TECHNIQUES IN SCALABLE AND EFFECTIVE PARALLEL PERFORMANCE ANALYSIS

BY

CHEE WAI LEE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant Kale, Chair
Professor Marc Snir
Professor Michael Heath
Doctor Luiz DeRose, Cray Inc.

Abstract

Performance analysis tools are essential to the maintenance of efficient parallel execution of scientific applications. As scientific applications are executed on larger and larger parallel supercomputers, it is clear that performance tools must employ more advanced techniques to keep up with the increasing data volume and complexity of the performance information generated by these applications as a result of scaling.

In this thesis, we investigate the useful techniques in four main thrusts to address various aspects of this problem. First, we study how some traditional performance analysis idioms can break down in the face of data from large processor counts and demonstrate techniques and tools that restore scalability. Second, we investigate how the volume of performance data generated can be reduced while keeping the captured information relevant for analysis and performance problem detection. Third, we investigate the powerful new performance analysis idioms enabled by live access to performance information streams from a running parallel application. Fourth, we demonstrate how repeated performance hypothesis testing can be conducted, via simulation techniques, scalably and with significantly reduced resource consumption. In addition, we explore the benefits of performance tool integration to the propagation and synergy of scalable performance analysis techniques in different tools.

To my family and towards an age of reason.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Kalé for his great patience and guidance throughout the long and oftentimes difficult process of completing the thesis. His talent at managing so many students with so many diverse interests in our field of High Performance Computing is a wonder to behold and a fantastic source of inspiration.

I would also like to thank the rest of my thesis committee: Professor Marc Snir, Professor Michael Heath and Dr. Luiz DeRose, for their kind guidance and constructive feedback.

My very special thanks to Celso Mendes, Isaac Dooley, Eric Bohm, Abhinav Bhatele, Gengbin Zheng and JoAnne Geigner for the immense amount of help and support toward helping me complete the thesis. Also, a big thank-you to the many many hang-out buddies at the Parallel Programming Laboratory over the years for making the lab simply a riot of fun: Eric Bohm, Ramprasad, Esteban Meneses, Filippo, Abhinav, Pritish, Jessie, Rahul Jain, Lukasz, Kumaresh, Phil, Dave, Isaac, Aaron, Esteban Pauli, Chao Mei, Eric Shook, Viraj, Ekaterina, Terry, Sayantan, Rahul Joshi, Chao Huang, Yan, Greg, Orion, Sameer, Hari, Amit, Nilesh, Tarun, Yogesh, Vikas, Sharon, Apurva, Ramkumar, Rashmi, Sindhura, Guna, Mani, Theckla, Jonathan, Josh, Arun, Neelam and Puneet. Special shout-outs go to (you know who you are): my chess buddies, my drinking buddies, my gym buddies and my punching bags (sorry!).

Many thanks for the close friendships with so many people I have met at the University of Illinois at Urbana-Champaign. Your support and company have always been appreciated, even when it contributed to the dreaded “grad student procastination”. To the “net party” gang, we sure had many crazy times together: Fredrik Vraalsen, Jim Oly, Chris and Tanya Lattner, Mike

Hunter, Rushabh Doshi, Bradley Jones, Bill Wendling, Howard and Melissa Sun, Nicholas Riley, Andy Reitz, Vinh Lam, Adam Slagell and Joel Stanley. To the "CS 105" gang, thanks for all the fun times working together to teach non-CS majors the joys of Computer Science: Dominique Kilman, Yang Yaling, Marsha and Roger Woodbury, Francis David, Tony Hursh, Reza Lesmana and Shamsi Iqbal. To the other "crazy" grad students of the Computer Science Department, thanks for all the fish!: Lee Baugh and Dmitry Yershov.

To the various faculty and staff of the Computer Science Department who have helped me with so many administrative burdens, special thanks go out to Professor Geneva Belford, Barb Cicone, Mary Beth Kelley, Sheila Clark, Molly Flesner, Shirley Finke and Elaine Wilson.

To Hamid at the Jerusalem restaurant along Wright Street, thank you for the fine food and company. I have enjoyed every moment working on this thesis at your establishment and partaking in the refreshing Turkish coffee you brew.

Last but most certainly not least, to my parents and my grandfather, thank you all for bearing my long absence with such grace, love and unwavering support. To my now-deceased grandmother, thank you for holding out as long as you did in the hope of seeing me graduate. To my sister, thanks for keeping me company over the last few years on Facebook. To my fiancée, thank you for your continued patience and love, and for sticking it out with me long-distance over the thirteen or so years we have been together.

Grants and other Acknowledgements:

NAMD was developed at the Theoretical Biophysics Group (Beckman Institute, University of Illinois) and funded by the National Institutes of Health(NIH PHS 5 P41-RR05969-04). Projections and Charm++ are supported by several projects funded by the Department of Energy (subcontract B523819) and the National Science Foundation (NSF DMR 0121695)

This work is supported by grant(s) from the National Institutes of Health P41-RR05969. The author gladly acknowledge supercomputer time provided by Pittsburgh Supercomputer Center and the National Center for Supercomputing Applications via Large Resources Allocation Committee grant MCA93S028.

Table of Contents

List of Tables . . . . . **ix**

List of Figures . . . . . **x**

Chapter 1 Introduction And Motivation **1**
1.1 Parallel Performance Tuning Tools 2
1.2 Challenges To Performance Tool Effectiveness Due to Application Scaling . . . 4
1.3 Background 6
1.4 Thesis Objectives And Scope 8

Chapter 2 Software Infrastructure **10**
2.1 Charm++ 10
2.1.1 Adaptive MPI 13
2.1.2 Adaptive Overlap 14
2.1.3 Automatic Load Balancing 16
2.2 The Projections Performance Analysis And Visualization Framework 17
2.2.1 Charm++ Performance Events 18
2.2.2 Projections Event Log Formats 19
2.2.3 The Projections Performance Visualization Tool 20
2.2.4 The Performance Analysis Process Using Projections 21

Chapter 3 Scalable Analysis **23**
3.1 Scalably Finding Object Decomposition With Poor Grainsize Distributions . . 23
3.2 Scalable Idioms Based On Time Profiles 25
3.2.1 Time Profile Heuristic: Probable Load Imbalance 27
3.2.2 Time Profile Heuristic: Comparative Substructure Analysis 29
3.2.3 High Resolution Time Profiles 31
3.3 Finding Interesting Processors 34
3.3.1 Case Study: Detecting Bad Multicast Implementation 34
3.3.2 Case Study: Finding Causes For Load Imbalance 35
3.3.3 Scalable Tool Support For Finding Interesting Processors 37
3.3.4 Conclusion 41
3.4 Related Work 43

Chapter 4 Data Volume Reduction 45
4.1 Basic Approach 45
4.2 Related Work 47
4.3 Quantifying The Problem 49
4.4 Applying k-Means Clustering To Performance Data 52
4.4.1 The k-Means Clustering Algorithm 52
4.4.2 Important Choices For k-Means Clustering 55
4.4.3 Choosing Representative Processors 57
4.5 Post-mortem Versus Online Data Reduction 59
4.6 Case Study: NAMD Grainsize 65
4.6.1 NAMD - NAnoscale Molecular Dynamics 65
4.6.2 Experimental Methodology 67
4.6.3 Results 70
4.7 Future Work: Extensions To The Basic Approach 74
4.7.1 Choosing Data Subsets By Phase 74
4.7.2 Considerations For Critical Paths 76

Chapter 5 Online Live Analysis 78
5.1 Related Work 78
5.2 The Converse Client-Server Interface 80
5.3 Continuous Streaming Of Online Performance Data 80
5.4 Live Parallel Data Collection Of Performance Profiles 82
5.5 Case Study: Long-Running NAMD STMV Simulation 88
5.6 Future Work 95

Chapter 6 What-If Analysis Through Simulation 96
6.1 Related Work 97
6.2 The BigSim Simulation Framework 99
6.2.1 BigSim Emulator 100
6.2.2 BigSim Simulator 101
6.3 Performance Analysis Using Fewer Processors 103
6.3.1 Reducing Number Of Procesors Used At Emulation Time 104
6.3.2 Number Of Processors Used At Simulation Time 105
6.4 General Methodology For Hypothesis Testing 106
6.5 Analyzing Changes To Network Latency 106
6.5.1 Case Study: Seven-point Stencil Computation 107
6.5.2 Validation With Seven-point Stencil Computation 109
6.6 Analyzing Hypothetical Load Balancing Changes 120
6.6.1 Case Study: Simple Load Imbalance Benchmark 122
6.6.2 Validation With Simple Load Imbalance Benchmark 124
6.7 Future Work: Variations In Object-to-Processor Mapping 128

Chapter 7 Extending Scalability Techniques To Third Party Tools 130
7.1 Performance Call-back (Event) Interface 130
7.2 Integrating An External Performance Module 133
7.2.1 TAU Profiling Integration 133
7.2.2 Instrumentation Overhead Assessment 134
7.2.3 Experimental Results On Current Machines 137
7.3 Scalability Benefits Of Integration 137

Chapter 8 Conclusions 139

References 142

List of Tables

4.1 Total volume of trace data summed across all files. **ApoA1** is a NAMD simulation with 92k atoms, **F1-ATPase** simulates 327k atoms while **stmv** simulates 1M atoms. This data was generated on the XT3 at PSC. 51
4.2 Total volume of trace data summed across all processors in **stmv** Projections logs with different Particle Mesh Ewald (PME) long-range electorstatics configurations. The data was generated on an XT5 at NICS. 52
4.3 Number of non-empty clusters found by clustering algorithm when varying the number of initial seeds uniformly distributed in the sample space The * indicates that processor 0 was alone in its own cluster. 71
4.4 Reduction in total volume of trace data for **stmv**. The number of processors selected in the subsets are 51, 102, 204 and 409 for 512, 1024, 2048 and 4096 original processors respectively. 71
4.5 Reduced dataset quality by proportionality based on total height of histogram bars. 72
4.6 Measure of data reduction quality through number of least idle processors retained in reduced datasets for 2,048 processors. 73
4.7 Measure of data reduction quality through number of least idle processors retained in reduced datasets for 4,096 processors. 74

5.1 Overhead of collecting utilization profile instrumentation. 92

6.1 Experimental setup for baseline latency tolerance experiments executed on 48 processors. 110
6.2 Comparison of the *SimpleImbalance* benchmark average iteration times with a dummy load balancing strategy on 100 processors. 127
6.3 Comparison of the *SimpleImbalance* benchmark average iteration times with a greedy load balancing strategy on 100 processors. 127

7.1 Overhead of performance modules (microseconds per event). 136

List of Figures

2.1 User view of the CHARM++ adaptive run time system with migratable objects on the left versus a possible system mapping of the same migratable objects on the right. . . . 11
2.2 Partial interface code showing how Chares encapsulate entry methods. 12
2.3 A basic illustration of the runtime scheduler implementation on each processor. 12
2.4 7 AMPI "processes" implemented as user-level threads bound to CHARM++ chare objects which are then bound to 2 actual processors for execution. 14
2.5 Adaptive Overlapping 15
2.6 *Overview*: A summary of overall parallel structure. 20
2.7 *Usage Profile*: Processor utilization of various activities. 20
2.8 Timeline visualization of processor activity. 21

3.1 Histogram of activity grainsizes in a parallel discrete event simulation. 25
3.2 Low resolution time profile display of the structure of parallel event overlap in an 8,192 processor run of OpenAtom at 100 ms intervals. 26
3.3 Higher resolution time profile display of the same 8,192 processor run of OpenAtom at 10 ms intervals. 27
3.4 Time profile of a ChaNGa time step presenting its event-overlap substructure and exhibiting visual cues that point to possible load imbalance. 28
3.5 Time profile of ChaNGa time step showing poorer performance after the application of a greedy load balancing strategy. The substructure hints at poorer communication performance as a result of the attempt to balance the load. . . . 29
3.6 Time profile of ChaNGa time step showing the result of applying a load balancing strategy that also took communication into account. 30
3.7 Very high resolution time profile display of 8 NAMD time steps with high-detail substructure at 100 microsecond intervals. 31
3.8 Correlation of OpenAtom time-profile and communication-over-time graphs. . 33
3.9 Zoomed-in view of communication-over-time graph showing the number of messages received for OpenAtom at 10 ms intervals. 34
3.10 A small sample of processor timelines out of 1,024 showing (red) events with inefficient multicasts. 35
3.11 A small sample of processor timelines out of 1,024 showing much shorter (red) events with improved multicasts and a corresponding improvement in overall performance. 36

3.12 An approximately 800-processor slice of Projections processor usage profiles out of 1,024 processors worth of data. 37
3.13 An Extrema Tool visualization showing the usage profiles of only the top 20 least-idle processors. 38
3.14 Time Profile showing deep troughs between computation iterations, indicating possible load imbalance. 40
3.15 The Extrema Tool presenting the top list of least idle processors. 41
3.16 Timelines of the top two least idle processors as well as processors that communicate with spanning tree activity. 42

4.1 Basic sequential k-Means Pseudocode. 53
4.2 An example showing a 3-step convergence with X and Y as the initial cluster seeds and eventually becoming the cluster centroids. 54
4.3 Parallel k-Means Initialization Algorithm on the root processor. 60
4.4 Parallel k-Means Initialization Algorithm on all processors hosting performance data. 61
4.5 Parallel k-Means Iterative Algorithm on the root processor. 62
4.6 Parallel k-Means Iterative Algorithm on all processors hosting performance data. 63
4.7 Overhead of applying parallel k-Means implementation on NAMD **stmv** performance data. 64
4.8 NAMD 2 hybrid force/spatial decomposition. Atoms are spatially decomposed into *patches*, which are represented on other nodes by *proxies*. Interactions between atoms are calculated by several classes of *compute objects*. 66
4.9 Parallel structure of NAMD 67
4.10 Histogram plot of **stmv** in Projections for 4,096 processors. The vertical bars show the number of occurrences of CHARM++ entry methods, distinguished by their colors, that took a certain amount of time to execute. The first bar shows the number of entry methods that executed for 0.1 ms to 0.2 ms, the second for 0.2 ms to 0.3 ms, etc 68
4.11 Histogram plot of **stmv** with poor-grainsize in Projections for 4,096 processors. Note the shift of the number of CHARM++ entry method occurrences rightward where the bins represent longer execution times. 69
4.12 Histogram plot of **stmv** with poor-grainsize in Projections using only the 409 logs from the reduced dataset. 70
4.13 Simple example of time-variation that will confuse a clustering algorithm into believing all 4 processors behave similarly based on metrics recorded between time 0t to 2t. 75
4.14 Application of data volume reduction techniques to individual execution phases. 76

5.1 An overview of the Utilization Profile Tool. The tool is comprised of two separate mechanisms. The first mechanism (A) periodicially gathers performance data in the parallel runtime system. The second mechanism (B) allows a visualization client to retrieve the previously buffered performance data from the parallel program. 83

5.2 Compressed utilization profile format. 85
5.3 A screenshot of the streaming view in our tool. This view represents 10 seconds of execution during startup for NAMD on 1,024 processors running the STMV molecular system. 89
5.4 This streaming view represents 10 seconds of execution during the early steps when load balancing takes place. Two valleys, corresponding to the two load balancing steps, are clearly seen. This screenshot comes later in the execution of the same program run as in figure 5.3. 90
5.5 This streaming view represents 10 seconds of execution during the later simulation steps once a more uniform load balance has been achieved. Each bar represents the average utilization for 100ms of execution time. For this program, the timesteps are shorter than this duration and hence the utilization doesn't reach 100% in this plot. This screenshot comes later in the execution of the same program run as in figures 5.3 and 5.4. 91
5.6 This detailed view plots the full 1ms resolution utilization profile for 0.25 seconds of execution time for a NAMD STMV run on 1,024 processors. This snapshot captures steps with poor load balance. 92
5.7 This detailed view shows a later plot from the same run shown in figure 5.6, after load balancing improves the performance by shortening the time per step. 93
5.8 Sizes of the compressed utilization profiles received by the visualization client. The total bandwidth required to stream the data is thus under 12KB/second for this program, namely NAMD simulating the STMV system on 1,024 processors. 94

6.1 BigSim Structure and Output. 100
6.2 Dependent Execution Block. 101
6.3 Computation interaction between data elements at each seven-point stencil iteration. At iteration $i + 1$, element $E0$ gets the value of $(E0 + E1 + E2 + E3 + E4 + E5 + E6) \div 7$. 108
6.4 Parallel object E1 sends its data slab to object E0 to be kept as a ghost array for the iteration's computation. 108
6.5 BigSim-predicted latency tolerance trends for a 64x64x48 7-point stencil computation code. 111
6.6 BigSim-predicted latency tolerance trends for the 256x256x192 7-point stencil computation code. 112
6.7 Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation with a simulated latency of 1 us. 113
6.8 Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation at virtualization factor 8 with a simulated latency of 2 ms. 114
6.9 Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation with a simulated latency of 5 ms. 115
6.10 Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation at with a simulated latency of 10 ms. 116

6.11 Visualization of timeline and communication behavior for the 64x64x48 7-point stencil computation at a latency of 1 us. 117
6.12 Visualization of timeline and communication behavior for the 64x64x48 7-point stencil computation at a latency of 100 us. 118
6.13 Histogram visualization of activity grain sizes for the 64x64x48 and the 256x256x192 7-point stencil computation experiments. 119
6.14 Illustration of object placement and execution of the basic *SimpleImbalance* application. 123
6.15 Illustration of object placement and execution of the *SimpleImbalance* application with the *GreedyLB* load balancing strategy applied. 124
6.16 Projections timeline showing the baseline performance profile of the benchmark with deliberate load imbalance. 125
6.17 Projections timeline showing what happens if the Greedy load balancing strategy is applied. 126
6.18 Projections timeline showing what happens if the Rotate load balancing strategy is applied. 127

7.1 Simplified fragment of framework base class. 132
7.2 CHARM++ runtime event transition. 134
7.3 TAU integration with CHARM++ framework. 135
7.4 Instrumentation overhead for TAU and Projections with NAMD running on Ranger. 138

Chapter 1

Introduction And Motivation

Modern parallel machines are getting larger. In the June 2009 list of the top 500 most powerful supercomputers [1], the fastest possessed 129,600 processor cores running at a peak performance of 1,456.70 GigaFLOPS. Meanwhile, the largest machine weighed in at 294,912 cores. The processor growth trends are crystal-clear when one compares the bottom 100 machines in the June 2009 list with the one for June 2008. In June 2008, most machines in the bottom 100 had fewer than 2,000 processor cores. By June 2009, only a couple had fewer than 2,000 processor cores.

Bigger and faster parallel machines bring both opportunity and challenges to the user community. Opportunity, because of the greater computational resources made available to scientific applications. Challenges, because attaining good application performance on larger machines can be non-trivial. Achieving good performance for complex scientific applications require good tool support for performance analysis and tuning. However, as applications are scaled to execute on larger and larger processor counts, many traditional idioms guiding performance tool usage to study these applications break down in the face of the large performance datasets gathered for the purpose.

This thesis is motivated by the challenges to the scalable conduct of performance analysis for applications executed on large numbers of processors. The reasons behind the limitations on performance tool effectiveness are explored. Techniques are introduced to overcome some of these limitations and new scalable analysis idioms are investigated.

In this chapter, we begin by presenting a detailed context for the field of performance anal-

[1] http://www.top500.org

ysis and tuning for parallel applications in general. We discuss the motivations for scaling applications to larger numbers of processors and the resulting impact it has on the effectiveness of tools use to study the performance of those applications. Existing literature on the wide range of performance tools and the effort to address tool scalability are then discussed. Finally, we detail the main objectives of the thesis.

1.1 Parallel Performance Tuning Tools

Performance tools are employed to study the parallel performance of large complex applications. A large class of such applications are scientific simulation codes that include molecular dynamics simulations, rocket propellent simulations and cosmological simulations amongst many others. Performance tools enable *performance studies* to be conducted on *performance data* captured during an application's execution. Performance studies generally aim to find and correct performance bottlenecks in the parallel code. Studies for this purpose are also refered to as "performance debugging". In any complex scientific application, non-trivial interplay between computation, communication across processors and even computational dependencies within a processor can make the application suffer unexpectedly poor performance.

Performance debugging can be conducted for an application which is executed on a fixed number of processors. The purpose is to tune the application to run faster on the same number of processors. Performance tools are also used to understand the effects on the performance of a parallel application when it is *scaled* to larger numbers of processors. The purpose in this case is to identify root causes inhibiting performance so as to eliminate or mitigate the negative effects in order for the application to scale efficiently to higher numbers of processors. Allowing applications to scale efficiently is important as there are a number of reasons that drive scientists to scale their codes:

1. to run a particular simulation input faster so they may produce results faster; or

2. to run a particular simulation input faster so they may produce results for longer simulation observations (e.g. a protein folding over a simulated time of 1 us instead of 100 ns) within the same execution time constraint; or

3. to run larger or more detailed simulations that may not otherwise be possible on smaller numbers of processors because of memory or execution time constraints.

For the purposes of our discussion on performance tool effectiveness, we define *performance space* to be a 4-dimensional domain, inherent in every parallel application. The 4 dimensions of performance space are:

1. the continous domain of *elapsed execution time* of the application.

2. the discrete domain of *processors* or threads used in the execution.

3. the discrete domain of the *types of activities* (e.g. functions calls, messaging activity) that can occur.

4. the discrete domain of the *types of metrics* that can be recorded for the activities that occur (e.g. time spent, number of bytes sent, number of level 1 cache misses).

We define an *instrumented application* to be one which captures and generates performance data. Performance data can be captured in many different ways in the context of performance space (see Section 1.3 for various well-known systems of each). For typical *statistical profiles*, a counter is maintained on each processor for each activity that can occur in an application. As the application executes, it is sampled at some regular time interval. At each sample, the current activity's counter is incremented. Thus, at the end of the application, one can compute an approximate breakdown of the proportion of time spent by each activity on each processor over the application's total execution time. *Direct measurement profiles* perform some accumulation operations on various performance metrics when encountering activity triggers or *events*. Pairs of events can represent the start and end of *activities* in the application. Different profiling tools

can perform many different forms of metric accumulation. Finally, *event trace logging* typically records all the details of pertinent performance metrics when encountering activity triggers. The design decisions behind the different methods of capturing performance data are usually based on the trade-off between detail and data volume.

1.2 Challenges To Performance Tool Effectiveness Due to Application Scaling

We can loosely quantify the measurement of effectiveness for performance tools based on the time-to-solution for specific performance debugging session. The time-to-solution covers multiple components:

1. the time taken for an instrumented application to be executed at the necessary processor counts;

2. the time taken to produce performance data for consumption by a performance tool. This component includes the writing of data to disk and transfer of the written logs, if necessary, to a remote machine where the performance tool is executed;

3. the time taken for a performance tool to acquire the necessary performance data and process it for analysis in response to a request by a human analyst. This component includes the reading of log data from the disk;

4. the time taken for an analyst to locate specific performance problems given some performance information (visual or otherwise) provided by the tool. This involves an iterative process where the analyst explores alternative hypotheses, and develop an intuitive understanding of the performance issues by studying multiple views and analysis supported by the tool;

5. the time taken to attempt to fix the performance problem. This can be as simple as changing application configuration parameters or as complicated as application code modification;

6. the time taken to re-execute the application to validate that an attempted solution has worked.

