Benchmark Using C*. `http://aces.snu.ac.kr/SNU_NPB_Suite.html`. 2013.

[197] *spark-perf Benchmark*. `https://github.com/databricks/spark-perf`.

[198] Jonathan Sparks, Howard Pritchard, and Martha Dumler. “The Cray Framework for Hadoop for the Cray XC30”. In: (). URL: `https://cug.org/proceedings/cug2014_proceedings/includes/files/pap160.pdf` (visited on 11/11/2015).

[199] Thomas Sterling et al. “SLOWER: A performance model for Exascale computing”. In: *Supercomputing frontiers and innovations* 1.2 (2014), pp. 42–57.

[200] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: `10.1109/MCSE.2010.69`.

[201] Ching-Long Su and Alvin M Despain. “Cache designs for energy efficiency”. In: *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*. Vol. 1. IEEE. 1995, pp. 306–315.

[202] M Aater Suleman, Moinuddin K Qureshi, and Yale N Patt. “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs”. In: *ACM Sigplan Notices* 43.3 (2008), pp. 277–286.

[203] Yanhua Sun, Jonathan Lifflander, and L. V. Kale. “PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications”. In: *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*. Munich, Germany, June 2014.

[204] Lingjia Tang, Jason Mars, and Mary Lou Soffa. “Contentiousness vs. Sensitivity: Improving Contention Aware Runtime Systems on Multicore Architectures”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT ’11. New York, NY, USA: ACM, 2011, pp. 12–21. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000419. URL: `http://doi.acm.org/10.1145/2000417.2000419` (visited on 05/27/2014).

[205] Wittawat Tantisiriroj et al. “On the duality of data-intensive file system design: reconciling HDFS and PVFS”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 67.

[206] Cristian Țăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. "Active Harmony: Towards Automated Performance Tuning". In: *2002 ACM/IEEE Conference on Supercomputing*. SC '02. Baltimore, Maryland: IEEE Computer Society Press, 2002, pp. 1–11. URL: `http://dl.acm.org/citation.cfm?id=762761.762771`.

[207] Valerie Taylor, Xingfu Wu, and Rick Stevens. "Design and implementation of prophesy automatic instrumentation and data entry system". In: *Proceedings of the Parallel and Distributed Computing and Systems Conference (PDCS)*. 2001.

[208] V. Taylor et al. "Prophesy: automating the modeling process". In: *Third Annual International Workshop on Active Middleware Services, 2001*. Third Annual International Workshop on Active Middleware Services, 2001. Aug. 2001, pp. 3–11. DOI: 10.1109/AMS.2001.993715.

[209] Traileka Glacier Team. *What Is OCR?* June 2014. URL: `https://xstack.exascale-tech.com/wiki/images/d/d3/What-is-OCR.pptx`.

[210] Rajeev Thakur and William Gropp. "Test Suite for Evaluating Performance of Multithreaded MPI Communication". In: *Parallel Comput.* 35.12 (2009).

[211] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of Collective Communication Operations in MPICH". In: *IJHPCA* (2005), pp. 49–66.

[212] The National Energy Research Scientific Computing Center (NERSC). "Edison". `https://www.nersc.gov/users/computational-systems/edison/`. Apr. 2015.

[213] Peter Thoman, Philipp Gschwandtner, and Thomas Fahringer. "On the quality of implementation of the c++ 11 thread support library". In: *Parallel,*

Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on. IEEE. 2015, pp. 94–98.

[214] Ananta Tiwari et al. “Auto-tuning full applications: A case study”. In: *Int. J. High Perform. Comput. Appl.* 25.3 (Aug. 2011), pp. 286–294. ISSN: 1094-3420. DOI: 10.1177/1094342011414744. URL: `http://dx.doi.org/10.1177/1094342011414744`.

[215] Antana Tiwari. “Tuning Parallel Applications in Parallel”. PhD thesis. University of Maryland, College Park, 2011.

[216] *TOP500 List*. Nov. 2015. URL: `http://www.top500.org/lists/2015/11/`.

[217] UC Berkeley AmpLab. *Big Data Benchmark*. URL: `https://amplab.cs.berkeley.edu/benchmark/`.

[218] UPC Consortium. `upc.lbl.gov/docs/user/upc_spec_1.2.pdf`.

[219] UPC Consortium. *UPC Language and Library Specifications, v1.3*. Tech Report LBNL-59208. Lawrence Berkeley National Lab, 2013. URL: `http://upc.lbl.gov/publications/upc-spec-1.3.pdf`.

[220] UPC Consortium. *UPC Optional Library Specifications- version 1.3*. upc-specification.googlecode.com/files/upc-lib-optional-spec-1.3-draft-3.pdf. Nov. 2012.

[221] *UPC-FT benchmark*. `https://www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/npb-upc-ft/`.

[222] Jeffrey Vetter and Chris Chambreau. “MPIp: Lightweight, scalable MPI profiling”. In: (2005).