We now discuss how application scaling impacts the performance space defined above. We will then discuss how this impact on performance space translates to impact on performance tool effectiveness based on the list of time-to-solution components.

There are two forms of application scaling. The first is known as *strong scaling* where an application runs the same simulation input on larger numbers of processors. The total computation work for the simulation across all processors, excluding parallel overheads, typically remains the same. The second is known as *weak scaling* where an application runs larger or more detailed simulation inputs on larger numbers of processors. The total computation work for the simulation across all processors is increased for weak scaling.

For both strong and weak scaling, the number of processors in the processor-dimension of performance space is increased. This directly translates to a larger exploration space for an analyst when attempting to make sense of the performance information. Hence human analysis time and performance tool response time may be affected. It also means the application potentially spends a longer time waiting in the queue of a supercomputing facility as larger processor resources are required.

The number of activity instances over time also increases for both forms of application scaling. In the case of strong scaling, this can be attributed mostly to more communication events overall. In the case of weak scaling, we have additional communication events as well as activities due to additional work for the larger simulation input. Depending on the way performance data is captured, this can impact overall performance data volume which in turn affects the effectiveness of a performance tool. We can now identify broad categories for techniques to

enhance the effectiveness of a performance tool. These include:

1. addressing performance *tool scalability* with respect to performance data volume. Scalability here refers to a performance tool's ability to present usable and useful performance data in the face of increasing data volume.

2. addressing time it takes for performance data to become *available* to a performance tool.

3. addressing the *turn-around time* for performance hypothesis experiments. This includes the time it takes to schedule and execute an application on a supercomputer to generate performance information for performance debugging or validation purposes.

1.3 Background

There are many different performance analysis and visualization tools that work on different programming models, use various forms and formats of performance data, and adopt different ways of presenting or analyzing recorded performance information.

In this section, we perform a quick sweep of the various tools to get a flavor of what is involved in performance analysis research. Detailed discussions of related research are deferred to the relevant related work sections in subsequent chapters.

Profile logs capture some summary of performance metrics, usually over an application's lifetime. Profiles can be generated by statistical sampling methods like that used by gprof [31]. Statistical profiles are perhaps the most compact, showing the contribution to total sample counts by each activity, typically the function calls invoked in the application. Statistical profiles are most useful for capturing timing information about activities, they are not as useful for events nor for dealing with metrics not associated with execution time. Other profiles can be generated by methods based on direct measurement of performance metrics of pre-defined events and activities, as is done in Projections [44] summaries or TAU profiles [69] using the Paraprof tool. Wylie et. al. [84] describes a similar approach for the KOJAK/Scalasca project

by what they call runtime measurement summarisation. Profile-based tools generally do not capture dynamic performance changes to the application over time. It is, however, possible to take snapshots [55] of profiles in TAU, allowing the capture of performance changes over known user-defined phases, but at the cost of storing an additional profile for each snapshot in the buffers. Profile summaries in Projections do the same, but uses fixed time intervals instead. Profiles tend to be compact, depending on their specific format, but not as effective for the purposes of studying the nature of complex inter-processor dependencies and interactions.

For the latter, detailed event trace logs are most effective. Detailed event logs store the faithful, usually chronological, time-stamped recording of performance metrics per pre-defined event or activity (e.g. function call, message sends) encountered in the application. These logs tend to be extremely large and care has to be taken to control the generated data volume for effective post-mortem analysis. Vampir [56], Jumpshot [85], Paragraph [33], Paradyn [52], KOJAK [80], Pablo [64], Cray Apprentice^2 [19, 20] and Projections [44] are examples of tools and systems that can use various forms of detailed event trace logs.

Regarding scalability for performance analysis tools, Wolf et. al. [79] offered an excellent overview of the technical issues, challenges and broad approaches to handling large event traces. Specifically, they surveyed nine approaches implemented by various research groups: using frame-based data formats [82]; periodicity detection [24]; call-graph compression [47]; distributed analysis [11]; automatic pattern search [81]; topological analysis [6]; holistic analysis [83]; granularity reduction [54]; and statistical analysis [2]. Meanwhile, the Scalasca project [27, 26] adopts scalable approaches to analysis based on automatic analysis methods. They search for patterns indicative of performance problems specific to MPI using a distributed analysis tool on full event traces. Roth and Miller [66] focus on an online approach to automated performance analysis.

The approach in this work sometimes complements and is sometimes similar to the scalability-oriented research mentioned above, and is contemperoneous with them. The unique aspects to our work include a pragmatic semi-automatic flavor that keeps the analyst in control of all

stages of analysis, exploiting the capabilities of an adaptive runtime system, and using the extra abstractions available to performance analysis in the context of a data-driven object-oriented model of programming.

1.4 Thesis Objectives And Scope

The objective of the thesis is to develop techniques to address scalability challenges that impact the effectiveness of performance analysis idioms. We have implemented our techniques and approaches in this thesis using our performance tool named Projections for the Charm++ parallel programming model based on message-driven execution and migratable objects. Projections is heavily-used for the performance study and tuning of many complex production codes including the classical molecular dynamics simulation application NAMD [62], OpenAtom [8, 75] which simulates molecular dynamics using the Car-Parrinello ab initio method and the cosmological simulator ChaNGa [29]. Each of these production codes have been scaled to many thousands of processors. To remain relevant, any analysis tool would have to maintain its effectiveness as applications are further scaled to tens and hundreds of thousands of processors.

There are four main thrusts to the thesis research, each addressing different aspects of the challenges described in section 1.2. The first covers new tool support for scalable performance analysis idioms. To effectively find performance problems in the enlarged performance space when applications run on more processors, scalable analysis idioms are required. We will investigate how some performance analysis idioms are not scalable and how the tools and methods we developed will help.

The second is the exploration of techniques for the reduction of performance data volume without significant loss of detail from performance event traces. This reduction is possible because most of the data in an event trace tend to capture normal application behavior. The challenge in event trace compression is to retain data that highlight performance anomalies while preserving sufficient normal behavior to maintain the necessary context for an analyst to

visually explore performance space. While the techniques can be applied to performance event logs that have already been written to disk, they can more importantly be applied to performance logs still in memory at the end of an application's execution.

The third thrust explores the benefits of live analysis and interaction with running applications. We look at how live interaction can help enhance some capabilities described in the research thrust for data reduction. At the same time, live interaction is an enabling mechanism for the online streaming of performance data as the application executes. It enables new and powerful idioms for studying application performance at large scales, allowing long-running behavior patterns to be revealed. It also improves the timeliness of performance data delivery to a performance tool.

Finally, we target methods for the reduction of job turn-around time. This idea is based on the observation that performance analysis is usually an iterative process where hypotheses are tested through re-execution and re-analysis. This can take a prohibitively long time on the job queues on typical large-scale machines, especially for the study of an application's full-machine scaling. We explore simulation-based methods that enable analysts to test performance hypotheses using far fewer processors than would otherwise be necessary if an application had to be re-executed. These methods also have the added benefit of significantly reducing the usage of precious computational resource allocation units.

Chapter 2

Software Infrastructure

In this chapter, we describe the basic software infrastructure employed to demonstrate the performance tool scalability and effectivenes techniques discussed in this thesis. The CHARM++ runtime system implements the message-driven object-oriented parallel programming paradigm used for our studies. Adaptive MPI is an implemention of the message-passing paradigm that makes use of many of the adaptive features provided by the CHARM++ runtime system. The Projections performance analysis framework is part of the CHARM++ runtime system. It provides the necessary capabilities for the instrumentation, generation and visualization of performance data.

2.1 Charm++

CHARM++ [40] is a portable C++ based parallel programming language based on the *migratable object programming model* and resultant *virtualization* of processors. In this model [42], a programmer decomposes a problem into N *migratable objects* that will execute on P processors, where ideally $N>>P$. The application programmer's view of the program is of the migratable objects and their interactions; the underlying runtime system keeps track of the mapping of the migratable objects to processors and performs any remapping that might be necessary at run-time. The programmer, not being constrained by physical processors, is able to focus on the interaction between the work partitions and concentrate on better expression of the parallel algorithm. At the same time, the runtime system can perform efficient resource management with the large number of migratable objects. The difference between the user view

and the system's implementation are illustrated in figure 2.1. In CHARM++, migratable objects are known as *chares*. Chares are C++ objects which encapsulates special *entry methods* that are invoked asynchronously from other chares through messages. In the basic CHARM++ model, entry methods are encapsulated functions that cannot be pre-empted after they have been invoked. CHARM++ uses message-driven execution to determine which entry method, and hence chare, gets control of a processor. The partial code example in figure 2.2 highlights how this model works.

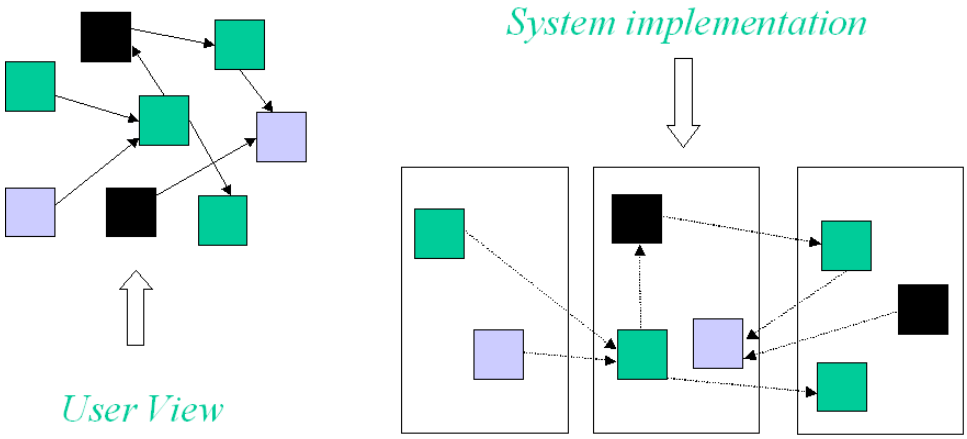


Figure 2.1: User view of the CHARM++ adaptive run time system with migratable objects on the left versus a possible system mapping of the same migratable objects on the right.

An advantage of this approach is that no chare can hold a processor while it is idle but waiting for a message. Since $N>>P$, there may be other chares on the same processor that can overlap their computation with the communicating chare. This *adaptive overlap* is enabled by a runtime scheduler on each processor. The scheduler picks up incoming messages and queues them in a local buffer. When an entry method completes on the processor, the local scheduler will invoke the appropriate handler routine for the next entry method associated with the message at the head of the queue. If the queue is empty, the scheduler enters into an *idle state*, waiting for new messages from other processors. The operation of the runtime scheduler on each processor is illustrated in figure 2.3.

```
mainmodule pingpong {
  ...
  chare Ping {
    // constructor
    entry Ping(void);
    // generates a PongMsg and calls recv() on a
    // remote object of chare class Pong
    entry void recv(PingMsg *);
  };

  chare Pong {
    entry Pong(void);
    // generates a PingMsg and calls recv() on a
    // remote object of chare class Ping
    entry void recv(PongMsg *);
  };
  ...
}
```

Figure 2.2: Partial interface code showing how Chares encapsulate entry methods.

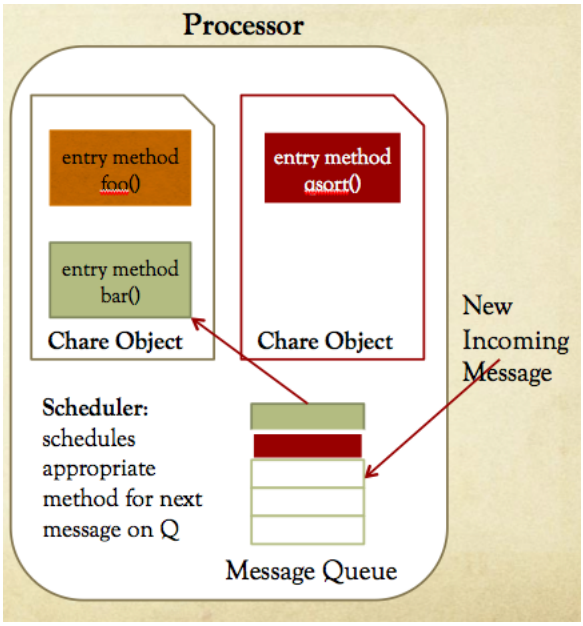


Figure 2.3: A basic illustration of the runtime scheduler implementation on each processor.

CHARM++ is actively used in a number of major real-world scientific applications [60, 75, 29] that demand high scalability for some of the phenomena the scientific community wishes to study.

2.1.1 Adaptive MPI

The Message Passing Interface (MPI) [1] is a library specification for message-passing as the paradigm for communication and synchronization between processors in a parallel program. It is a standard that defines an application programmer's interface for messaging but leaves flexible the details for specific machine implementation. The interface was also designed with language independent semantics with convenient C and Fortran bindings. Since its introduction, it has become a widely accepted standard for parallel programming over a wide range of hardware.

Adaptive MPI (AMPI) [35] tries to add important adaptivity capabilities available to CHARM++ to MPI's rich feature set. These capabilities will be highlighted in sections 2.1.2 and 2.1.3. As of this writing, AMPI is compliant with the MPI 1.1 standard. AMPI implements its MPI processes as user-level threads bound to chares as illustrated in figure 2.4. This binding of user-level threads to migratable chare objects is key to automatic load balancing support for AMPI programs. Message passing between AMPI processes is implemented as communication between chares. Send calls in AMPI map directly to equivalent CHARM++ send semantics. Receive calls, however, have to be implemented as CHARM++ entry methods to bridge the semantic gap between the CHARM++ and AMPI models for the support of basic MPI 2-sided communication. MPI 1-sided communication, introduced in the MPI 2.0 standard, are currently not supported.

Standard MPI programs divide the computation onto P processes and the typical MPI implementation assigns and executes each of these processes on its own physical processor. An AMPI programmer would divide the computation into a V virtual processors (user-level thread) which then gets mapped onto P physical processors. Unlike the scenario of writing an MPI for

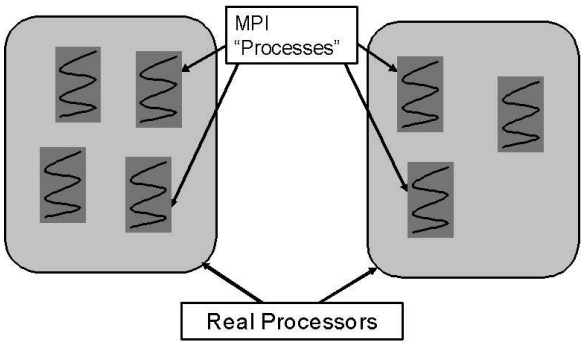


Figure 2.4: 7 AMPI "processes" implemented as user-level threads bound to CHARM++ chare objects which are then bound to 2 actual processors for execution.

a typical MPI implementation, an AMPI programmer would not worry about the number of physical processors, similar in nature to how a CHARM++ programmer would concentrate on the decomposition rather than the number of processors the application would run on. When V = P, the AMPI program would execute in the same way it would for typical MPI implementations. To exploit the adaptivity capabilities of the underlying CHARM++ runtime system, V needs to be significantly larger than P.

2.1.2 Adaptive Overlap

One important feature of the CHARM++ runtime system is its natural ability to dynamically schedule work based on incoming messages to a processing thread. This means that the computation work from two independent parallel software components could be overlapped dynamically. When one software component becomes idle on a processor, work from another could be scheduled. The adaptive overlap can also be refined by the ability to set different message priorities for different software components. This will have significance in the context of the work on continuous performance data streaming in Chapter 5.

To highlight this feature, consider the scenario illustrated in Figure 2.5 (This example is taken from [41]). There are three parallel components A, B and C spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style

programming model, the programmer has to choose one component between B and C to call from A first on all processors. Only when the first chosen component returns can A call the remaining one on all processors. This model can be inefficient because when one component idles the CPU, for example, when waiting for communication to complete, other components are not allowed to take over and do useful computations, even though there is absolutely no dependence between the components.

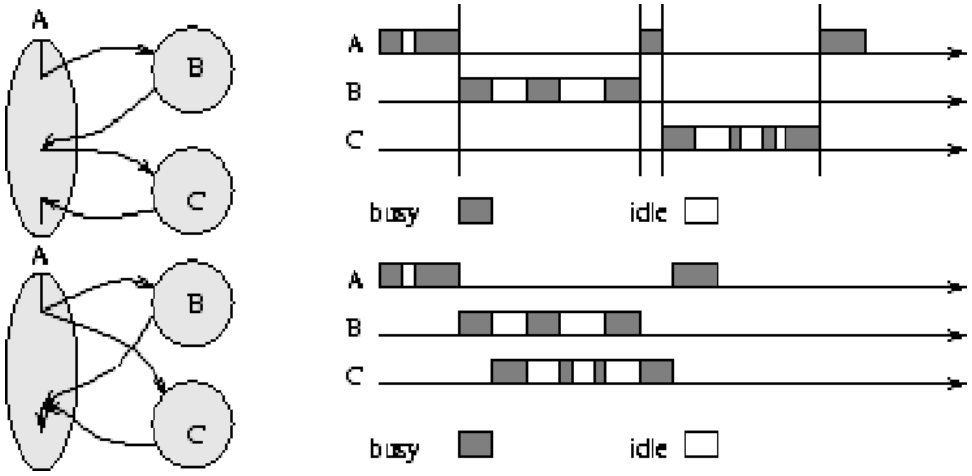


Figure 2.5: Adaptive Overlapping

With the CHARM++ runtime support, A can invoke B on all the processors, initiating computation and sending out messages, and since there is no dependence between B and C, A can also start off C in a similar fashion, and thereby components B and C can interleave their execution. When one component blocks due to communication or load imbalance, the other component can automatically overlap the idle time with computation, based on the availability of data, as illustrated in Figure 2.5. With non-blocking calls and careful programming, the programmer could achieve the same effects with MPI, but the price is additional programming complexity and a loss of modularity.

2.1.3 Automatic Load Balancing

One of the most prominent benefits of an adaptive runtime system like CHARM++ with migratable objects is the capability of dynamic load balancing by migrating objects across processors. This capability benefits parallel applications because of the *principle of persistence*. The principle of persistence is an empirical heuristic about parallel program behavior over time. For most parallel applications expressed in terms of virtualized processors, the computation loads and communication patterns tend to persist over time. This heuristic applies to many programs with dynamic behavior, including those using adaptive mesh refinement with abrupt but infrequent changes, to those simulating molecular dynamics with slow and gradual changes over time.

Based on the principle of persistence, the CHARM++ runtime system employs a measurement-based load balancing scheme. The load balancer automatically collects statistics on each object's computation loads and communication patterns, and using the collected load database, the runtime system decides on when and where to migrate the objects by adopting a load balancing strategy. A variety of such strategies have been developed for applications with different dynamic behaviors. Some strategies are centralized, others fully distributed. Some use only computation load when making a decision, and others take into account communication patterns and even topology of the platform. Our previous work on NAMD[43] demonstrates the significant benefits of automatic load balancing in real-life applications.

Application performance can be significantly affected by the choice of load balancing strategy and its frequency of application. In this thesis, we make use of the CHARM++ load balancing framework for two things. First, it can serve as a clear means of validating the data reduction work described in chapter 4. If in an application's run, we start with deliberate load imbalance and then fix it using an appropriate load balancing framework, any reduced dataset generated must necessarily capture the earlier load imbalance. Second, the load balancing framework in CHARM++ will serve as the instrument for re-mapping event log entries in recorded BigSim (see Section 6.2) simulation logs to test the effects of various load balancing strategies without

the need to re-execute a large-scale application run. Object migration decisions by different load balancing strategies can be made using the load and communication information of the original large-scale run. These decisions can then be applied to transform BigSim event logs such that, when simulated, the performance information generated would reflect behavior as if a different load balancer had been used in the original application run.

2.2 The Projections Performance Analysis And Visualization Framework

The *Projections Analysis Framework* [44] consists of an instrumentation and data generation component within the CHARM++ runtime as well as a java-based visualization/analysis tool. Projections was designed to be a tool framework that supports the post-mortem and expert user-directed performance analysis of parallel applications. It's features have been used extensively to tune many CHARM++ applications in order to enhance their scalability and performance, especially NAMD [62, 44, 49].

Projections instrumentation is fully automated by default. Specifically, the runtime system knows when it is about to schedule the execution of a particular method of a particular object (in response to a message being picked up from the scheduler's queue), and when an object sends a message to another object. Default instrumentation code are written into the runtime system at these important abstraction points. When conducting performance analysis studies of applications, the instrumentation modules are invoked at these abstraction points, recording the necessary performance data into memory buffers at the cost of some time overhead. Timing data is typically recorded at microsecond resolutions. To ensure minimal performance impact in application runs, the instrumentation code overhead at these abstraction points can be reduced significantly when performance analysis is not required for that run-instance. The performance framework design can further allow overhead to be completely eliminated in fully optimized

production CHARM++ application binaries.

The recorded performance data (which can take multiple forms, see section 2.2.2) is typically written out at the end of the run. If it exhausts all the memory space allocated to it in the middle of a run, or if the user desires (e.g. at known global synchronization points), the data can be flushed to disk. In our experience, however, asynchronous flushing of log data causes such severe perturbation of application performance that the performance information after the first instance of such a flush becomes effectively useless.

2.2.1 Charm++ Performance Events

Gaining insight into a Charm++ application's parallel performance depends on what events are observable during execution. There are several points in the Charm++ runtime system's code base that can capture important events and information about their execution context. These include:

1. Start of an entry method.
2. End of an entry method.
3. Sending a message to an object.
4. Change in scheduler state: *active to idle* (entry method completes and no new message is available, *idle to active* (new message is available)

The first two relate directly to Charm++'s logical execution model. Observing message sends provides a runtime level perspective of object interaction. Scheduler state transitions expose resource-oriented aspects of the execution. The point is that the observation of multiple events at different abstraction levels is needed to get a full characterization of performance in a parallel language system such as Charm++.

2.2.2 Projections Event Log Formats

Projections' *summary* format is the most compact data format. It records processor utilization for each processor over k time intervals where k is fixed. The duration, d, of these time intervals is adjusted dynamically, doubling each time the application's execution goes past $k \times d$. The size of each file is dependent only on the desired number of time-intervals k. Therefore, the total size summed across all files grows in $O(P)$ where P is the number of processors. We use runlength encoding as an additional measure to reduce the size of each file. Projections also supports an intermediate *summary detail* format. It employs the same dynamically adjusted time interval scheme for recording data as described above for the *summary* format. However, we record attribute information for each time interval such as the time spent or number of messages received separatedly for each CHARM++ entry method. Such a format is reasonably rich and useful for analysis, allowing a quick way to zoom in on a potential problem area. The size of each file is of the order of $k \times E$, where E is the number of CHARM++ entry methods in the application. However, E is relatively small, around 220 for NAMD, and depends on the static design of the application. As a result, E can be considered in practice to be a constant factor for all applications. So, the total data size grows in $O(P)$ similar to *summary* files. Like the *summary* format, runlength encoding is used to reduce the size of the file.

Projections event trace logs are written in a text format with one event per line. Some events contain more details than others. For example, we record events like the start and end of each CHARM++ entry method, every time a message was sent and each time the runtime scheduler goes idle or returns from being idle. For each event, different types of attributes may be recorded. This includes the timestamp, size of a message, the CHARM++ object id, and any performance counter information (e.g. PAPI [10]) associated with the event. Recording performance counter information is optional and the default instrumentation policy has small overhead, involving a low-cost timestamp request and accessing the in-memory instrumentation log buffers.

2.2.3 The Projections Performance Visualization Tool

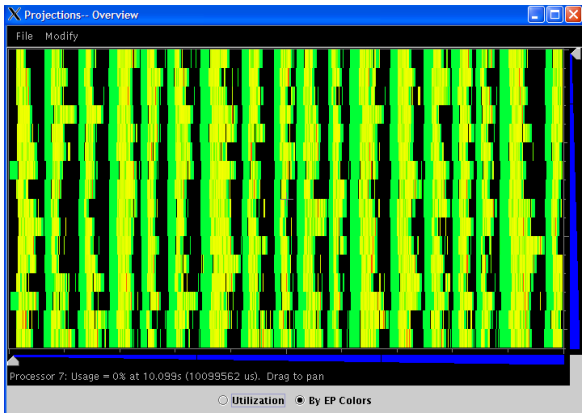


Figure 2.6: *Overview*: A summary of overall parallel structure.

The visualization component of the Projections framework is implemented in Java and has the advantage of being generally portable across many platforms. The trace logs generated by the instrumentation modules in the CHARM++ run-time are platform neutral with respect to visualization and analysis. Kalé et. al. [44] describes the important visualization tools provided by Projections and how they were used to support analysis processes employed by analysts to tune and optimize NAMD[62], a molecular dynamics simulation application. The rest of this section describes some of the commonly-used visualization support in order to give a flavor of what is available to our user-directed approach to post-mortem performance analysis.

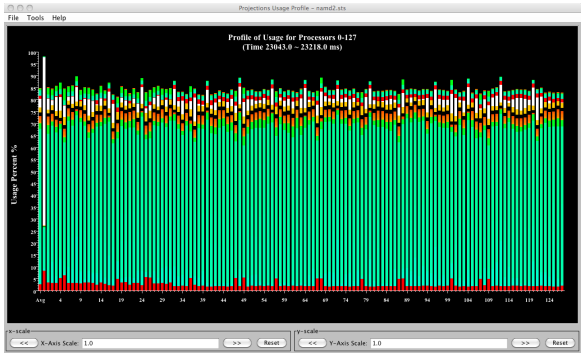


Figure 2.7: *Usage Profile*: Processor utilization of various activities.

The *overview* plot of Figure 2.6 can show the dominant activity within fixed time inter-

vals along the horizontal-axis over each processor along the vertical-axis. It presents a good summary of the overall parallel structure of the application.

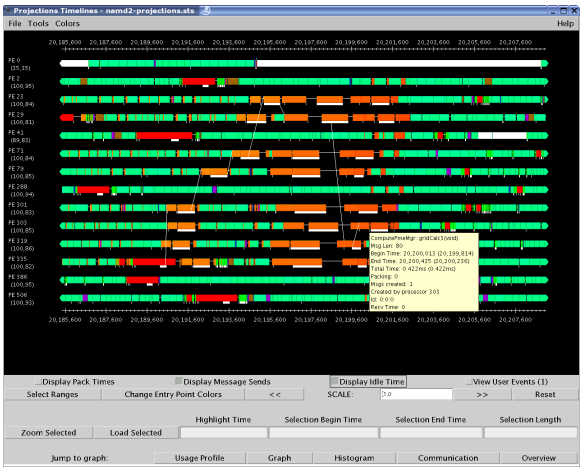


Figure 2.8: Timeline visualization of processor activity.

The *usage profile* view of Figure 2.7 displays, for each processor, the utilization percentage of all activities over a user-selected range of the application's execution time. This includes the percentage of idle time, when no useful work occurs. The left-most bar presents data averaged across all processors. This view is particularly useful for identifying the presence of load imbalance in the application, as well as its overall efficiency.

Finally, the *timeline* view in Figure 2.8 presents to the user a detailed look at the interactions between application activity and communication between processors. The horizontal time-axis marks the passage of time in the application while the vertical axis represents processors. Moving the mouse pointer over an activity pops up a small window with specific information about various performance properties of that activity (eg. number of messages sent).

2.2.4 The Performance Analysis Process Using Projections

The typical performance analysis process for a large-scale CHARM++ parallel application using Projections follows these general steps:

1. Instrument application if necessary.

2. Compile and build binaries.

3. Submit application to job submission queueing system.

4. Application generates performance data.

5. Use analysis idioms to find performance problems captured by the data.

   (a) Study application's overall performance profile.

   (b) Locate area in performance space with possible problems.

   (c) Restrict visualization to smaller and smaller areas of the performance space with higher levels of performance detail.

   (d) Identify specific problems, interpret and find solution in code, or back up if the problem was mis-identified at lower levels of performance detail.

6. Update application code given a hypothesis about the performance problem.

7. Validate the effectiveness of the solution. Repeat steps 2 through 5. If the solution was not effective, a new hypothesis is tested via steps 6 through 7.

In general, an iterative "drill-down" approach is adopted for performance analysis using the trace-based data generated by CHARM++ or AMPI applications. This is very similar to the way trace-based performance data from traditional MPI tools are analyzed, the key differences being the activity/event semantics and the interpretation of bottlenecks. Analysts typically have to understand the underlying code structure of the application they are trying to analyze.

Chapter 3

Scalable Analysis

The art of seeking performance problems involves the use of many heuristics, application-dependent or otherwise. For each performance analysis system, the heuristics translate into a variety of tool usage idioms. These idioms depend on the tool's features to help an analyst find specific classes of performance problems from the different forms of performance data compatible with the tool. To be scalable, an idiom must express an analysis process that does not overwhelm the analyst with the information produced by those features of the tool.

Many of the common idioms used in performance analysis break down with a large number of processors. In this chapter, we set about identifying, cataloging and supporting some scalable idioms, along with the development of appropriate tools and features. We describe how powerful scalable idioms are supported by the histogram, high resolution time-profiles and a tool designed to provide scalable ways of quickly picking only the relevant detailed timelines for processors that exhibit extreme behavior. To place the utility of this work in context, note that significant parts of the work were carried out and published over the past seven years.

3.1 Scalably Finding Object Decomposition With Poor Grainsize Distributions

We define grainsize to be the amount of computation work performed per incoming message. Events that are too fine-grained suffer from high scheduling overheads while events that are too coarse-grained can hold up a processor's ability to process incoming messages for other

possibly high-priority work. Poor object grainsize distributions could also interfere with load balancing strategies' ability to produce optimal load balance.

In our experience, prior to this work, object grainsize problems are almost always stumbled upon as a side-effect of examining the detailed timelines of processors. Performance events associated with objects with grainsizes that are too coarse show up on timelines as blocks of large sizes. These visual cues were then picked up by the analyst, mostly by intuition, and examined in detail by mouse-overs to determine if the same parallel object was associated with the event blocks. Such grainsize problem discoveries often result as a side-effect from the study of poor load balance, particularly from scenarios where no load balancing strategy appears to help due to poor object grainsize distributions. Clearly, this approach to discovering grainsize problems is not scalable. If an analyst presciently sought to locate such specific problems, he or she would have to scan detailed timelines until anomalies are spotted. Even then, there is no easy way to tell how common or severe these anomalies are.

To scalably support this idiom, we developed the Histogram tool [1] in Projections to identify potential grainsize problems [43]. The histogram tool (see figure 3.1) counts the frequency distribution of different types of executed events across a range of bins of time spent by the events. This differentiation of event types is extremely helpful as it helps the analyst identify the appropriate code segments to modify if a change of event grainsize is desired. Various patterns in the display can be recognized by the analyst. For example, an observed "tail" in the distribution could mean a problem involving coarse-grained events are holding up the application, particularly if the duration of these events are significant relative to the expected execution time of a single iteration or phase. The end of section 6.5.2 provides a detailed example of how we made use of this analysis idiom to explain an observed but unexpected performance phenomenon while investigating the effects of object grainsizes on application performance.

With the histogram tool, the idiom becomes scalable. The histogram tool simply counts all events over all processors in the execution time range specified by the analyst and bins

[1] Developed with the assistance of Sindhura Bandhakavi.

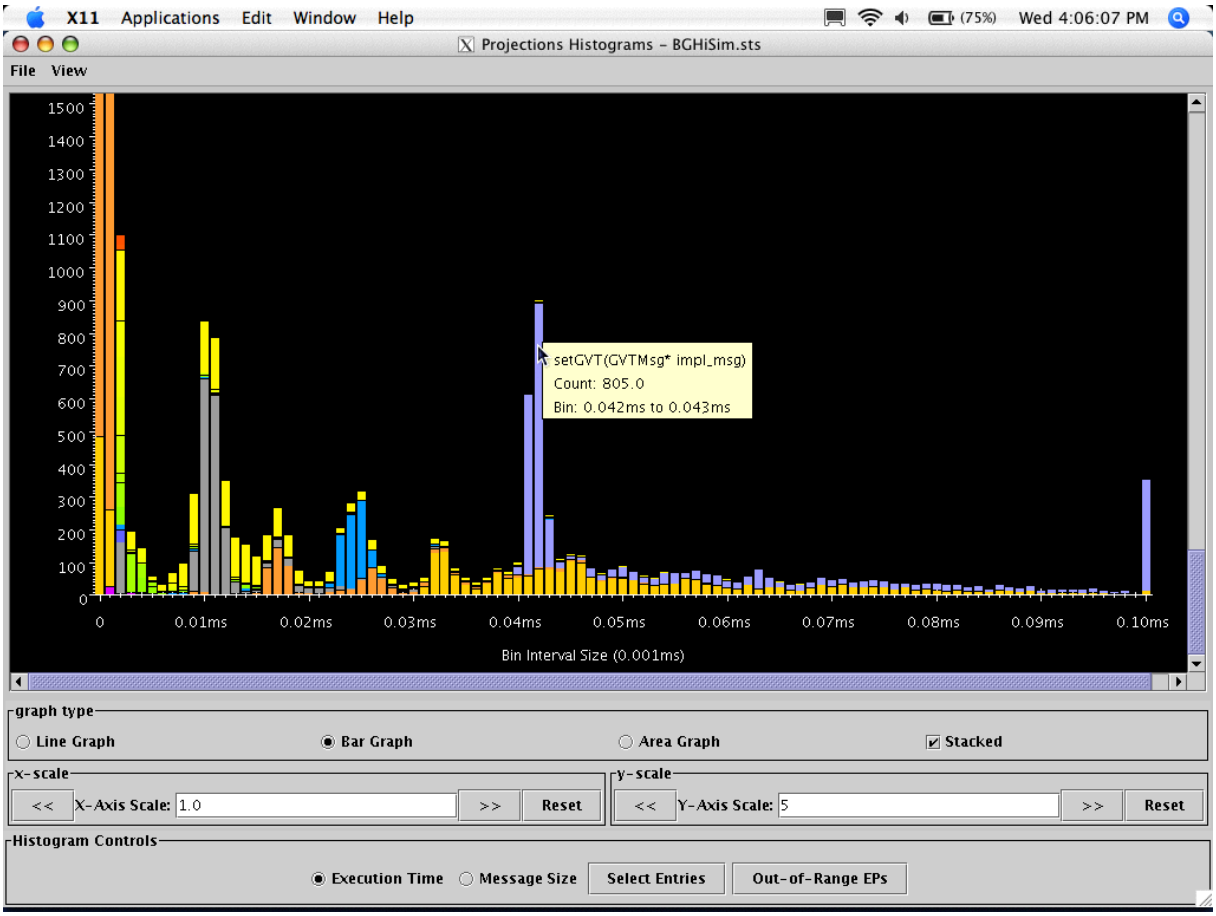


Figure 3.1: Histogram of activity grainsizes in a parallel discrete event simulation.

them accordingly. The displayed distribution plot does not require the analyst to look for any particular non-scalable details to derive any conclusions about bad object decomposition.

3.2 Scalable Idioms Based On Time Profiles

Analysis idioms employing views that show some attributes as a function of processors do not scale, as we saw earlier. However, we can study performance information along the time dimension with finer granularity, while abstracting away the processor dimension. We next describe a tool (or view) we developed to explore this idea, and the idioms and analysis heuristics based on this tool that scale well.

"Time profile" is a view of a portion of execution, where the time is broken down into a large

number of relatively small-size intervals. For example, in some applications one may use a 100 microsecond interval-size to analyze execution of several application iterations taking (say) 100 milliseconds. That leads to about 1,000 intervals, which are amenable to visual analysis. For each interval, we compute and display a bar chart that shows the time spent in different activities added across all the processors. The interval size should be adapted to the natural time-scale of an application. For example, for molecular dynamics running with a fine-grain decomposition, with a time-step of a few milliseconds, the 100 microseconds mentioned above is a reasonable interval-size, whereas for a run of quantum chemistry simulation with a time-step of 5 seconds, one may chose a few milliseconds as a time interval. The important point is that capturing the fine performance structure does not require display size proportional to the number of processors. Performance trends also flow far more naturally along the time-dimension than along the processor-dimension (i.e. the analyst can often gain new insight into application performance by comparing nearby columns).

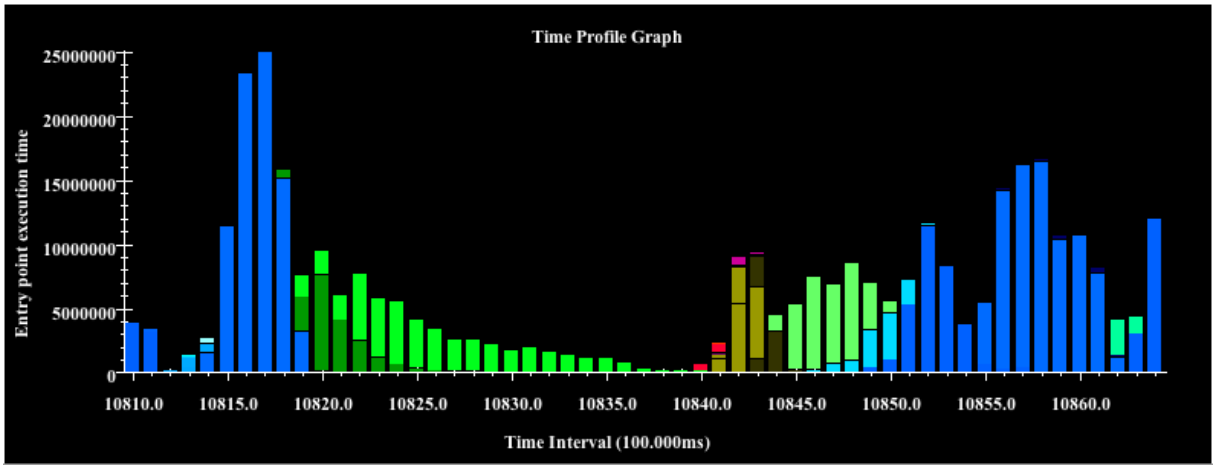


Figure 3.2: Low resolution time profile display of the structure of parallel event overlap in an 8,192 processor run of OpenAtom at 100 ms intervals.

Figure 3.2 shows a simple time-profile display, with a small number of intervals chosen to illustrate the components of the view. This was taken from a run of OpenAtom, a quantum chemistry application, running on 8,192 cores of BlueGene/L, simulating (only) 256 water molecules. For each 100 millisecond interval (which was chosen so as to show the structure

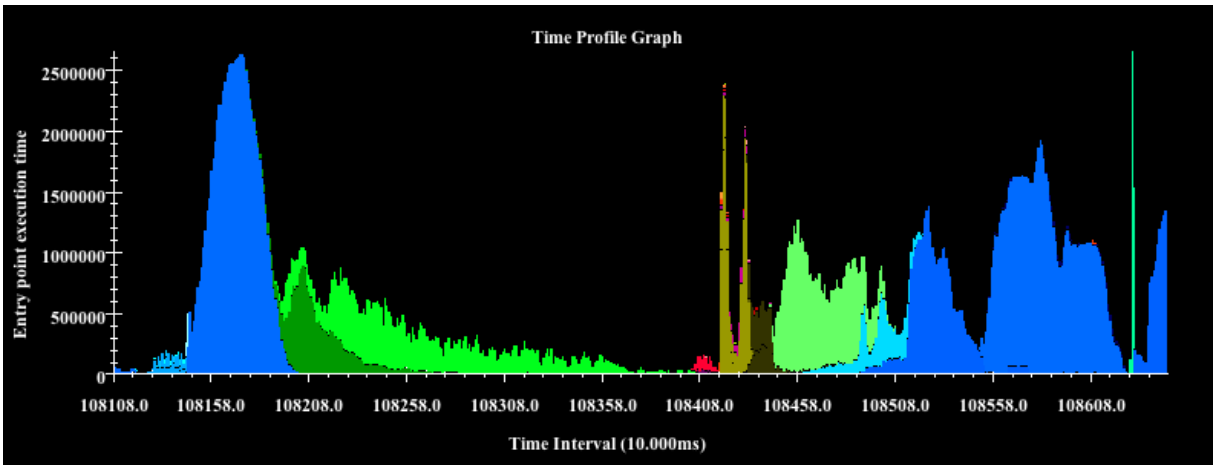


Figure 3.3: Higher resolution time profile display of the same 8,192 processor run of OpenAtom at 10 ms intervals.

of each stacked bar clearly), one can clearly see the time spent in different activities, indicated by different colors in a given bar, added across all the processors. Figure 3.3 shows a more common use of time-profile, where the time intervals are smaller, for the same performance data. Such finer view better shows variation of quantities over time.

Although the time profile view is scalable, the question remains as to what idioms and heuristics exist, that can make use of this view for scalable performance analysis. We describe a few heuristics and illustrate them with examples in the rest of this section.

3.2.1 Time Profile Heuristic: Probable Load Imbalance

When a time profile shows a trough with a relatively sharp slope to the right of this trough, but a somewhat long tail to the trough's left, there is likely to be load imbalance.

The justification for this heuristic is simple. As processors finish their work early, they go "idle", waiting for the other processors to also finish. This leads to a slope on the left side of the trough. If only a few processors are overloaded and hence contribute to a load imbalance problem, this will be visually expressed as a long tail to the slope.

A real-life example that illustrates this scalable use of time-profile involved a computational astronomy application ChaNGa, executing the Barnes-Hut algorithm. A case study similar to

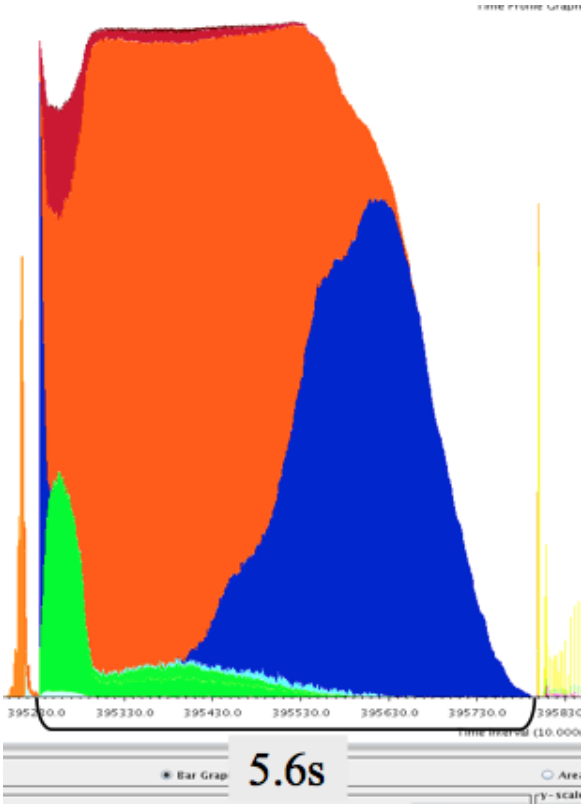


Figure 3.4: Time profile of a ChaNGa time step presenting its event-overlap substructure and exhibiting visual cues that point to possible load imbalance.

this example was published as part of broader analysis by Jetley et. al. [38] on 1,024 Blue Gene/L processors. The time profile of one time-step is shown in figure 3.4. The different colors show various components of the execution activity. We highlight four activities in the timestep, two types of computations colored blue and orange, and two communication-related functions that handle request for particles (red) and handle response to requests for particles (green). Please note that, at this point, the details of this substructure are not necessary to diagnose the potential load imbalance problem. The envelope of all four activities combined shows the characteristic trough between the activities of the time-step and a yellow-colored activity in a later phase. The envelope exhibits a gradual slope to the left of the trough that ends with a small blue tail. Because no load balancer was initially employed, it was an indication that some load balancing strategy should be used to handle the situation.

3.2.2 Time Profile Heuristic: Comparative Substructure Analysis

With time profiles available from different runs or even different time phases within the same run, it becomes possible to scalably examine detailed changes in the substructure to make comparative studies. In particular, we can identify any activity or activities that exhibit unexpected or disproportionate differences between time profiles. Changes in shape hint at a different parallel overlap between events while changes in size show changes in the time spent by events.

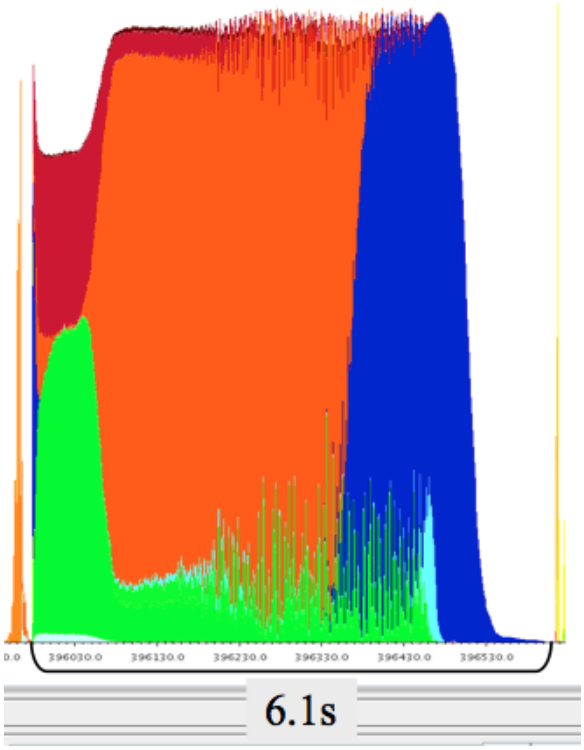


Figure 3.5: Time profile of ChaNGa time step showing poorer performance after the application of a greedy load balancing strategy. The substructure hints at poorer communication performance as a result of the attempt to balance the load.