[223] Richard Vuduc, James W Demmel, and Katherine A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[224] Michael Wagner, Tobias Hilbrich, and Holger Brunst. “Online Performance Analysis: An Event-Based Workflow Design towards Exascale”. In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*. IEEE. 2014, pp. 839–846.

[225] Wei Wang et al. “Using Per-Loop CPU Clock Modulation for Energy Efficiency in OpenMP Applications”. In: *Energy* 1.1.4 (2015), pp. 1–6.

[226] Yandong Wang et al. “Hadoop Acceleration Through Network Levitated Merge”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. New York, NY, USA: ACM, 2011, 57:1–57:10. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063461. URL: `http://doi.acm.org/10.1145/2063384.2063461` (visited on 11/11/2015).

[227] Alexander Wert, Jens Happe, and Lucia Happe. “Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 552–561. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486861` (visited on 05/13/2014).

[228] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27 (2000), p. 2001.

[229] R. Clint Whaley and Jack J. Dongarra. “Automatically tuned linear algebra software”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing ’98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X. URL: `http://dl.acm.org/citation.cfm?id=509058.509096`.

[230] K.B. Wheeler, R.C. Murphy, and D. Thain. “Qthreads: An API for programming with millions of lightweight threads”. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Apr. 2008, pp. 1–8. DOI: `10.1109/IPDPS.2008.4536359`.

[231] Tom White. *Hadoop: The Definitive Guide*. 1st. O’Reilly Media, Inc., 2009. ISBN: 0596521979, 9780596521974.

[232] M. Wilde et al. “Swift: A Language for Distributed Parallel Scripting”. In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[233] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[234] Samuel Williams et al. “Extracting Ultra-scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-tuning”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. ACM, 2011.

[235] Samuel Williams et al. *Implementation and Optimization of miniGMG - a Compact Geometric Multigrid Benchmark*. Tech. rep. LBNL 6676E. Lawrence Berkeley National Laboratory, Dec. 2012.

[236] Samuel Williams et al. "Optimization of geometric multigrid for emerging multi- and manycore processors". In: *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. IEEE Computer Society Press, 2012.

[237] Thomas Williams and Colin Kelley. "Gnuplot Homepage". `http://www.gnuplot.info`. Apr. 2015.

[238] Felix Wolf et al. "Automatic Analysis of Inefficiency Patterns in Parallel Applications: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 19.11 (Aug. 2007), pp. 1481–1496. ISSN: 1532-0626. DOI: 10.1002/cpe.v19:11. URL: `http://dx.doi.org/10.1002/cpe.v19:11` (visited on 05/01/2014).

[239] J. M. Wozniak et al. "Turbine: A Distributed-Memory Dataflow Engine for Extreme-Scale Many-Task Applications". In: *Proceedings SWEET 2012*. Scottsdale, AZ, May 2012. URL: `https://sites.google.com/site/sweetworkshop2012/`.

[240] J.M. Wozniak et al. "Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). May 2013, pp. 95–102. DOI: 10.1109/CCGrid.2013.99.

[241] Xingfu Wu et al. "Design and development of Prophesy Performance Database for distributed scientific applications". In: *10th SIAM Conference on Parallel Processing for Scientific Computing*. 2001.

[242] Yuanyuan Yang and Jianchao Wang. "Efficient All-to-All Broadcast in All-Port Mesh and Torus Networks". In: *International Symposium on High Performance Computer Architecture*. HPCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 290–. ISBN: 0-7695-0004-8. URL: `http://dl.acm.org/citation.cfm?id=520549.822773`.

[243] Yuanyuan Yang and Jianchao Wang. "Near-Optimal All-to-All Broadcast in Multidimensional All-Port Meshes and Tori". In: *IEEE Trans. Parallel Distrib. Syst.* 13 (2 Feb. 2002), pp. 128–141. ISSN: 1045-9219. DOI: 10.1109/71.983941. URL: `http://dl.acm.org/citation.cfm?id=506358.506361`.

[244] Asim YarKhan. "Dynamic task execution on shared and distributed memory architectures". PhD thesis. Knoxville, TN, USA: University of Tennessee, 2012. URL: `http://trace.tennessee.edu/utk_graddiss/1575/`.

[245] Katherine Yelick et al. "Productivity and performance using partitioned global address space languages". In: *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM. 2007, pp. 24–32.

[246] Kamen Yotov et al. "A Comparison of Empirical and Model-driven Optimization". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. New York, NY, USA: ACM, 2003, pp. 63–76. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781140. URL: `http://doi.acm.org/10.1145/781131.781140` (visited on 05/01/2014).

[247] Matei Zaharia et al. "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*. USENIX Association, 2012, pp. 2–2. URL: `http://dl.acm.org/citation.cfm?id=2228301` (visited on 06/17/2015).

[248] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10. URL: `http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf` (visited on 06/17/2015).

[249] Gengbin Zheng. "Achieving high performance on extremely large parallel machines: performance prediction and load balancing". PhD thesis. Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[250] Yili Zheng et al. "UPC++: A PGAS Extension for C++". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. May 2014, pp. 1105–1114. DOI: `10.1109/IPDPS.2014.115`.