This idiom for the utilization of time profile in scalable analysis is illustrated with the help of the same example in section 3.2.1. The time profile of the same timestep was obtained again after a greedy load balancing strategy was employed. This is illustrated in figure 3.5. Clearly, the load balancer had made a difference as the slope is now steeper at the left end of the trough. However, the execution time has actually increased, from 5.6 seconds to 6.1 seconds!

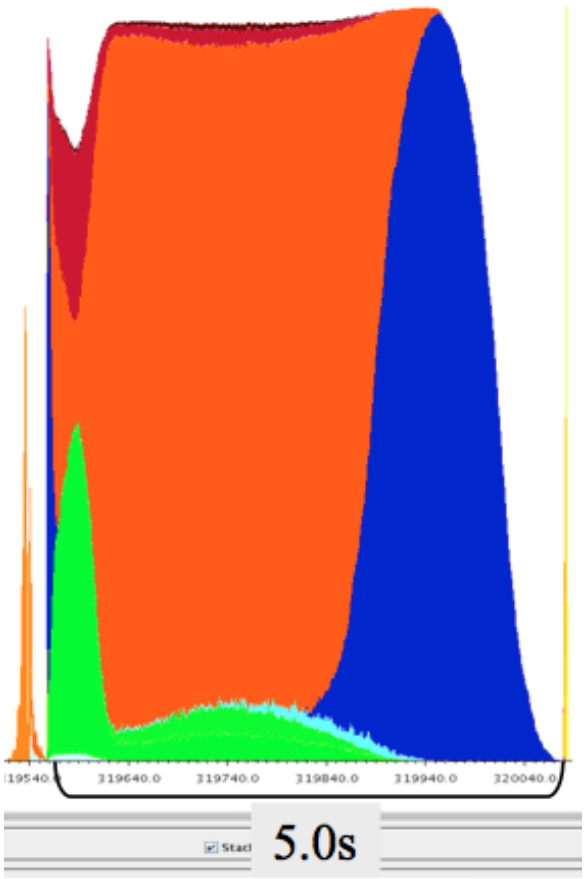


Figure 3.6: Time profile of ChaNGa time step showing the result of applying a load balancing strategy that also took communication into account.

The substructure of ChaNGa's overlapping events shown by the time profile is now handy to identify the problem. Comparing figures 3.4 and 3.5, one can see that the communication-related functions colored red and green are now taking more time. Thus, we can see that the greedy load balancer had redistributed work in such a way as to increase overall communication overheads. This analysis led to the use, and refinement, of a load balancing strategy that tries to also reduce communication by keeping chunks of objects that communicate more, closer to one another. The final time profile of the same timestep obtained with the new communication-aware balancer in shown in figure 3.6. Performance has improved to 5.0 seconds while both the envelope and substructure of the timestep appear satisfactory.

3.2.3 High Resolution Time Profiles

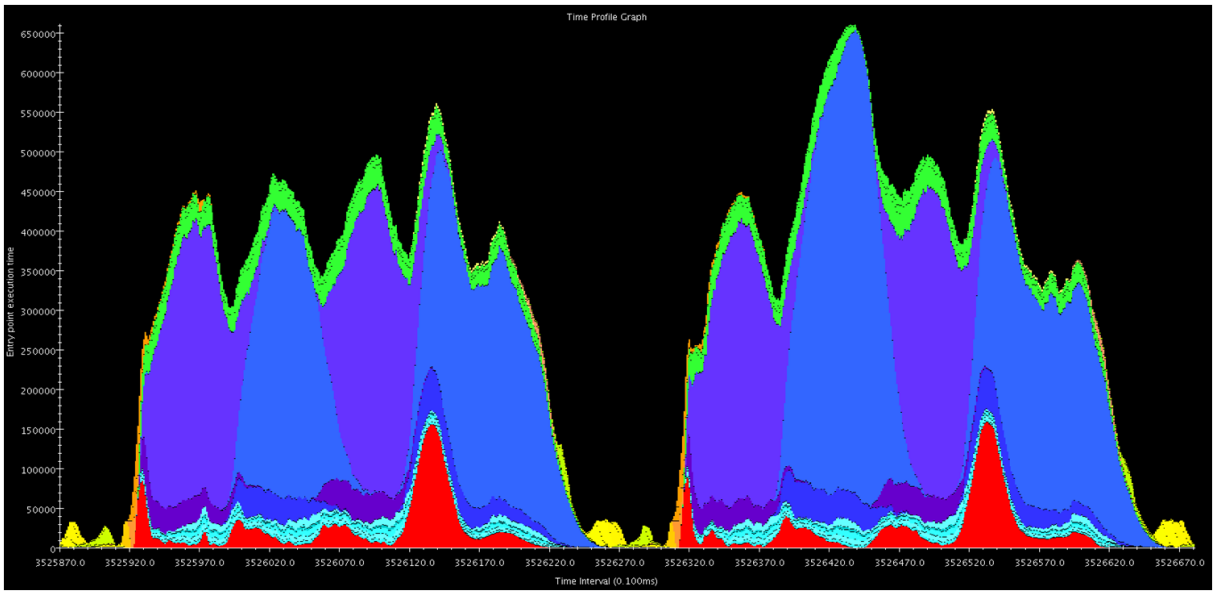


Figure 3.7: Very high resolution time profile display of 8 NAMD time steps with high-detail substructure at 100 microsecond intervals.

It is important that the time-profile view be used at a resolution that matches the time resolution of the structure of the problem. The idioms we have described can require fine-grained intervals, and our method of generating these from trace data allows us to match any required resolution. To emphasize this, we show a time-profile view of NAMD running on 8,192 processors of Blue Gene/P. In figure 3.7, the display corresponds to only 80 ms of execution time, which includes 8 time steps of NAMD. To clearly examine the substructure, we need to use the time profile tool at 100 microsecond intervals. The large valley after the first four time steps is only 5 ms wide. At the bottom of this valley can be seen the poorly overlapping substructure of several yellow-beige colored events associated with various Particle Mesh Ewald FFT computation phases. The fine-grained display clearly shows the Particle Mesh Ewald FFT phases holding up the computation. This diagnosis is not possible with any coarse-grained time-profile.

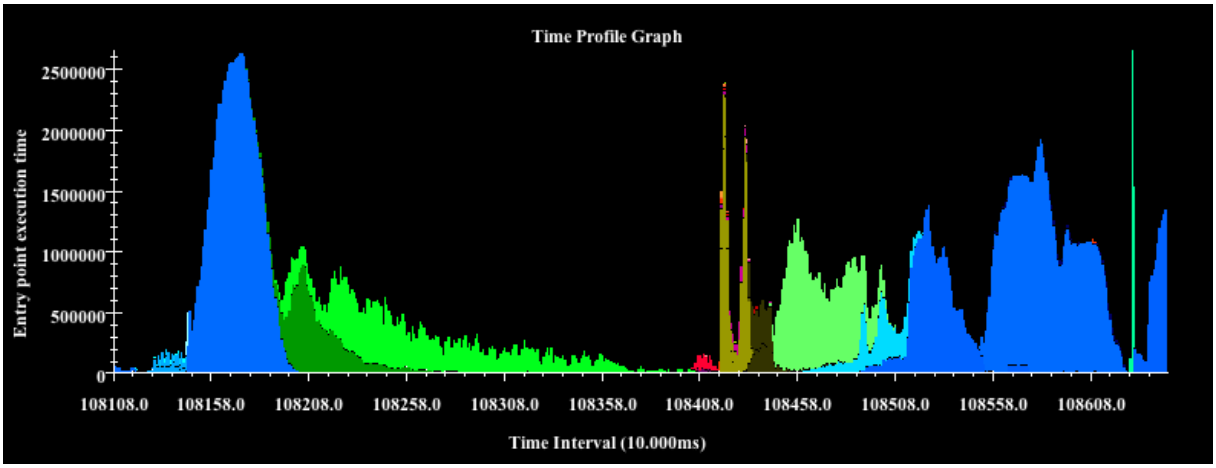
Building up on the idea that high resolution time intervals can capture interesting substructures of the application, another set of scalable idioms become feasible when we utilize the communication-over-time profile. In this view, the x-axis is time intervals, as with time pro-

files described above. The bar (the y-axis) for each interval now shows communication metrics added across all processors. There are 4 possible communication metrics:

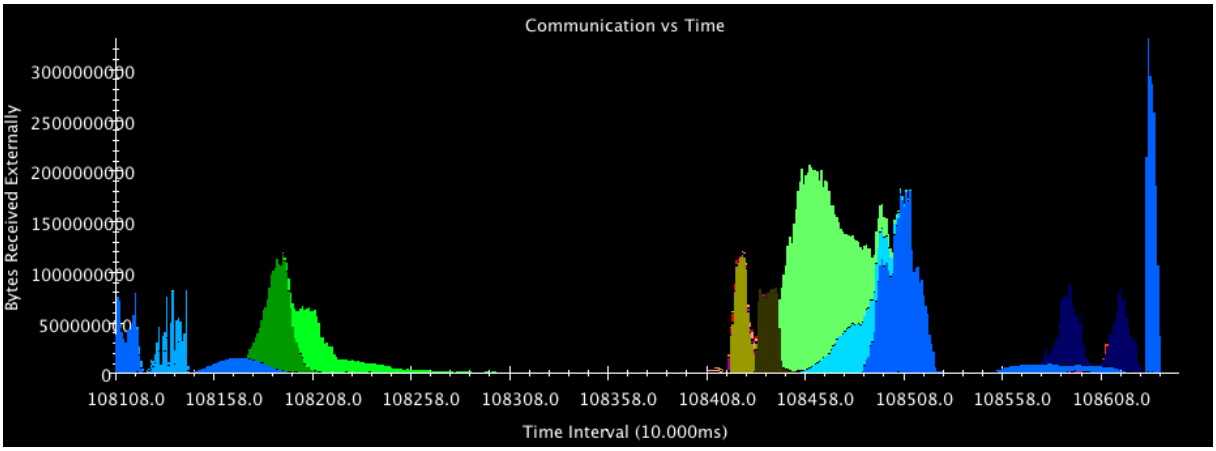
1. Communication volume sent, in bytes.
2. Communication volume received, in bytes.
3. Number of messages sent.
4. Number of messages received.

It is especially useful to use time-profile and communication-over-time plots at the same time scales and correlate their substructure along the time dimension. As an example, we show in figure 3.8 two communication-over-time graphs correlated with the OpenAtom run previously shown in the time profile figure 3.3. The hypothesis that the low efficiency in the middle time-range is caused by the communication effects can be tested at an aggregate level via these combined views. In this particular case, we see that the column of communication is large, but within the supposed limits of the machine. This points to possible contention in the network or other random interference as potential performance problems. We know the OS interference is not likely to be an issue on Blue Gene/P machines, because of its use of a light-weight kernel. As such, communication "hiccups" are a probable cause to investigate further. From the figure showing communication-over-time for number of messages received (figure 3.8(c)), we zoomed-in and increased the scale along the y-axis. The zoomed-in figure 3.9 present evidence that these "hiccups" may indeed be at play.

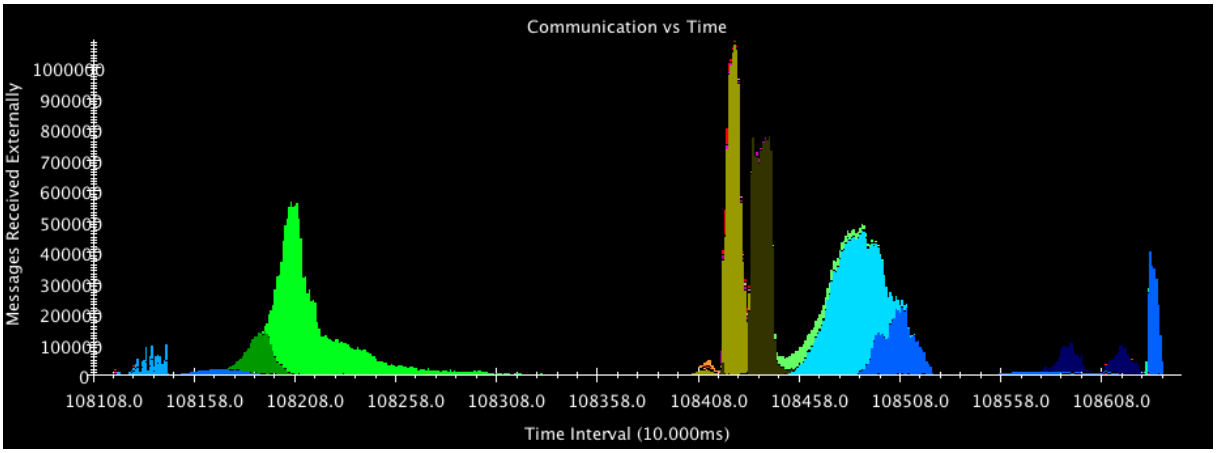
In this example, the particular performance problem in these runs and how it was eventually resolved is not pertinent here. The point is in the insight provided by the combined views in a highly scalable manner.



(a) High resolution time profile display of 8,192 processor run of OpenAtom at 10 ms intervals.



(b) Correlated communication-over-time graph showing the volume of communication received in bytes for the same OpenAtom run at 10 ms intervals.



(c) Correlated communication-over-time graph showing the number of messages received for the same OpenAtom run at 10 ms intervals.

Figure 3.8: Correlation of OpenAtom time-profile and communication-over-time graphs.

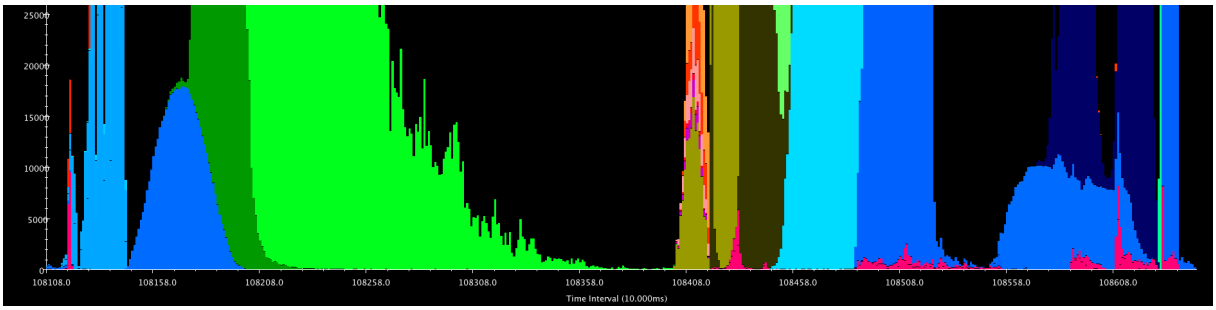


Figure 3.9: Zoomed-in view of communication-over-time graph showing the number of messages received for OpenAtom at 10 ms intervals.

3.3 Finding Interesting Processors

A recurring theme in the search for detailed performance problems is the identification of processors that look "interesting" or "unusual" while scanning the performance space. The appropriate visualization of the performance space for this purpose is typically provided by some profile like usage profiles (see figure 2.7 from chapter 2) or overviews (see figure 2.6 from chapter 2). We now consider some case studies of analysis idioms encountered that fit this theme and present the tool developed to scalably execute those idioms.

3.3.1 Case Study: Detecting Bad Multicast Implementation

A poorly implemented multicast routine revealed itself in the form of extended entry method computation events where multicasts were used [43]. Only after we had somehow picked out the processors most affected by the use of these badly implemented multicasts, were we able to detect the problem. We observed on the detailed timelines of these processors that the lengthened computation events included many communication calls. This finally provided the visual cue, illustrated in Figure 3.10, that we needed to infer the source of the problem and fix it.

Once the inefficient multicast operations were fixed, figure 3.11 showed an improvement in performance for the events affected on the same processors. Overall application performance was also improved as a result.

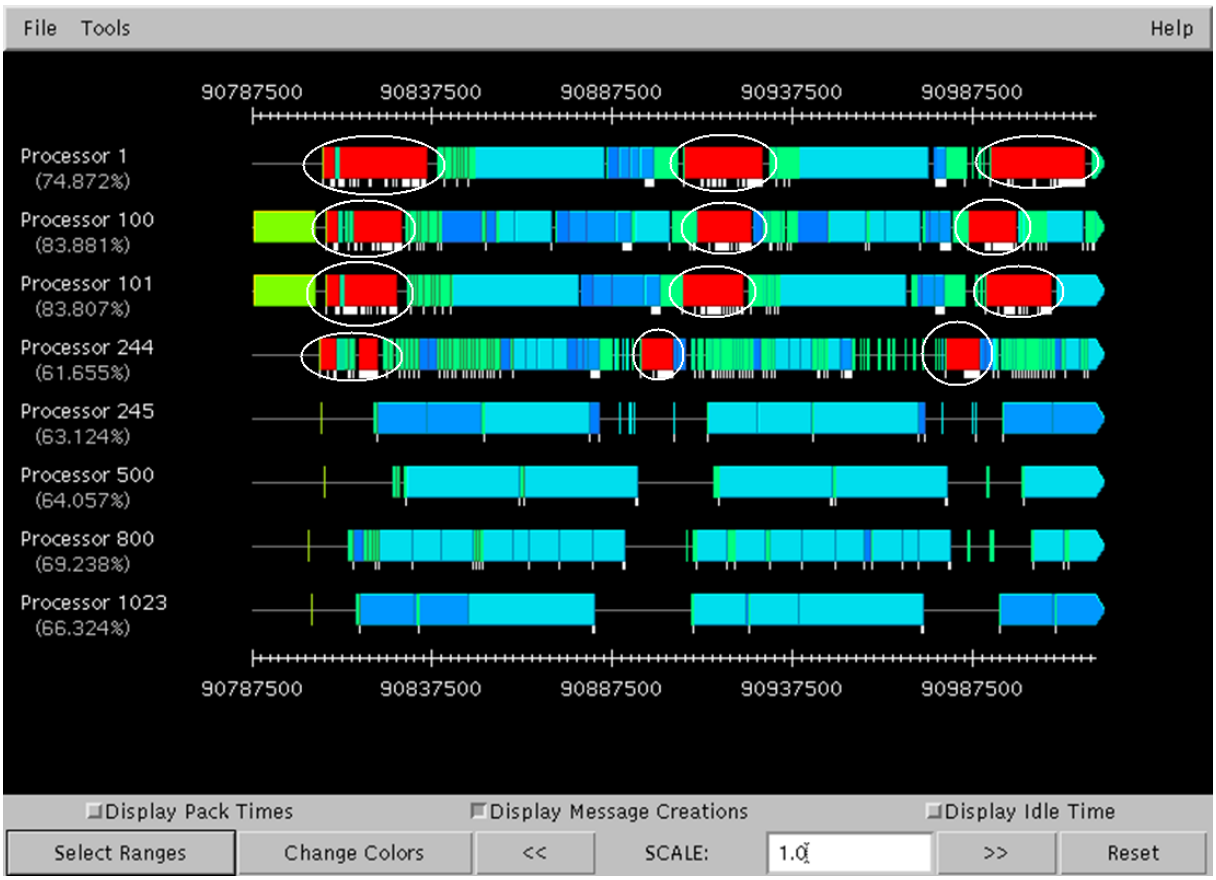


Figure 3.10: A small sample of processor timelines out of 1,024 showing (red) events with inefficient multicasts.

It is important to note that the effort to pick out the appropriate processors over the full set of 1,024 processors was painstaking and involved a fair amount of luck. This process would be even harder, and practically infeasible to depend on, for tens of thousands of processors and more.

3.3.2 Case Study: Finding Causes For Load Imbalance

One common idiom uses the "usage profile" feature to support the discovery of possible load balance problems. Usage profiles (as shown in figure 2.7) show the utilization per processor averaged over a user-specified time-range. The idiom employed to detect potential load imbalance problems requires us to find occurences of one or more processors with significantly higher uti-

Figure 3.11: A small sample of processor timelines out of 1,024 showing much shorter (red) events with improved multicasts and a corresponding improvement in overall performance.

lization (overloaded) than the typical utilization amongst all processors. Processors that exhibit low utilization (underloaded), on the other hand, do not contribute to a load imbalance problem. Unlike overloaded processors, underloaded processors are not execution time bottlenecks that limit the overall application's performance. Once overloaded processors are identified, we have to study the detailed timelines of performance events on those processors in order to confirm if load imbalance was indeed the cause for poor performance. It is important to answer questions about object-placement on the overloaded processors and compare that with object-placement on other processors which communicate with the overloaded processor. For example, the overloaded processor may include a mix of statically-placed objects and migratable objects. If the most overloaded processor contains mostly migratable objects, it suggests a better load balanc-

ing strategy may solve the problem. However, if the most overloaded processor contains mostly statically-placed objects, it suggests that a better mapping for those statically-placed objects is in order.

However, using traditional usage profiles to locate overloaded processors for detailed study is also not scalable. Usage profiles become harder to use when the application is scaled to very large numbers of processors. Figure 3.12 demonstrates the debilitating effect many such usage profiles can have on an analyst when displayed on a screen.

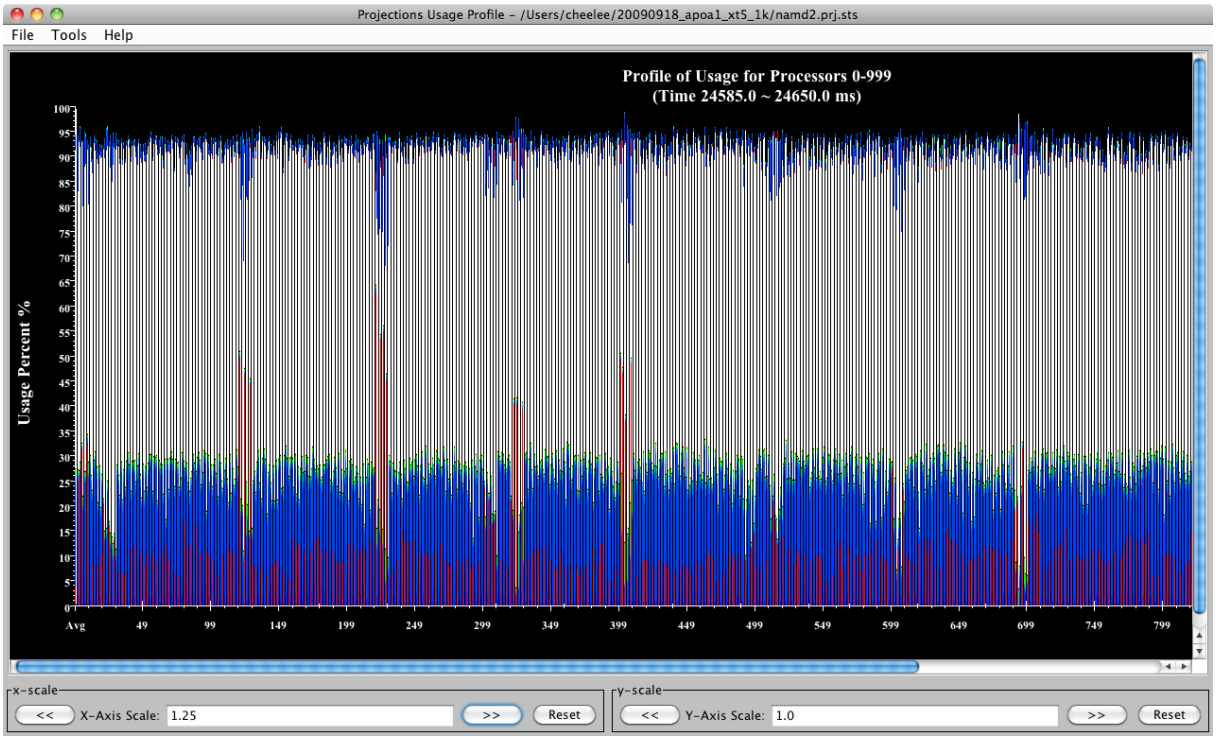


Figure 3.12: An approximately 800-processor slice of Projections processor usage profiles out of 1,024 processors worth of data.

3.3.3 Scalable Tool Support For Finding Interesting Processors

The difficulties associated with picking appropriate processors over a very large set of processors for finding various performance problems led us to develop a more scalable approach. The general idea is to automate the discovery of important performance properties of each proces-

sor's performance structure or profile and allow the analyst to pick the processors with the most extreme performance profiles.

Figure 3.13: An Extrema Tool visualization showing the usage profiles of only the top 20 least-idle processors.

This scalable tool feature, which we call the *Extrema Tool*, takes as input performance criteria to determine processors which best satisfy the criteria. Returning to the case study for the detection of possible load imbalance (see Section 3.3.2), the heuristic generally used is that overloaded processors tended to have the least idle time. Figure 3.13 shows an instance of using the new scalable feature to pick out the usage profiles of the top 20 least idle processors as the performance criteria. These usage profiles are sorted by significance, with the most extreme processor profiles on the right. On the left are six extra columns that include the averaged processor profiles for the set of displayed extremes as well as the averaged processor profiles for the rest of the processors in the dataset. The profiles on the left allow an analyst to quickly determine if the extreme processor elements really do deviate significantly from the rest and

warrant further study. In figure 3.13, the minimal difference between the averaged profiles of the non-outliers and outliers in the left of the image suggests that load imbalance is not likely to contribute significantly to any performance problems in the application being analyzed.

In general, any criteria can be used to pick processors of interest to the analyst. The case study involving poor multicast behavior could find processors that send messages most frequently. k-Means clustering can be used to find processors whose performance behavior are the most unusual within their behavioral equivalence classes.

As noted above, once the interesting processors are identified, the analyst often needs to examine their behavior in greater detail. To enhance the analysis process using the Extrema Tool, a simple mouse-click on any of the usage profiles of extreme processors will prompt Projections to load the timeline for that processor. This feature allows the analyst to incrementally load extreme elements of the processor set for selective study. With the timelines loaded, the analyst may proceed to investigate the performance and find any problems. Typical followup action includes: scanning the timeline of the extreme processors for problems; loading the timelines of processors that sent a message causing a late event to occur on the extreme processor; or loading the timelines of processors that logically belong to the same physical hardware node as the extreme processor.

We now use another case study to exemplify the use of this idiom. A similar study on 16,384 processors was published by Kumar et. al. [50] as part of a broader study of NAMD (see section 4.6.1) performance on the Blue Gene/L. In that study, a time profile (see figure 3.14) was used to reveal possible load imbalance by the presence of deep troughs in between timestep iterations in the simulation executed on 8,192 processors. This was unexpected, as separate performance information from our load balancers had indicated that the balance had been good.

Adhering to our idiom for finding possible load imbalance scalably, the Extrema Tool was used with least idle time as the criterion. This is illustrated by figure 3.15. The averaged non-outlier processors' performance profiles on the left were significantly different from those of the averaged outlier processors. The surprise came in the form of the size of the lime-green bars at

the top of each column. These bars indicated time spent in performance events associated with the intermediate nodes of a spanning tree used in a multicast. The developers had expected the intermediate spanning tree nodes to incur very little overhead.

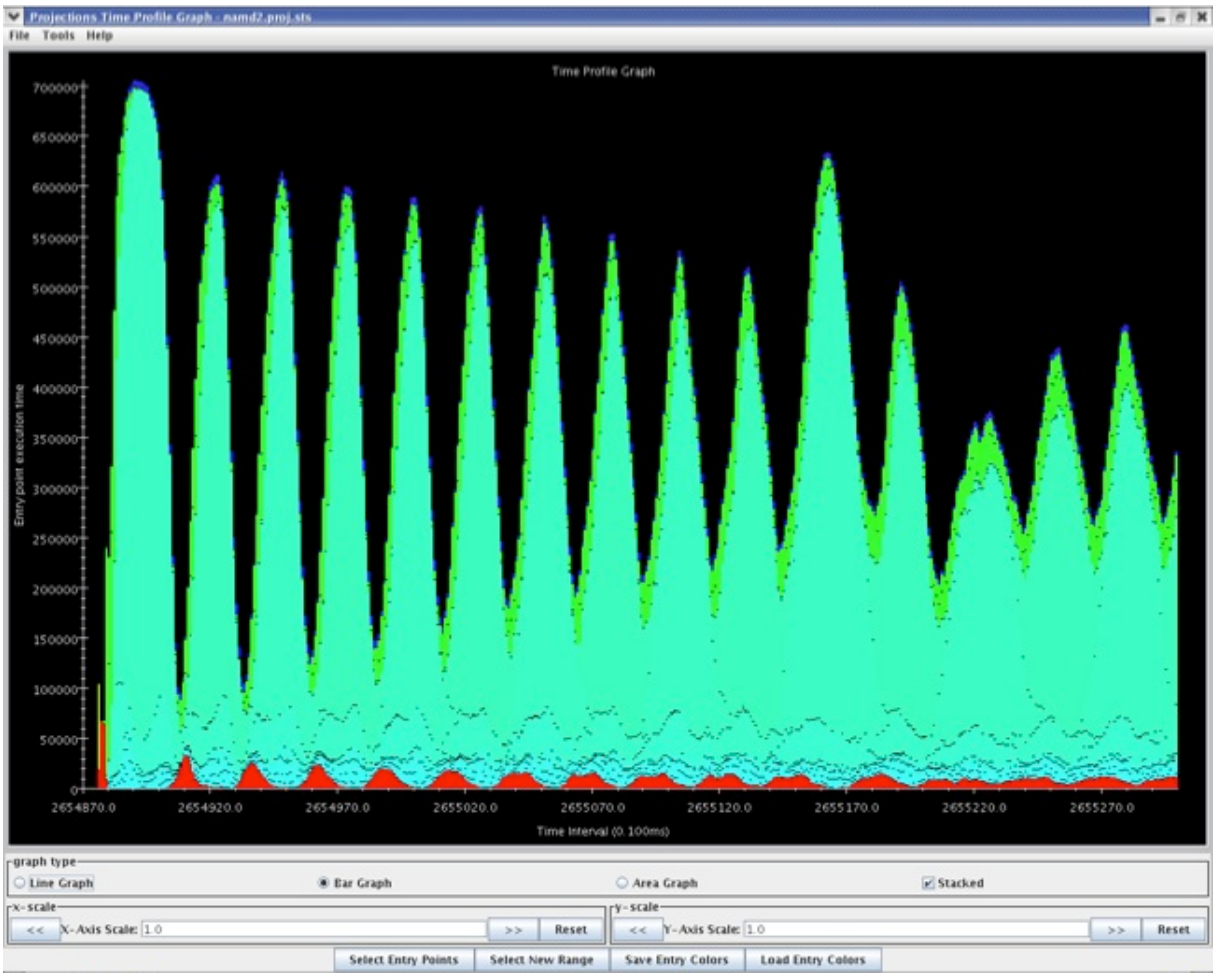


Figure 3.14: Time Profile showing deep troughs between computation iterations, indicating possible load imbalance.

Using the click-and-load feature of the Extrema Tool, we were able to quickly bring up the timelines for the two most interesting processors on the right of the figure, processors 2940 and 980 respectively. By further loading the timelines of processors that had spanning tree nodes communicating with processors 2940 and 980, we were able to construct a picture, illustrated by figure 3.16, showing not only the unexpectedly high cost of intermediate spanning tree work but that multiple intermediate spanning tree nodes were placed on the same processor. Because the

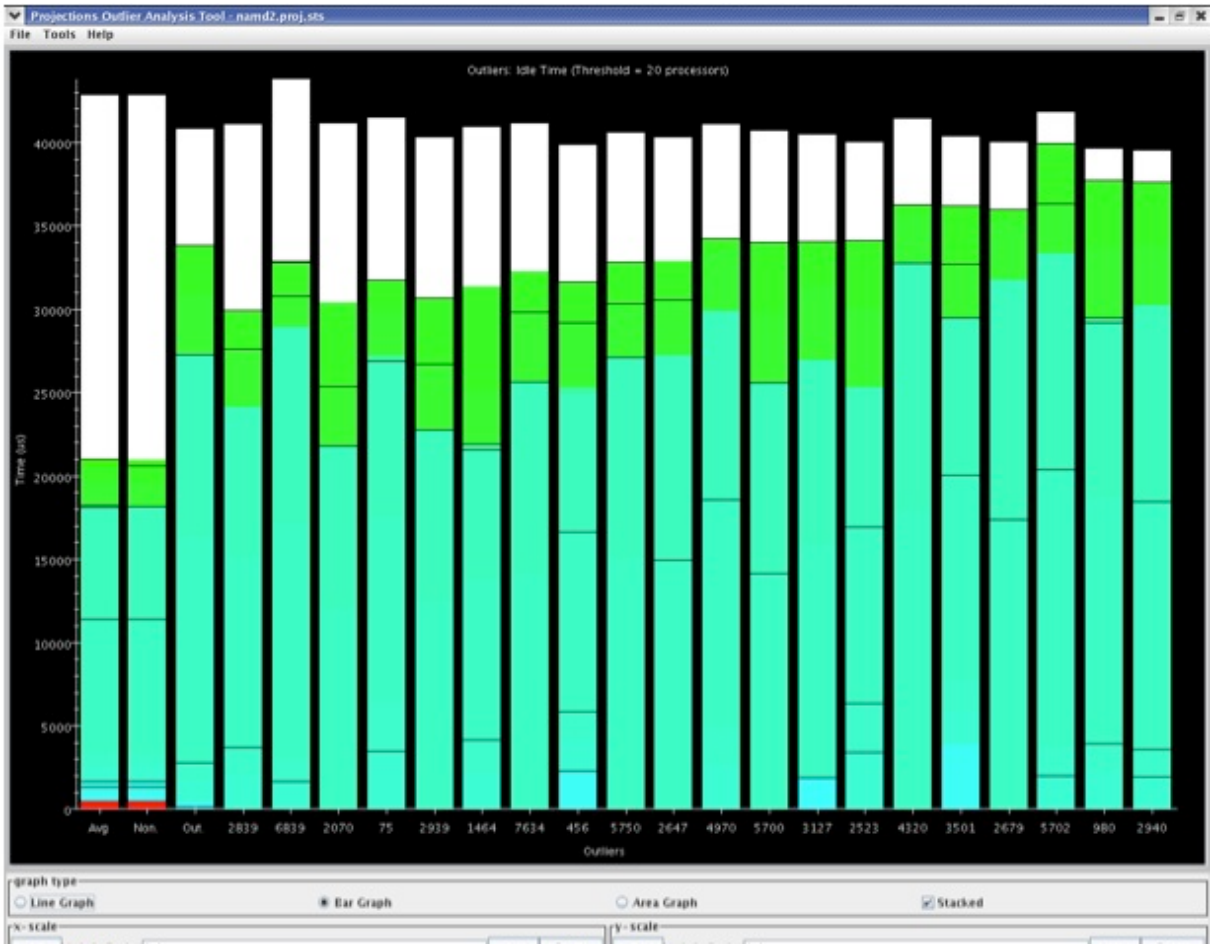


Figure 3.15: The Extrema Tool presenting the top list of least idle processors.

spanning tree can only be computed after load balancing decisions are made, these intermediate nodes introduced the load imbalance after the load balancing strategy believed it had done a good job.

This case study demonstrates just how much more productive the analysis task became with the use of the Extrema Tool for data from so many processors.

3.3.4 Conclusion

The idea behind scalably picking interesting processors (Section 3.3.3) can naturally be applied to the reduction of data volume in performance event traces (see Chapter 4). The supporting tools enabled us to load only the timelines that mattered when finding extreme processor be-

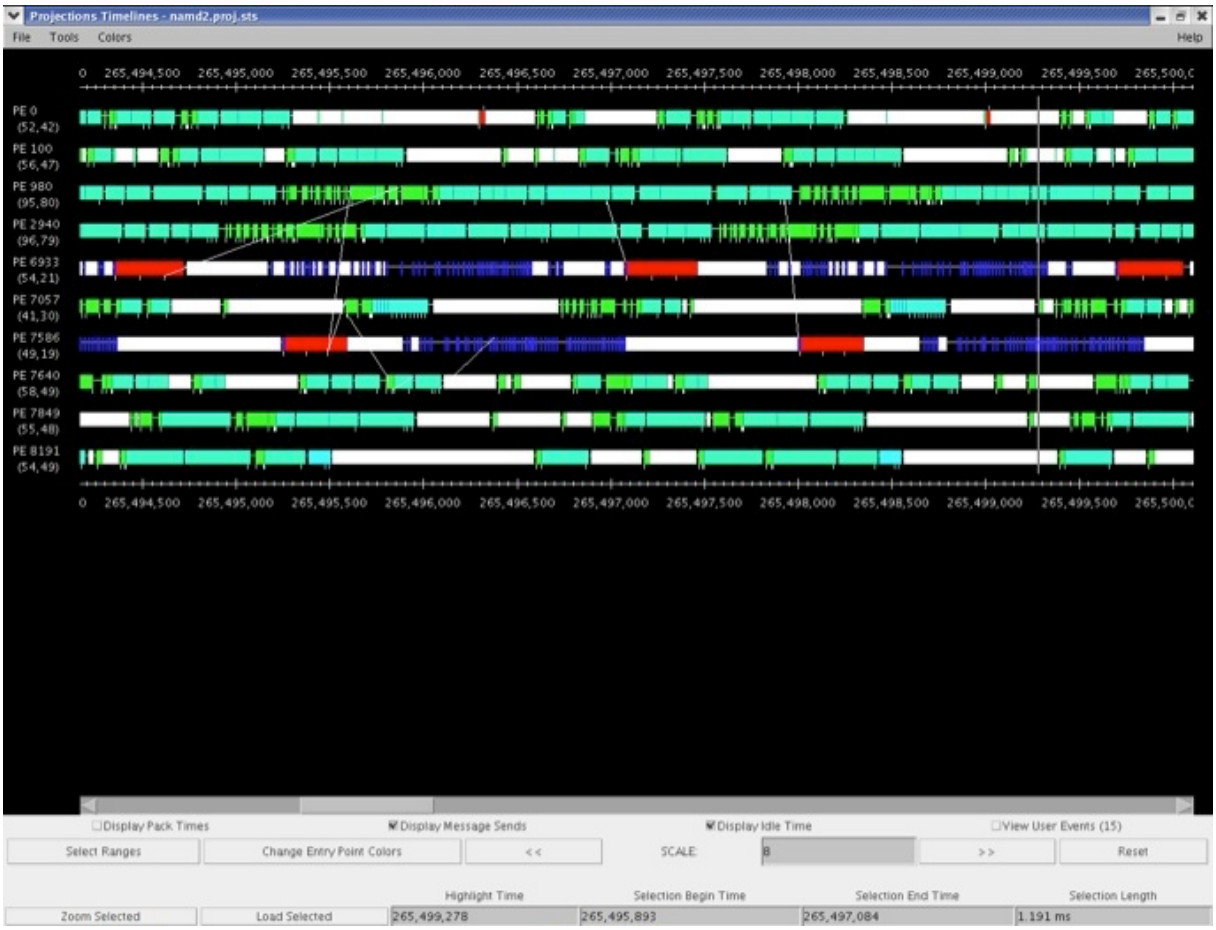


Figure 3.16: Timelines of the top two least idle processors as well as processors that communicate with spanning tree activity.

havior in order to continue the analysis to the root cause of the problem. The majority of the performance data not directly involved with these extreme processors described in the analysis process were never touched for the purposes of detailed timeline analysis. Other idioms that do require the detailed data do well enough with summary profiles supported by Projections. So, for all intents and purposes, the detailed performance data for processors which are neither extreme processors nor party to interactions with extreme processors tended to be useless.

3.4 Related Work

Histogram views are supported by many tools such as Paradyn [52] and Jumpshot [85]. However, the literature appear to always describe histogram views "as-is". Their value as a class of supporting tools for scalable performance analysis idioms have not been identified.

The use of high resolution time profiles as a supporting tool for scalable performance analysis idioms has not been widely adopted in the performance tool community as far as we know. The TAU group had developed a snapshot capability [55] for their profile format. Snapshots are essentially performance profiles captured for each user-defined application phase. The snapshots are written to individual files and then read by their visualization and analysis tool. Their tool presents a view similar to our time profiles but at per-application-phase resolutions. Jumpshot-3 [2] supported a tool named "Statistical Preview" which provided a similar view of stacked event metrics but was intended as a preview for the entire instrumented portion of the run divided into 512 coarse time intervals along the x-axis. The tool does not appear to be supported in Jumpshot-4. Paragraph [33] supported a communication traffic view over time that supports message volume and count, much like the communication-over-time view we describe in section 3.2.3. Cray Apprentice[2] [20] supported a similar tool to show how hardware counter values change over time. Each hardware counter metric had its graph displayed on its own horizontal bar rather than stacked in a single chart to reveal overlapping structural changes like in our time profiles.

Tools that pick unusual processors for analysts to conduct more detailed analyses in the manner of our Extrema Tool (see section 3.3) are useful for many analysis idioms. The most similar approaches rely on automatic analysis and tuning tools to find or fix any specific performance problems they discover. In the case of automated analysis tools, the specific details of the problems found are reported to the analyst, often bypassing the need for any visualization support. These systems typically rely on a pre-determined set of common performance prob-

[2]http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/jumpshot-3/index.html

lems. We believe this to be inadequate in practice. Like debugging, performance analysis must be flexible enough to deal with the unexpected. Often, we stumble upon different problems while exercising an idiom to look for, and find a solution to an expected problem.

Sinha and Kalé [71] developed an early automated analysis system for CHARM++ applications called "Projections:Expert" to identify potential performance problems. The results were displayed as list of detailed descriptions of performance problems sorted by some severity measure. Chung and Hollingsworth [15] use Active Harmony to automatically tune large-scale scientific applications by probing the performance search-space in the form of different tunable parameters in an iterative fashion. New parameter settings are selected according to the performance observed in previous iterations. Such a technique requires knowledge, by the programmer, of the parameters in the application or its libraries that are more critical for performance, and of a practical range of values that those parameters can assume. This typically demands intensive knowledge of the entire application. Geimer et. al. [28] described a way, as part of their KOJAK framework, to apply expert analysis on distributed event trace logs by employing parallel replay of messages. This was designed as a scalable alternative to the sequential automatic analysis tool called "EXPERT" they had previously been using. With it, they hoped to find inefficient message-passing patterns scalably. The patterns are presented by a tool named "CUBE" as a performance metric tree and a call-tree with severity measures. The nodes of the trees representing the worst problems are highlighted.

Chapter 4

Data Volume Reduction

In chapter 3, we described the development of tool features to support scalable performance analysis idioms that seek interesting processors for more detailed study. These features have been employed on the full event traces of all the processors generated from an application. There will be times the volume of performance data generated by an application will be so large, it becomes cumbersome or infeasible to transfer and load onto performance tools for analysis.

In this chapter, we investigate a technique to allow performance tools to more effectively handle the performance data generated by parallel applications at very large scales. We had previously alluded to the fact that the principles behind the Extrema Tool (See section 3.3.3) could be extended for the purposes of performance event trace data reduction. The data reduction achieved is lossy, the volume of data is reduced by keeping only the data relevant for the purpose of finding detailed performance problems [1]. The goal is to enable the same scalable tool features to be used effectively for performance anlaysis, but on the reduced dataset.

4.1 Basic Approach

The basic idea behind this approach is the retention of detailed performance event traces for only a representative subset of processors on which the application is executed. These representatives

[1]This chapter is written with portions reprinted, with permission, from "Towards Scalable Performance Analysis and Visualization through Data Reduction" by Chee Wai Lee, Celso Mendes and Laxmikant V. Kalé at the *13^{th} International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008*, ©2008 IEEE

are retained because of their potential value in revealing performance problems when the data is analyzed. As described in section 3.3.4, the approach is based on the idea that analysis idioms pick out only processors that mattered for detailed analysis. Since we do not know a priori the nature of the performance problems we expect to encounter, the most general approach we can take is to attempt to pick out the processors exhibiting the most unusual performance behavior. We observe that in a large class of parallel applications, sets of processors often exhibit similar patterns of performance behavior, forming equivalence classes. For example, in NAMD (see section 4.6.1), there is the concept of a set of "patch" processors that perform integration work exclusive to themselves for each of the cubes of simulated 3-dimensional space. As such, our approach depends on some ability to identify these equivalence classes. To identify equivalence classes of processor behavior, we make use of simple implementations of the well-known k-Means clustering algorithm [37]. Using the output from k-Means clustering, we can employ simple heuristics to identify processors whose behavior are representative of the run. These *representative* processors have two components:

1. *exemplar processors* whose behavior are closest to mean processor behavior of each equivalence class; and

2. *outlier processors*, whose behavior deviate the most from mean processor behavior in each equivalence class.

For the processors whose detailed event traces we choose not to keep, we will still retain their Projections summary profiles (see section 2.2.2). The profiles are very small and do not significantly impact the overall size of the performance dataset. These profiles are rich in information in their own right and are highly useful for many of the performance analysis idioms. As such, profiles can serve as a reasonable source of contextual performance information when used in conjunction with a reduced set of event traces.

For the rest of this chapter, we will discuss related literature before proceeding to describe our approach in detail. We will also explore various extensions to the basic idea.

4.2 Related Work

The use of clustering and other statistical methods for the general goal of performance visualization and analysis is not new. Pablo [59, 63] demonstrated an early prototype the most similar to our basic idea for data reduction. They retain trace data segments for processor representatives of clusters that are discovered. They re-cluster by some chosen frequency to account for performance data changes over time. There are a number of main differences between our approaches. First, by choosing only representatives, which we refer to in this thesis as "exemplar" (see Section 4.4.3), they are able to claim a high degree of data reduction simply from the fact that the number of equivalence classes tend to be small compared to the number of processors executed, especially at large scale. We believe their goal may be to simply discover a problem with the application's performance, for general inefficiency features will show up with their approach. For our purposes, however, we seek the detailed source of the performance problem. For that, performance data from "outlier" processors of the equivalence classes we discover are our focus. They are likely to contain the details of performance problems, while "exemplar" processors preserve the general performance context. This idea is, to an extent, reinforced by the work of Benkert et. al. [4] which demonstrated the use of hierarchical clustering to identify outliers. Second, they consider the application of clustering at some arbitrary points in the execution by specifying a clustering frequency. This required them to consider a good choice of clustering frequency and taking into consideration the costs of clustering. We know, however, within our application context when different phases occur. Performance properties necessarily change from one computation phase to the next. We also apply clustering at the end of the application's execution, either after the data is written to disk or while the complete data still resides in memory.

Meanwhile, Ahn and Vetter [3] have shown the value of multivariate statistical methods to gather performance insight about very large performance datasets. They use a variety of clustering techniques along with factor analysis to identify performance factors or metrics (their

concern was with hardware performance counters) that vary the most across processors and why they form different clusters. The results of their research was only intended to be the first step for further performance analysis. Aguilera et. al. [2] followed up on Ahn and Vetter's research by using hierarchical clustering on communication data to identify clusters of communicating processor pairs in written event trace logs. The use of clustering algorithms for performance characteristics to be visualized using a scatterplot matrix has been explored in TAU [69]. The developers of TAU do not make use of clustering for data reduction but have highlighted the importance of dimensionality reduction and the removal of correlated performance metrics.

Gamblin et. al. [25] have described the use of clustering or stratification to enhance a statistical sampling approach used to reduce the volume of performance trace data. They showed how stratification allowed fewer samples to be taken from cluster equivalence classes with lower variance. As a result, they were able to show improvements to the data reduction using stratification over the basic approach. The main difference between our approaches is the use of random statistical sampling to preserve the capture of overall application behavior as with Nikolayev et. al., while we adopt a bias for extreme processor behavior to attempt to pick out processors most likely to demonstrate performance problems.

Other research on data reduction tends to focus on the temporal dimension, reducing the length of event trace logs rather than the width of the trace data. The research in this area is highly related to the automatic pattern search technique described earlier in Wolf's survey paper [80]. Mohror and Karavanic [53] evaluated a number of different classes of similarity metrics for data reduction, including the Euclidean distance measure used in this thesis. The data reduction approach adopted involved the partitioning of the application code into segment contexts. As the application executes, segment instances are compared using an equivalence operation based on the chosen similarity metric and some threshold. Only segments dissimilar to previously retained segments are retained. Full traces can then be reconstructed for analysis purposes. Supinski et. al. [17] employs regular section descriptors as data structures to capture and compress on-the-fly the performance characteristics for MPI events in loops on a single

node. Performance characteristics are either captured via statistical aggregation or size-limited histograms of the delta of MPI event timestamps. Performance studies are made by replaying the compressed trace using a replay engine to re-create the communication structure of the application. Inter-node structures are merged at the end of the application execution, further compressing the data. Freitag et. al. [24] described the use of a periodicity detector for function streams in OpenMP applications. They toggle performance instrumentation of the OpenMP application to avoid tracing data if an identical periodic pattern is encountered. Chung et. al. [16] sought repeated communication patterns in MPI codes as a source for compression, augmenting the visualization tool to re-generate the full details when needed. Knupfer and Nagel [48] demonstrated potential for temporal data reduction through the construction and subsequent compression of complete call graphs. They further introduce a distributed architecture for performance analysis [46] that enhances scalability by allowing parallel analysis. Szebenyi et. al. [73] extended the approach taken by Knupfer and Nagel by considering a time-series of call-path profiles representing phases in an application instead of the complete call graph. They then applied incremental clustering to compress the time-series of profiles. Casas et. al. [12] meanwhile applied signal processing techniques like non-linear filtering and spectral analysis directly on event traces in order to identify similar regions along the time dimension and achieve data reduction by removing multiple instances. Vetter and Reed [76] studied the reduction of performance data by removing uninteresting performance metrics through the technique of dynamic statistical projection pursuit.

4.3 Quantifying The Problem

In this section, as a motivating case study, we quantify the data volume of various large event traces generated by applications instrumented by Projections. We also report the time required to transfer the data from the supercomputing facility to local performance analysis resources. The intention is to provide a general picture of the challenges faced by analysts of performance

data at large scales. The data was generated over two sessions, the first on bigben, a Cray XT3 operated by Pittsburgh Supercomputing Center (PSC) and the second on kraken, a Cray XT5 operated by the National Institute of Computational Sciences (NICS) and housed at the Oak Ridge National Laboratory.

We first describe a simplified model for the components of event trace data volume when an application is scaled. For a non-speculative scientific code, we may consider the total performance data volume to be of the order of $O(W(P) + C(P))$ where P is the number of processors, $W(P)$ is the total number of computational events generated by an application due to the required work as well as any variable algorithmic component influenced by P or input configurations, and $C(P)$ is the total number of communication events generated by the application given P.

In table 4.1, we quantify the volume of trace data generated for three different NAMD simulations (see Section 4.6.1 for details) generated at PSC's XT3 from 512 to 4,096 processors. **ApoA1** [61] is a simulation of the Apolipoprotein A-1 protein with 92,224 atoms. **F1ATPase** [21] simulates the F1 part of the protein ATP synthase with 327,506 atoms. Finally, **stmv** [23] simulates the Satellite Tobacco Mosaic Virus using 1,066,628 atoms. The data was generated using the default NAMD configurations of the respective simulations which limited event trace generation to 200 time steps of the simulation. The diagonals of the table shows how data volume tends to grow in the case of weak scaling, where a larger simulation is used as we increase the processor count. Data volume growth in the case of strong scaling can be followed by reading down each column of the table.

Table 4.2 looks at different configurations of the **stmv** simulation. NAMD can be configured to run with different particle mesh ewald (PME) computation settings. PME is responsible for computing forces due to long-range electrostatics. PME computations may be invoked once every 4 NAMD time steps, which is the default configuration. It may also be invoked every time step, in which case some additional computation and housekeeping work is avoided. The purpose of this table is to highlight the magnitude and variability of event volume due to

nCPUs	ApoA1	F1-ATPase	stmv
512	827 MB	1,800 MB	2,800 MB
1024	938 MB	2,200 MB	3,900 MB
2048	1,200 MB	2,800 MB	4,800 MB
4096			5,700 MB

Table 4.1: Total volume of trace data summed across all files. **ApoA1** is a NAMD simulation with 92k atoms, **F1-ATPase** simulates 327k atoms while **stmv** simulates 1M atoms. This data was generated on the XT3 at PSC. *Reprinted, with permission, from "Towards Scalable Performance Analysis and Visualization through Data Reduction" by Chee Wai Lee, Celso Mendes and Laxmikant V. Kalé at the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008, ©2008 IEEE.*

configuration or algorithmic changes. The table is based on a later dataset acquired on the Cray XT5 at NICS.

From the data, we can approximately isolate the effects of work due to PME on overall data volume. The difference between the corresponding values for no PME work and for PME work every four steps gives us an estimate of the additional contribution to total data volume by a single PME operation per time step for a given processor count.

We can also see that the data volume increased significantly from 4,096 to 8,192 processors. This is because of a recent automatic heuristic in the NAMD code which attempts to decrease the grainsize of the work by splitting the size of a patch in one dimension. This decision is made at some threshold which is determined in part by the number of processors involved and the number of atoms being simulated. This captures event volume growth due to an application's algorithmic changes. Algorithmic changes do not come about directly as a consequence of strong scaling but are often required in order to maintain good performance in the face of strong scaling.

Finally, we felt it useful to document the wide-area transfer bandwidth for the transfer of our performance event trace logs. When we transfered our performance data from NICS to the University of Illinois where our own servers were held, the typical observed bandwidth was between 300 to 600 kilobytes per second. Our data totalled 42 gigabytes after compression

nCPUs	No PME Work	PME every step	PME every 4 steps
512	2,321 MB	4,347 MB	2,789 MB
1024	2,619 MB	7,999 MB	3,926 MB
2048	3,255 MB	10,024 MB	4,886 MB
4096	4,895 MB	12,189 MB	6,511 MB
8192	13,007 MB	21,813 MB	15,898 MB

Table 4.2: Total volume of trace data summed across all processors in **stmv** Projections logs with different Particle Mesh Ewald (PME) long-range electorstatics configurations. The data was generated on an XT5 at NICS.

which resulted in transfer times in the order of hours. It is therefore clear that data volume reduction will be helpful in reducing the time required for transfer if the reduction can be done without affecting the effectiveness of performance analysis using local resources.

4.4 Applying k-Means Clustering To Performance Data

We now consider how to effectively apply the k-Means clustering algorithm to the domain of parallel performance metrics. We begin by describing the general k-Means algorithm along with the form performance metric data must take in order to serve as appropriate input to the algorithm.

4.4.1 The k-Means Clustering Algorithm

The k-Means clustering algorithm is used in general to partition a set of objects by minimizing some chosen measure of dissimilarity across the set of objects. A detailed formal definition of the k-clustering problem can be found in Inaba et. al. [36]. For our purposes, the objects in question are the behavior profiles of individual processors. A processor's behavior profile can be defined as an m-tuple of performance metrics that characterize the processor's performance behavior. These tuples will serve as input to the clustering algorithm in our context. Because performance metrics are recorded in the form of simple numerical values, we can choose to employ the Euclidean Distance between two points in m-dimensional space as our measure of

similarity subject to some normalization considerations, as we will see in section 4.4.2.

The basic k-Means algorithm pseudocode is shown in figure 4.1. Using terminology described in the preceeding paragraph, K is the input number of clusters to be found while x_1 to x_N are the vectors of performance metrics (of length m) for each of the N processors.

```
kMeans({x_1, ... ,x_N},K) {
  (seed_1, ... ,seed_K) =
    selectRandomSeeds({x_1, ..., x_N}, K);
  for k = 1 to K {
    centroid_k = seed_k;
  }
  while stoppingCriterionIsNotMet() {
    for k = 1 to K {
      cluster_k = emptySet();
    }
    for n = 1 to N {
      j = closestKByEuclideanDistance({centroid_1, ..., centroid_k},
      x_n);
      cluster_j = setUnion(cluster_j, {x_n});
    }
    for k = 1 to K {
      centroid_k = computeCentroid(cluster_k);
    }
  }
  return {cluster_1, ... , cluster_k};
}
```

Figure 4.1: Basic sequential k-Means Pseudocode.

Figure 4.2 shows a step-by-step example of the final result from the hypothetical application of k-Means clustering with 2 clusters, over 2 arbitrary and unlabelled performance metrics. X and Y are initial cluster seeds randomly chosen from the 6 data points. At each step, the labels on the data points indicate which cluster seed it is closest to. At the end of each step, each cluster seed moves to a location which is the averaged distance from each data point closest to it. As we can see from the second step in figure 4.2, this can make a seed move far away enough from prior closest data points for another seed to become considered closer. In this case, Y loses a data point to X. As a result of the change, another step is required and both X and Y

move to their new positions based on the averaged distance from their new set of closest points. The algorithm stops when there are no more membership changes.

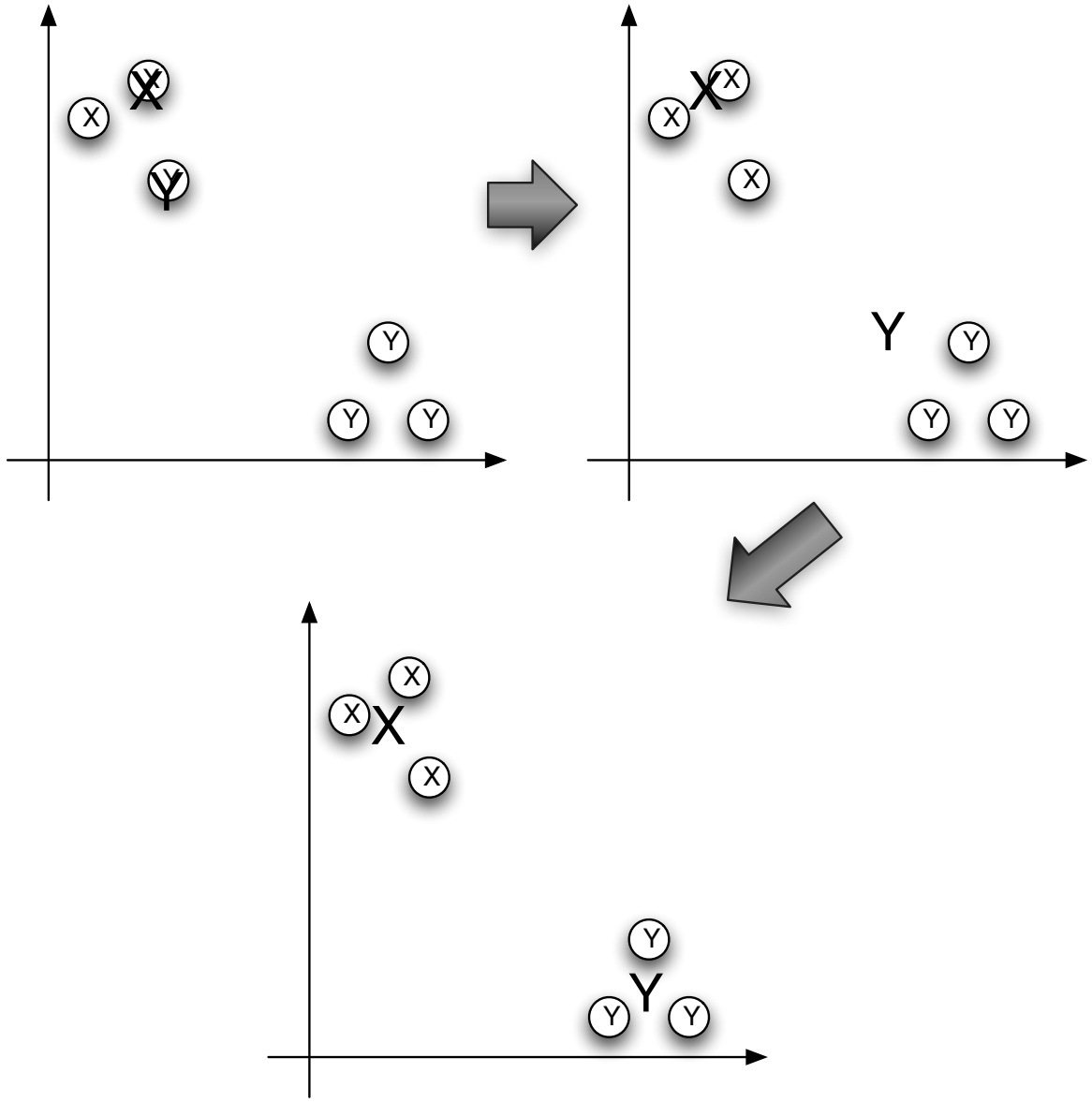


Figure 4.2: An example showing a 3-step convergence with X and Y as the initial cluster seeds and eventually becoming the cluster centroids.

4.4.2 Important Choices For k-Means Clustering

Effective use of the k-Means clustering algorithm require us to consider the challenges posed by variety of the algorithm's parameter choices. It is important at this point to note that the thesis does not try to explore the optimality of data clustering to our problem domain for which the k-Means algorithm is but one of many. Instead, we seek to demonstrate the data reduction gains provided by a rudimentary but reasonable data clustering technique.

Choice and Initialization of k

The choice of the numeric value of k can affect the quality of the equivalence classes discovered through k-Means clustering. In general, the user-specified value of k tells the algorithm to locate k clusters in the metric space. This may not correspond to a more natural number of equivalence classes. Most tools making use of k-Means clustering either make use of domain knowledge or trial and error for the choice of k. The key challenges in our context where the choice of k is concerned are:

1. that k must either be chosen by the analyst before the application is executed or be determined by some automated trial-and-error mechanism.

2. there are no guarantees that all k cluster centroids will be non-empty at the end of running the k-Means algorithm.

The choice of initial placement of the k seeds can also affect the eventual membership in each non-empty cluster. Traditional k-Means clustering selects k random object data points as the initial seeding. Another option is to place the initial k cluster seeds uniformly by spreading them along the m-dimensional diagonal of the data's bounding-box. The bounding-box is formed by the minimum and maximum values of each of the m performance metrics over all P sample points where P is the number of processors. Finally, we can choose k random points within the data's bounding-box.

Similarity Measures and Normalization

In general, k-Means clustering requires some normalization of values across the m performance metrics to avoid bias. Failure to normalize values in the general case would mean that metrics with large absolute values but relatively small variation would make a disproportionately large contribution to the Euclidean distance measure compared to metrics with small absolute values but relatively large variation.

Three methods for data normalization are described by Visalakshi et. al. [77] for the application of the k-Means algorithm to different domains. The *Min-Max normalization* method maps a value v_m of the metric m to a real value v_{m_norm} from 0 to 1 as follows, where min_m is the minimum value for metric m and max_m is the maximum: $v_{m\_norm} = \frac{v_m - min_m}{(max_m - min_m)}$. Meanwhile, the *Z-score* method normalizes values based on the mean and standard deviation of values instead of the min and max. Finally, the *Decimal scaling* method simply normalizes values by moving decimal points.

In our domain of performance metrics, the situation is somewhat more complex. Let us consider metrics that capture entry method execution time. The values of individual metrics within this class should *not* be normalized. We want to retain the inherent bias for entry methods whose total execution time were high compared to entry methods whose total execution time was small. This is because the former entry methods are considered to have a far greater impact on the performance of the application as a whole. However, this does not mean we *do not* normalize performance metrics at all when using k-Means clustering. For example, now consider, alongside entry method execution time, the class of performance metrics that capture the number of bytes sent in a message. Again, within this new class of metrics, normalization should not be applied. However, because the two classes of metrics are not correlated to one another, normalization needs to be applied to remove the bias against the classes of metrics with smaller values.

In the context of this thesis, we restrict the set of performance metrics to the execution times

of a subset of the instrumented CHARM++ entry methods. This choice avoids the need for additional normalization effort to eliminate distance measure bias.

Useless Metrics

The efficiency of the clustering algorithm also depends on the number of metric dimensions it has to deal with. Many metrics in performance data under our context tend to have zero values, mostly because they were registered to the CHARM++ runtime but never used. These metrics will need to be removed.

At the same time, a number of metrics may be highly correlated. That is, whenever a large value is observed for metric X, a correspondingly large value is also observed for metric Y. These values are then used to compute the measure of similarity. Because the correlation causes a double-counting of the effect on similarity measures between two data points, it is best to eliminate either metric X or Y for more accurate clustering results.

4.4.3 Choosing Representative Processors

The output from the k-Means algorithm is a processor-membership set and a centroid for each of the k clusters. Some of these clusters can be empty. In addition, we know the similarity measures for each data point from the centroid of the cluster to which they belong.

The goal now is to pick out processors whose behavior best represents this collection of data points given this clustering output. The heuristics we will employ expects the analyst to supply some value for a reduction factor to the performance dataset. For the purposes of our discussion, let us refer to the value R as the number of processors' performance data to be chosen as representatives of the original data.

We now have the task of selecting two classes of representative processors. The first are what we call *exemplar* processors, whose behavior are the closest matches to the cluster centroids to which they belong. The second are *outlier* processors, whose behavior are furthest removed

from the cluster centroids to which they belong.

Choosing Exemplar Processors

Our choice of exemplar processors given a set of equivalence classes is a simple one. For each non-empty cluster C_i, we select one representative closest to the cluster centroid according to the Euclidean distance measure described in section 4.4 to be our exemplar.

This selection policy assumes a single processor's data point will sufficiently represent other processors in the same equivalence class. Depending on the nature of the clusters discovered, this is not necessarily true. A more refined method could take advantage of the statistical distribution of data points associated with the cluster to determine a small set of exemplar processors whose mean similarity measure is closest to the cluster centroid. However, we leave this as future work as exemplars are intended to provide the performance context for the performance analysis process rather than to capture performance anomalies in the execution.

Choosing Outlier Processors

In the preceeding section, we chose C data points as exemplar processors providing the performance context on which we may analyze the reduced dataset, where C is the number of non-empty clusters found.

We now describe the heuristic for selecting the remaining $R - C$ data points as outliers whose behavior is most different from the cluster centroids and where we expect to find performance problems. We apply a proportional selection policy based on the size of each non-empty cluster to choose approximately $|C_i| \div P$ outliers from each non-empty cluster C_i. By our heuristic, the outliers are simply the data points with the $(|C_i| \div P)^{th}$ largest dis-similarity measures from the cluster centroid. If a cluster has one or fewer data points, then nothing needs to be done, as the representative selection scheme has already picked out that processor as an exemplar.

This heuristic is sensitive to the quality of the equivalence classes generated by the clustering algorithm in 4.4. It does not take into account the way data points are distributed within each cluster. For instance, if one cluster is particularly tight relative to others, then perhaps the heuristic can be made to realize that this cluster is best captured by just selecting the exemplar. We leave this feature to be explored in future work.

4.5 Post-mortem Versus Online Data Reduction

Data reduction can be applied at two different points in time. The first and simplest is to use the k-Means clustering algorithm post-mortem on data already written to disk. The advantage of post-mortem data reduction is the ability of analysts to choose different k values or seed initialization policies to refine the results. The disadvantage of the post-mortem approach is the need to write out all the performance data to disk first.

We take the approach of online data reduction. Online data reduction takes place at the end of the application's execution but before performance data is written to disk. This approach takes advantage of the available memory on the parallel machine to hold all recorded performance data for use by the k-Means algorithm. After representative processors are chosen, only the pertinent detailed event logs get written to disk, the remaining data is discarded. A basic implementation of online data reduction requires k and other important clustering choices (see Section 4.4.2) to be determined before the execution of the application itself. With the capability made available through online interaction using CCS (see Section 5.2), we can allow these decisions to be made after the application has completed but before any data reduction. The idea is to let the analyst invoke clustering over different values of k and quickly observing the results. If a value of k generates too few or too many non-empty clusters for an application, the analyst can try a different value. The reduced dataset is then written to disk when the analyst is satisfied. It should be noted that interaction for this purpose should be used infrequently as this takes place in the middle of a job allocation. If this allocation was a large time-block

reservation, the use of interaction should be reasonable, perhaps even for the analyst to conduct a full performance analysis session on the in-memory performance data remotely. However, when employing a normal scheduler queue allocation, an analyst would have to over-allocate to account for this extra time. In terms of computation units consumed, the interaction could be extremely costly, particularly at large processor scales.

A naive approach to using k-Means clustering online in the parallel machine is to have each processor communicate to a root processor its position-vector in the M-dimensional space used for clustering, where M is the number of metrics observed. The sequential algorithm can then be executed on the root processor to discover the clusters. Unfortunately, this consumes $\Theta(P \times M)$ memory on that root processor and makes this approach inherently unscalable where memory is concerned. A good parallel k-Means clustering implementation with scalable memory usage requirements is the far better solution. Enabling parallel k-Means clustering with scalable memory usage depends on the fact that the position-vectors represented by each processors' metric data point obey the commutative and associative rules for addition and subtraction.

```
Root Processor (Initialization)

1. O(M): Receive results of reduction Metrics. For each metric m,

   m.min <- Metrics.m.min;
   m.max <- Metrics.m.max;
   m.mean <- Metrics.m.sum/numProcessors;

2. Eliminate Useless Metrics.

3. For each metric m, computeNormalizationFactors(m.min, m.max, m.mean).

4. O(K x M): Compute initial k centroid position-vectors.

5. O(K x M): Broadcast k centroid position-vectors and normalization
factors for each metric m.
```

Figure 4.3: Parallel k-Means Initialization Algorithm on the root processor.

```
All Processors (initialization).

1. O(M): localMetricVector <- compute from Projections event buffers.

2. O(M): Contribute via special reduction (min, max and summation) to
root processor the values of each metric in the metric vector.

3. O(K x M): Receive initial k centroid position-vectors (Centroids)
and normalizationFactor for each metric m.

4. myVector <- Normalize(localMetricVector, normalizationFactor);

5. For each k,

   kDist.distance <- ComputeDistance(myVector,
                                     Centroids.k.position-vector);

6. currentCluster <- outOfBandValue;

7. newCluster <- KWithMinDist(kDist);

8. For each k,

   ClusterModification.k <- NullVector;

9. ClusterModification.currentCluster <- myVector;

10. O(K x M): Contribute via summation reduction to root,
              ClusterModification.
```

Figure 4.4: Parallel k-Means Initialization Algorithm on all processors hosting performance data.

Let $\vec{K_i}$ be the position-vector of cluster seed i and p_i be the number of processors closest to cluster seed i. Finally, let $\vec{P_j}$ where $j = 1..p_i$ represent the position-vectors of the p_i closest processors.

Then,

$$\vec{k_i} = \frac{1}{p_i}(\sum_{j=1}^{p_i} \vec{P_j}) \tag{4.1}$$

```
Root Processor (Clustering Iteration)

1. O(K x M): Receive results of reduction ClusterModification. For
each cluster centroid k,

     k.positionVector += ClusterModification.k.positionVector;

2. If any k.positionVector changes,

             break;

   else

   O(K x M): Broadcast updated position-vectors for each cluster
             centroid k and loop to 1.
```

Figure 4.5: Parallel k-Means Iterative Algorithm on the root processor.

Now, we can assign a processor to each cluster seed. This uses $\Theta(M)$ memory for the seed's position-vector. At any particular iteration of the k-Means algorithm, if a processor decides it is closer to a different cluster seed, it will send its position-vector to the processor assigned to the original cluster seed as well as the processor assigned to the new cluster seed. The position-vector is added to the position-vector of the new cluster seed and subtracted from the position-vector of the original cluster seed.

At the end of an iteration, a reduction to the root processor accumulates the number of updates to each of the k cluster seeds. This operation requires $\Theta(k)$ memory. This information allows the root processor to perform two tasks:

1. deciding if the algorithm converges by detecting that no processor updated any cluster seeds; and

2. sending a message to each of the processors whose data points are associated with the k updated cluster seeds telling the processor how many updates to expect and hence synchronize accordingly.

```
All Processors (Cluster Iterations).

1. O(K x M): Receive k new centroid position-vectors (Centroids).

2. For each k,

   kDist.distance <- ComputeDistance(myVector,
                                     Centroids.k.position-vector);

3. currentCluster <- newCluster;

4. newCluster <- KWithMinDist(kDist);

5. For each k,

   ClusterModification.k <- NullVector;

6. if currentCluster != newCluster,

   ClusterModification.currentCluster <- Negative(myVector);
   ClusterModification.newCluster <- myVector;

7. O(K x M): Contribute via summation reduction to root,
             ClusterModification.
```

Figure 4.6: Parallel k-Means Iterative Algorithm on all processors hosting performance data.

Figures 4.3 and 4.4 describe the parallel algorithmic steps, for the root processor and data-hosting processors respectively, required to setup the initial conditions for the iterative portion of the k-Means computation. Bounds on the memory requirements for each parallel reduction or broadcast operation are provided at the start of each appropriate step. Where memory bounds are concerned, K is the number of cluster centroids chosen by the analyst while M is the number of useful performance metrics that are used to determine the position-vector data points for clustering.

Figures 4.5 and 4.6 describe the parallel algorithmic steps, for the root processor and data-hosting procesors respectively, required for each iteration of the k-Means computation until convergence conditions are met. Again, bounds on the memory requirements are presented.

Figure 4.7 shows an example of the time overhead of our implementation of the k-Means algorithm when applied on the in-memory performance data of a series of NAMD **stmv** simulations up to 19,200 processors on a Cray XT5 (kraken). The overhead currently includes the time required to parse the in-memory event traces in order to compute the metric data necessary as input to the k-Means implementation. This explains the shape of the curve as individual processors for runs with smaller numbers of processors have to hold more event trace data given the same simulation problem size. The parsing of the event traces can be avoided if the necessary metric data are accumulated as the events occur. Note that the overhead here does not perturb the application's performance characteristics. The overheads represent the extra time the application will consume as part of its job allocation on the supercomputing facility. The parallel k-Means algorithm is applied to recorded performance information stored on each processor's log buffers at the end of the application's execution but before any data is written to disk. Once the retention decisions have been made, performance data that is to be retained will be written to disk.

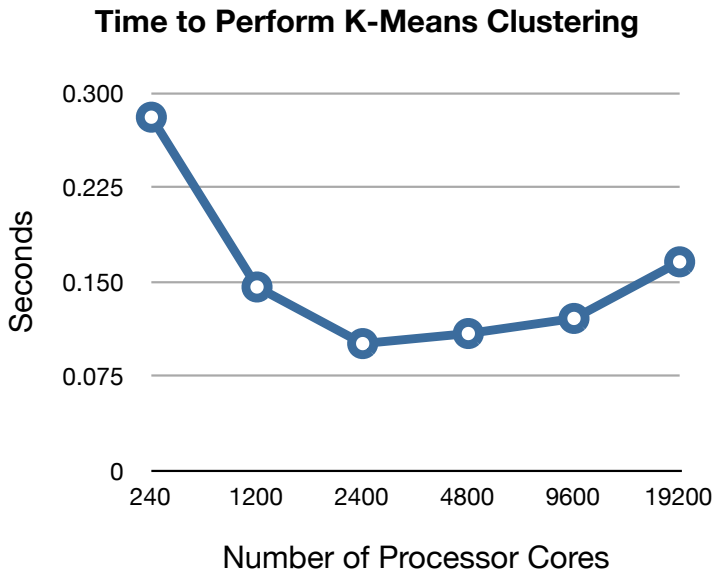


Figure 4.7: Overhead of applying parallel k-Means implementation on NAMD **stmv** performance data.

4.6 Case Study: NAMD Grainsize

4.6.1 NAMD - NAnoscale Molecular Dynamics

NAMD is a molecular dynamics program designed for high performance simulation of large biomolecular systems [62]. Each simulated timestep involves computing forces on each atom, and "integrating" them to update their positions. The forces are due to bonds, and electrostatic forces between atoms within a cut-off radius.

NAMD is parallelized using Charm++ via a novel combination of force and spatial decomposition to generate enough parallelism for parallel machines with a large number of processors. Atoms are partitioned among cubes whose dimensions are slightly larger than the cutoff radius. For each pair of neighboring cubes, we assign a non-bonded force computation object, which can be independently mapped to any processor. The number of such objects is therefore 14 times (26/2 pairwise interactions + 1 self interaction). One cell has 26 neighbors but only needs to calculate half of them since cell-to-cell forces are symmetric, the additional one is the self force calculation.

The cubes described above are represented in NAMD by objects called *home patches*. Each home patch is responsible for distributing coordinate data, retrieving forces, and integrating the equations of motion for all of the atoms in the cube of space owned by the patch. The forces used by the patches are computed by a variety of *compute objects*. There are several varieties of compute objects, responsible for computing the different types of forces (bond, electrostatic, constraint, etc.). On a given processor, there may be multiple "compute objects" that all need the coordinates from the same home patch. To eliminate duplication of communication, a "proxy" of the home patch is created on every processor where its coordinates are needed. The parallel structure of NAMD is shown in Fig. 4.9.

NAMD employs Charm++'s measurement-based load balancing. When a simulation begins, patches are distributed according to a recursive coordinate bisection scheme, so that each

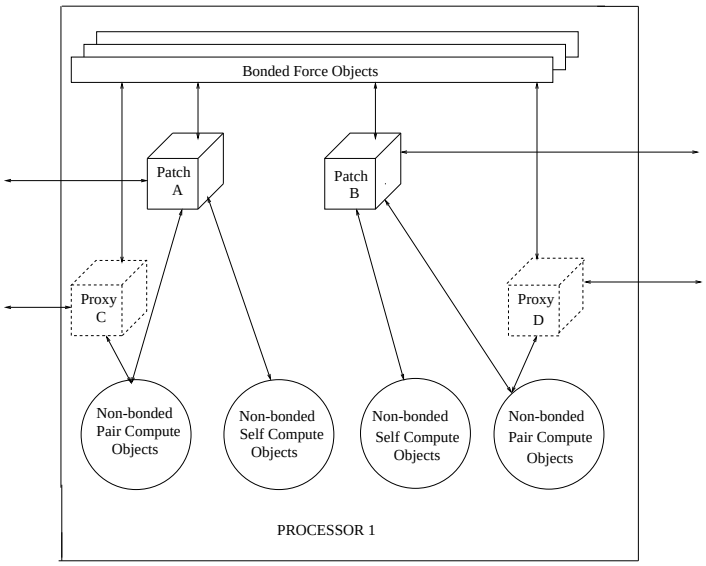


Figure 4.8: NAMD 2 hybrid force/spatial decomposition. Atoms are spatially decomposed into *patches*, which are represented on other nodes by *proxies*. Interactions between atoms are calculated by several classes of *compute objects*.

processor receives a number of neighboring patches. All compute objects are then distributed to a processor owning at least one home patch. The framework measures the execution time of each compute object (the object loads), and records other (non-migratable) patch work as "background load." After the simulation runs for several time-steps (typically several seconds to several minutes), the program suspends the simulation to trigger the initial load balancing. The strategy retrieves the object times and background load from the framework, computes an improved load distribution, and redistributes the migratable compute objects.

The initial load balancer is aggressive, starting from the set of required proxies and assigning compute objects in order from larger to smaller, avoiding the need to create new proxies unless necessary. Once a good balance is achieved, atom migration changes very slowly. Another load balance is only needed after several thousand steps.

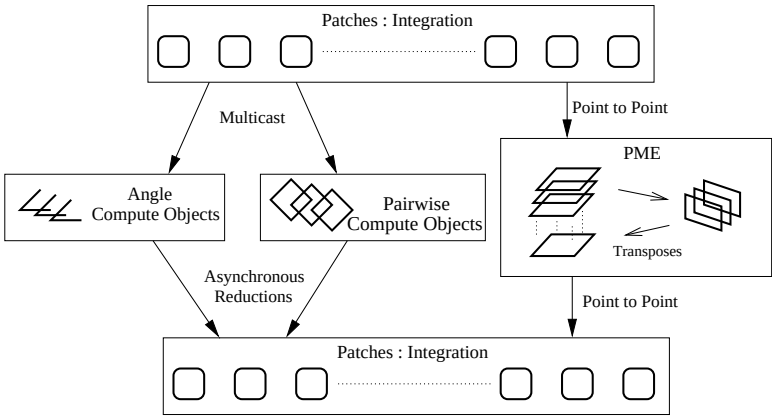


Figure 4.9: Parallel structure of NAMD

4.6.2 Experimental Methodology

The goal of our experiments is to investigate the effectiveness of our processor selection heuristics. We determine effectiveness to comprise of two components. The first is a quantitative measure of how much data was reduced. The second is an evaluation of the quality of the reduced dataset with respect to performance problem discovery which, in our case, is performed using Projections.

Our experiments are based on the 1 million atom NAMD simulation of the complete satellite tobacco mosaic virus (**stmv** in table 4.1). We made the runs from 512 to 4096 processors on the Cray XT3 installed at Pittsburgh Supercomputing Center through Teragrid [13] resources.

To enable an assessment of our technique, we injected a known poor-grainsize performance problem as described in our case study paper [44] into the simulation. The poor grainsize was caused by parallel objects whose electrostatic force computations require interaction with other objects hosting cubes that share a common face. This would manifest itself as a bimodal "camel hump" in a histogram plot that would not show up otherwise when the same plot was made of performance data from a run without the problem injection.

In our histogram plot, we display a stacked graph of occurrence counts of each instrumented CHARM++ entry method against the time it took. The histogram covers the occurrence of entry methods that range from 0.1 ms to 10.0 ms over 99 bins. The occurrence count data is

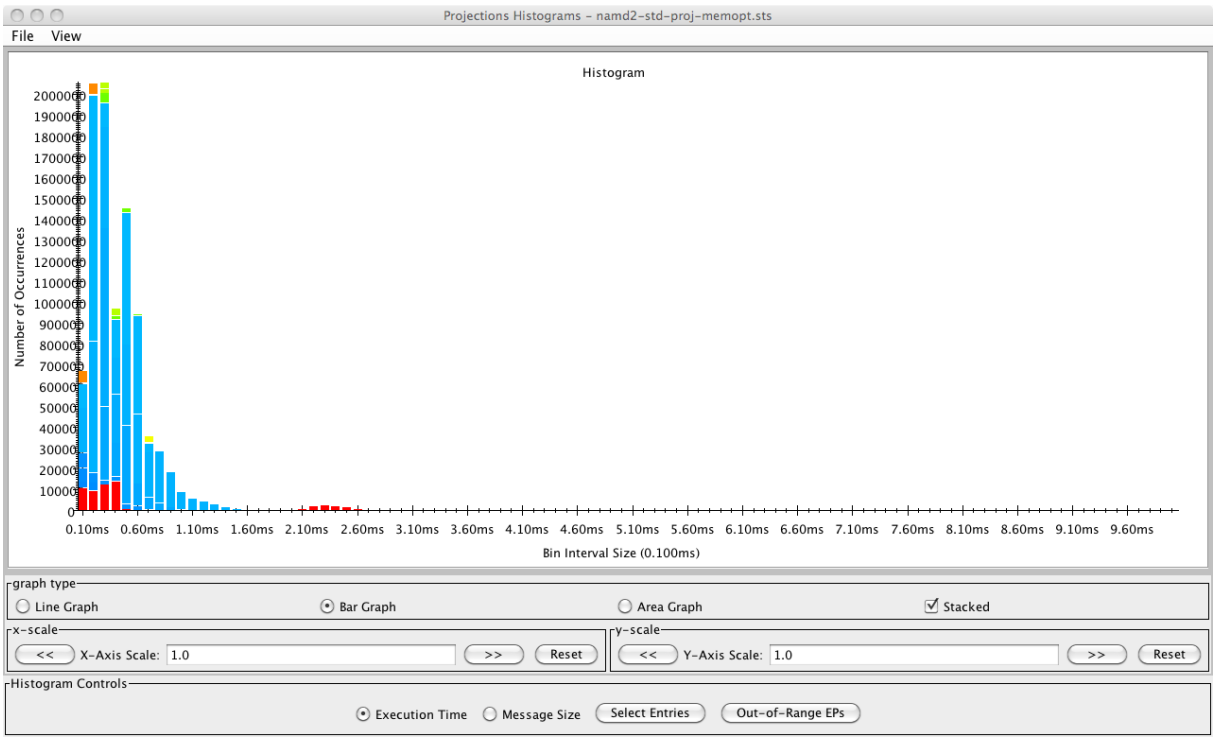


Figure 4.10: Histogram plot of **stmv** in Projections for 4,096 processors. The vertical bars show the number of occurrences of CHARM++ entry methods, distinguished by their colors, that took a certain amount of time to execute. The first bar shows the number of entry methods that executed for 0.1 ms to 0.2 ms, the second for 0.2 ms to 0.3 ms, etc

summed over all 4,096 processors over the 200 NAMD iterations. Figure 4.10 shows what the histogram looks like without the injected performance problem and the corresponding bimodal histogram after problem injection is shown in figure 4.11. Figure 4.12 shows the histogram chart with the poor-grainsize problem injected using the reduced performance data from only 409 representative processors. The counts can be seen to be approximately 10% of the full dataset.

Projections currently does not apply any proportional modifications to performance information of the processor representatives in order to extrapolate their contribution. As a result, we determine the quality of the reduced data set by two criteria. Let H_r^i be the total occurrence counts for the i-th histogram bar in the reduced data set. Let H_o^i be the total occurrence count for the i-th histogram bar in the original full data set. Let P_r be the number of processors in the

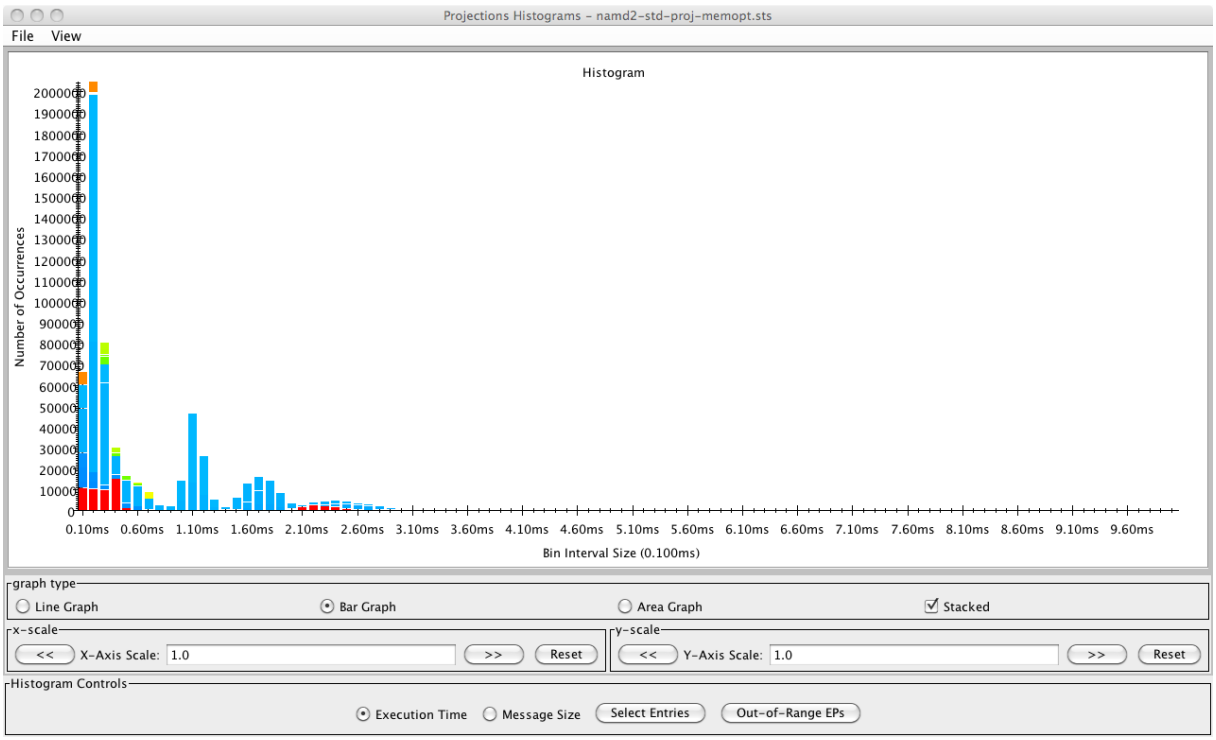


Figure 4.11: Histogram plot of **stmv** with poor-grainsize in Projections for 4,096 processors. Note the shift of the number of CHARM++ entry method occurrences rightward where the bins represent longer execution times.

reduced data set and let P_o be the number of processors in the original full data set. Our first criterion states that for each i, $\frac{H_r^i}{H_o^i}$ should be close to $\frac{P_r}{P_o}$ where $H_o^i \neq 0$. Our second criterion states that across all i where $H_o^i \neq 0$, $\frac{H_r^i}{H_o^i}$ should not vary by too much. We will refer to these two criteria as the *proportionality criteria*.

Varying Cluster Seed Counts

The value of k is in fact the number of initial seeds used in the clustering algorithm. Depending on seeding policy, they may or may not ultimately represent non-empty clusters. We made some trial and error experiments with NAMD, varying the number of processors and k, summarized in table 4.3. The results for $k = 15$, we felt, appeared more or less consistent with the number of processor-classes we have observed in the past while studying the performance of NAMD. The observed classes include processor 0 which has to perform special tasks, as well as certain

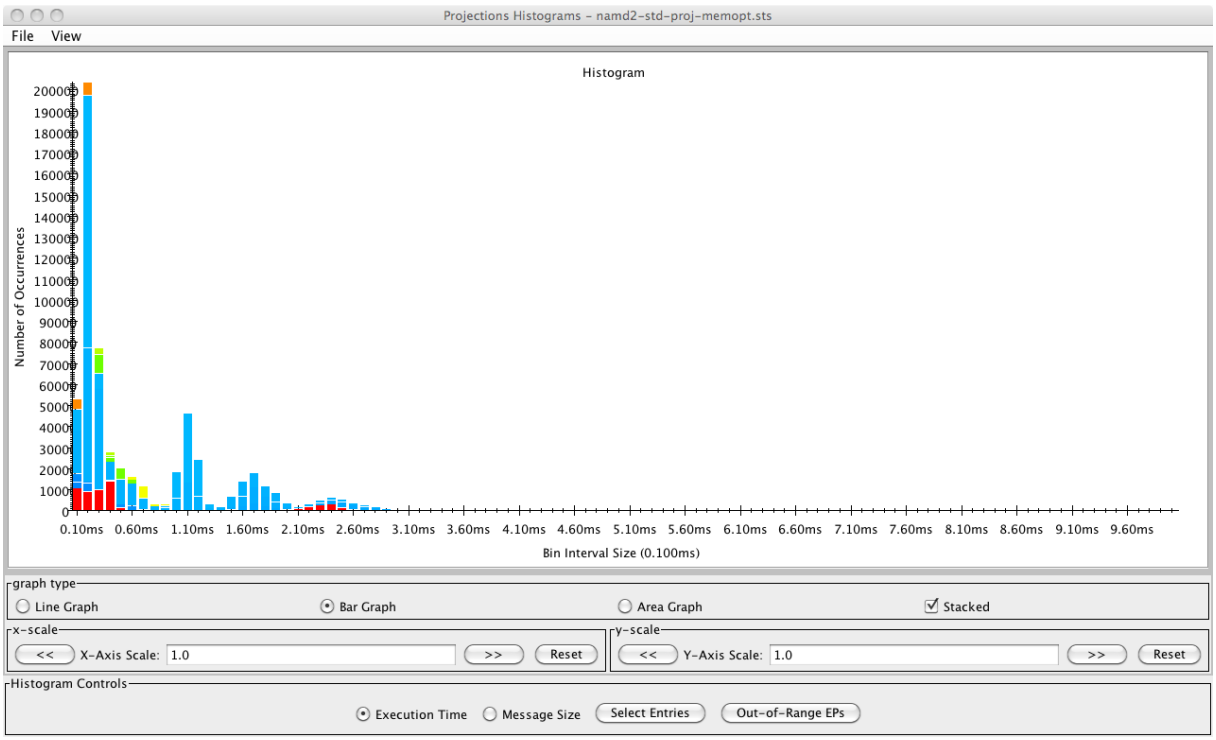


Figure 4.12: Histogram plot of **stmv** with poor-grainsize in Projections using only the 409 logs from the reduced dataset.

processors in NAMD which, when assigned certain work objects are not allowed to be assigned certain other types of objects. As a result, we chose to set k to 15 for the identification of clusters in our subsequent experiments. The initial seeding of the k starting points are, for now, uniformly distributed in the E dimensional space.

4.6.3 Results

We first show the reduction in trace data volume in table 4.4, through the selection of subsets of processors that number approximately 10% of the original dataset. As we can see, the reductions come as no surprise, although it is important to note that the number of traced events can vary significantly between processors. One cannot trivially expect to see a perfectly linear reduction in data volume.

For the quality measure, we applied the *proportionality criteria* to the reduced trace data

nCPUs	Number of non-empty clusters found with				
	5 seeds	10	15	20	25
512	1	4*	4*	6*	6*
1024	2	4	6*	6*	7*
2048	2	4*	5*	6*	7*
4096	3	6*	7*	9*	10*

Table 4.3: Number of non-empty clusters found by clustering algorithm when varying the number of initial seeds uniformly distributed in the sample space The * indicates that processor 0 was alone in its own cluster. *Reprinted, with permission, from "Towards Scalable Performance Analysis and Visualization through Data Reduction" by Chee Wai Lee, Celso Mendes and Laxmikant V. Kalé at the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008, ©2008 IEEE.*

nCPUs	original size	reduced data
512	2,800 MB	275 MB
1024	3,900 MB	402 MB
2048	4,800 MB	551 MB
4096	5,700 MB	667 MB

Table 4.4: Reduction in total volume of trace data for **stmv**. The number of processors selected in the subsets are 51, 102, 204 and 409 for 512, 1024, 2048 and 4096 original processors respectively. *Reprinted, with permission, from "Towards Scalable Performance Analysis and Visualization through Data Reduction" by Chee Wai Lee, Celso Mendes and Laxmikant V. Kalé at the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008, ©2008 IEEE.*

generated from full datasets ranging from 512 to 4096 processors and with processor reduction ranging from approximately 5% to approximately 20% of the original number of processor logs. This is summarized in table 4.5. In this table, $\overline{H} = \frac{\sum_{i=0}^{n} \frac{H_r^i}{H_o^i}}{m}$ where $H_o^i \neq 0$. For our experiments, $n = 100$ and $m = n - k$ where k is the number of instances where $H_o^i = 0$ (In other words, we ignore histogram bars where proportionality has no meaning). We use $\overline{H}$'s closeness to $\frac{P_r}{P_o}$ as the measure to satisfy the first criterion of the *proportionality criteria*. Likewise, the standard deviation σ over $\frac{H_r^i}{H_o^i}$ for all i where $H_o^i \neq 0$ is used as the measure to satisfy the second criterion for reduced data set quality. From the table, we see that with the exception of the data from 512 processors, the proportionality of the histogram bars generated

from reduced data sets matches very well with what is expected of the data. In addition, the standard deviation values appear to be good. We believe the 512 processor data sets involve entry methods of larger grain size, making them more sensitive to variation across processors. In any event, the values are not terribly off-the-mark although improvements may be possible.

We have visually confirmed the quality of the data using Projections on the partial processor logs generated by our approach.

P_o	$\frac{P_r}{P_o}$	$\overline{H}$	**Standard deviation** σ
512	0.0488	0.0641	0.00732
	0.0996	0.1180	0.00768
	0.1992	0.2237	0.00732
1024	0.0498	0.0511	0.00168
	0.0996	0.1008	0.00157
	0.1992	0.1921	0.00264
2048	0.0498	0.0487	0.00122
	0.0996	0.0977	0.00216
	0.1992	0.1883	0.00575
4096	0.0498	0.0501	0.00170
	0.0998	0.0981	0.00203
	0.1997	0.1975	0.00163

Table 4.5: Reduced dataset quality by proportionality based on total height of histogram bars. *Reprinted, with permission, from "Towards Scalable Performance Analysis and Visualization through Data Reduction" by Chee Wai Lee, Celso Mendes and Laxmikant V. Kalé at the 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008, ©2008 IEEE.*

To gain yet another perspective on the quality of data reduction via this approach, we measured how well the reduced data set matches the set of least idle processors. We consider the detailed performance event traces of the latter to be examples of useful traces as they tend to be frequently picked out when using the Extrema Tool (see section 3.3.3) for timeline study. It is important to note that the set of least idle processors do not completely define the set of useful processors for performance analysis purposes.

We first compared the list of processors generated as a result of our data reduction technique against the sorted list of processors with the least idle time. We then count the number of

processors that are found in both lists and tally those counts according to several strata for least idle processors:

1. the top 5 least idle processors;

2. the top 10 least idle processors; and

3. the top 20 least idle processors.

We did this using data sets from different total processor counts and varied the degree of data reduction. As a control, for each experiment, we also made the same comparisions using the sorted list of least idle processors against a list of processors randomly selected with the same degree of data retention.

Tables 4.6 and 4.7 show the results using this method of measuring quality for data reduction on 2,048 and 4,096 processors respectively. The rows represent the 3 different strata of top least idle processors outlined above. The columns represent the different degrees of data reduction as a percentage of the total processors retained. The values show the percentage of the top least idle processors of each strata that were found in the reduced data sets. Note that the set of processors in the smaller strata are included in the set of processors in each larger strata (i.e. all the top 5 least idle processors are also necessarily part of the set of the top 10 least idle processors).

Strata	**5% Retention**	**10% Retention**	**15% Retention**
Top 5 least idle	100%	100%	100%
Top 10 least idle	70%	90%	100%
Top 20 least idle	45%	70%	95%

Table 4.6: Measure of data reduction quality through number of least idle processors retained in reduced datasets for 2,048 processors.

The results are promising. In the 2,048 processor case, we see that as we increase the percentage of retained processors to 10% of the total number of processors, we were able to capture through our retention heuristics most of the top 20 least idle processors. Even at 5%

Strata	2.5% Retention	5% Retention	7.5% Retention
Top 5 least idle	40%	100%	100%
Top 10 least idle	20%	70%	100%
Top 20 least idle	10%	45%	100%

Table 4.7: Measure of data reduction quality through number of least idle processors retained in reduced datasets for 4,096 processors.

retention, all of the top 5 least idle processors were captured in the reduced data set. In the 4,096 processor case, we see that we only needed 7.5% retention to include each of the top 20 least idle processors in our reduced dataset. Note that in all the cases, the number of top least idle processors retained were well above the expected number were random selection to be used instead.

4.7 Future Work: Extensions To The Basic Approach

In this section, we discuss possible extensions to the data reduction approach. The goal is help increase the reduced performance dataset's usefulness to performance analysis idioms that involve the examination of detailed performance information.

4.7.1 Choosing Data Subsets By Phase

A weakness of the basic approach described above is the application of the k-Means algorithm over the entire execution time of the simulation. Since the metrics used are summed over the whole run, the clustering algorithm is blind to time-dependent variations in application behavior.

Consider the extreme instance where a program does no work on the first half of its processors and fully utilizes the other half in one phase. Then on the next phase, it fully utilizes the first half of its processors and does nothing on the second. Applying clustering on this data without any time or phase sensitivity will result in the algorithm convinced there is only one equivalence class of processors when in truth there are two (see Figure 4.13).

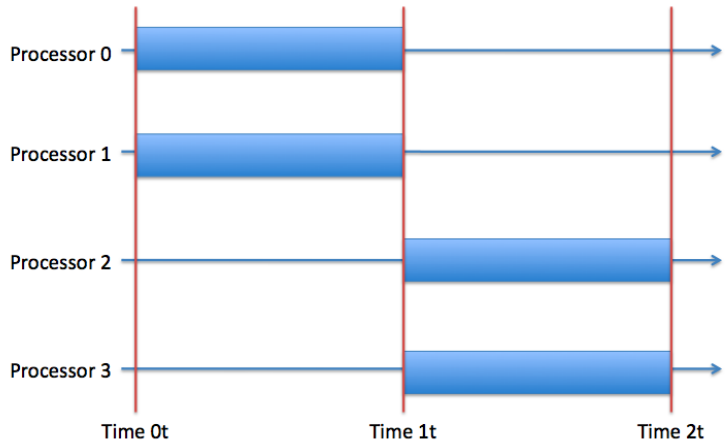


Figure 4.13: Simple example of time-variation that will confuse a clustering algorithm into believing all 4 processors behave similarly based on metrics recorded between time 0t to 2t.

To handle time-dependent variation in application performance, we add phase information into the performance event traces. With this information, we can apply the clustering and representative selection approach to the performance logs at each phase. If the clustering algorithm indicates for a phase that a processor is a member of a cluster and behaves closely enough to a cluster exemplar, all the events recorded over the time period of that phase can be replaced by a single surrogate event. The surrogate event simply identifies the exemplar processor and the phase identifier. This then allows visualization components to adopt different schemes for handling the unrecorded data. For example, aggregation-based visualization tools can read or generate the necessary profiles from the appropriate sections of the exemplar processor's logs. Timeline tools can do something as simple as visualizing the unrecorded region as an opaque time-block. Timeline tools can also display the details from the exemplar processor but highlight the fact that the visualized details are surrogated.

The volume of data reduction achieved by applying clustering to each phase differs slightly from the approach of applying clustering to entire processor logs. The differences will depend mainly on event densities on each processor over each phase. The insertion of phase and surrogate entries into the logs should not impact overall volume significantly.

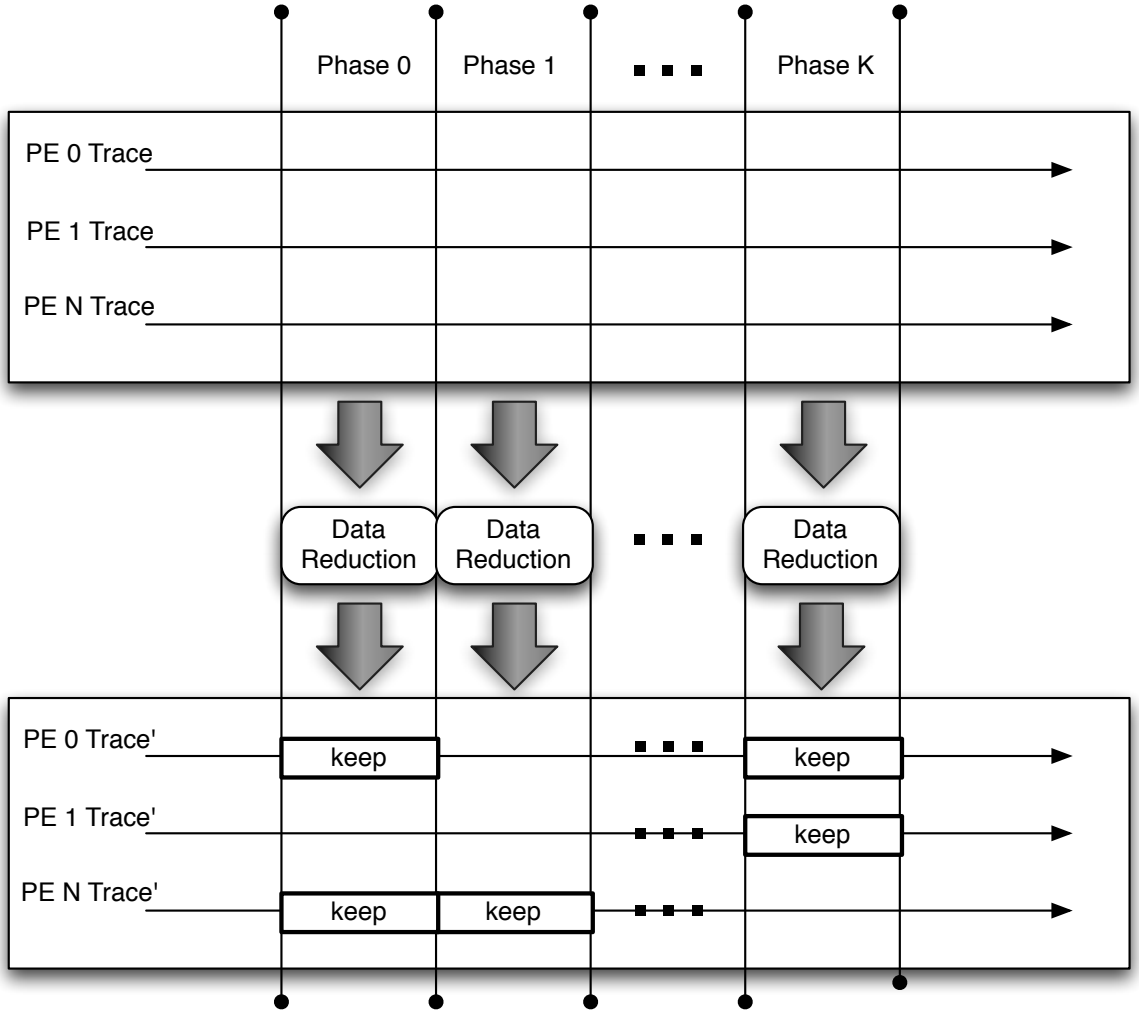


Figure 4.14: Application of data volume reduction techniques to individual execution phases.

4.7.2 Considerations For Critical Paths

Choosing processor subsets using the algorithms described above is likely to result in the loss of interprocessor dependency information. As this is a strong point for the use of event trace logs in analysis in the first place, we would want to preserve as much of this information as possible.

Dooley and Kalé [22] describes how the top critical path can be derived and stored in memory as a CHARM++ application executes. We can make use of this work to identify all proces-

sors that lie on the main critical path. Of interest to us is the union of the set of representative processors chosen by our approach and the set of processors that lie along the critical path. The combined set could potentially be large. Ideally, one would like all selected representatives to be connected along the critical path. We can do so by first finding the intersection of the set of representative processors and the set of processors on the critical path. We can then add more processors that lie along the critical path, selecting those that connect as many of the processors in the intersection as possible.

Chapter 5

Online Live Analysis

Post-mortem performance analysis and visualization tends to be the norm in parallel computing. This common approach is used primarily due to the lack of availability of continuous performance monitoring tools. Continuous performance monitoring is an approach where performance characteristics of a running parallel application are used as the program runs by a performance analyst. This approach has several benefits when used either by itself on in conjunction with post-mortem analysis tools. Continuous performance monitoring as an analysis idiom, although not yet widely used, could become more popular if it were technically feasible, and if it could be deployed without significantly degrading the performance of the running application.

In this chapter, we begin with the discussion of the current literature on large-scale online performance observation or monitoring [1]. We then describe our approach using the Converse Client Server (CCS) framework and how that, along with the CHARM++ adaptive runtime, enables powerful new scalable analysis capabilities difficult to achieve with the use of only post-mortem event traces.

5.1 Related Work

Distributed analysis and interaction with large running parallel applications is being studied in a number of projects to manage scalability as well as provide new analysis capabilities. Malony

[1] This chapter is written with portions reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the *16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009)*, ©2009 IEEE

et. al. [51] described a position for online analysis which incrementally led to the adoption of both online frameworks as a means to efficiently carry performance data while an application is still executing.

Supermon [72] is a cluster monitoring framework which makes use of a network of daemons called "mons" that communicate with a root daemon by the name of "Supermon". It was originally designed for monitoring system information through a kernel module. System level information is "concentrated" through the mon daemons to the Supermon daemon and then processed by connecting clients. However, because each mon daemon also supports the input of data from outside sources through a socket interface they call the "monhole", tools like TAU have been able to adapt to it through the TAUoverSupermon project [58] by hosting a mon daemon along with each application process and feeding performance data through the monholes. TAUoverSupermon has so far shown low overhead up to 512 processors.

MRNet [65] was designed directly as a framework for parallel performance analysis with the Paradyn system. It shares many of Supermon's features, with back-end processes serving the role of mons as used by TAU. These back-ends are connected to a root node by a tree of intermediate node processes intended to serve as an efficient multicast and reduction network. The authors suggest that while intermediate nodes may be co-located with application processes, perturbation makes that option sub-optimal. TAU over MRNet [57] demonstrates the framework's use with a performance tool. The paper reports overhead of less than a percent up to 512 processors.

The two frameworks described above differ from our approach (see Section 5.3) in that we directly exploit the natural abilities of the CHARM++ runtime system. The runtime system is able to adaptively schedule and interleave the collective operations for performance data collection efficiently along with application computation and communication. There is no need to avoid co-locating any intermediate processes like in the case of MRNet. Overhead costs were found to be at most 1.14% up to 8,192 processors, which is competitive with the overhead costs of the other frameworks.

5.2 The Converse Client-Server Interface

Converse Client-Server (CCS) [18] is a communication protocol that allows parallel applications to receive requests from remote clients. This protocol is part of CHARM++'s underlying system specifications and is therefore available to any CHARM++ application. Note that "application" does not mean only the user written code, but also the CHARM++ runtime system and its modules that run as part of the application itself. In this scenario, if a system module decides to use CCS, the user code does not require any change, unless it wants to explicitly take advantage of the feature. We shall see some examples of this in later sections.

CCS obeys normal CHARM++ semantics. Upon a request made by a CCS client, a message is generated inside the application. Computation by the application is triggered by the delivery of this message. As such, CCS requests are serviced asynchronously with respect to the rest of the application, which can proceed unaffected. When an application, or CHARM++ module, desires to use the CCS protocol, it must register one or more handler, each with an associated tag. This ensures that requests sent by clients can be correctly matched and delivered to the intended handler. Registration is performed by calling a function into the CCS framework. Moreover, at startup, a flag must be passed to the application to ensure that the runtime system opens a socket and listens for incoming connections. The connection parameters are printed to standard output by the CHARM++ RTS. Remote clients can send requests to the parallel application using this information. After receiving a CCS request message, the application can perform any kind of operation, including complicated parallel broadcast and reductions. Finally, a reply can be returned to the client via the CCS protocol.

5.3 Continuous Streaming Of Online Performance Data

We begin the discussion of our approach by observing that the power of a parallel computer can be applied towards some forms of analysis as performance data is produced or captured.

Portions of the analysis can be quickly executed while the performance data is still in memory. We will demonstrate one such example where a monitoring tool can add up profile information across all processors using reduction operations on the parallel computer itself. In contrast, a post-mortem tool would have to spend considerable time processing potentially large per-processor trace files on disk to generate the same information.

In the post-mortem approach, the data for a long performance trace or profile gets buffered until the end of the run. The data is then transfered all at once via the file system to disk, which takes significant time and parallel file system resources. This does not need to be so. After a piece of performance data is recorded, it sits unused on the assigned memory buffers. The longer a piece of data resides in memory, the greater the missed opportunity for a system to efficiently deliver the data to a performance tool. A pre-processed and efficiently compressed performance stream, coming continuously from the parallel computer during the program's run can possibly reduce the time spent to get performance data to a tool. The performance tool receiving this stream can store or display the data and can do so selectively if need be. In fact, once the performance data in question is processed and streamed to a tool, it is no longer necessary to keep it in memory. Buffer memory can now be freed and re-used for the capture of more performance data, enabling the capture of performance data for jobs running far longer.

For a long running job, a continuous performance monitoring tool can be used to look for unacceptable performance degradation in the performance stream. In some applications, the evolution of the application may cause its performance to degrade due to gradual or abrupt factors such as adaptive refinement, change in material properties (for finite element structural simulations), etc. Some occasional events and glitches such as long-duration OS interference, unexpected staggered I/O cause strong performance degradation, with lingering effects (due to shifting of computations), can be detected by monitoring a continuous performance stream. If the user can check the performance characteristics from time to time as the program runs, they will be able to terminate the job when such a scenario is reached. Typically, they can restart the simulation from the last saved checkpoint (with a new modified mesh partitioning, for exam-

ple). A continuous performance monitoring tool is also a prerequisite for online performance steering: visual performance feedback, and specific details emerging from it, may be used by the analyst to tell the application to adjust some parameters or to trigger a runtime load balancing phase, etc. These are all extremely powerful and scalable performance analysis idioms previously hard to achieve using only post-mortem event traces.

An adaptive runtime system such as CHARM++ makes it easy for a performance monitoring module to send messages that are independent of the messages being sent by the rest of the parallel program. Adaptive overlap (Section 2.1.2) allows out-of-band communication which in turn is the key to an efficient implementation of a framework for the delivery of continuous performance data. We will show how the CCS framework described in section 5.2 is employed to provide the mechanism by which an external tool may then interact with the running CHARM++ application to ship the processed performance data out while the application is still running.

5.4 Live Parallel Data Collection Of Performance Profiles

We demonstrate the power of continuous performance monitoring by implementing a performance data collection and processing mechanism for entry method utilization profiles within the Projections framework of CHARM++. We have also implemented a visualization client which will contact the running CHARM++ application to access the performance data stream remotely. CCS is used to connect this visualization client to the specific compute node where the performance data will be collected.

The utilization profile data gathered in this new system describes the fraction of the execution time spent in each activity over some period of time. The utilization profiles produced by the tool contain a number of bins, each representing 1ms slices of the execution of the program. Some fine-grained information about the execution of the activities will be lost, but the resolution of 1ms should be suitable for a variety of analysis tasks, including visualization.

Figure 5.1 shows an overview of the architecture of the utilization profile tool. It shows

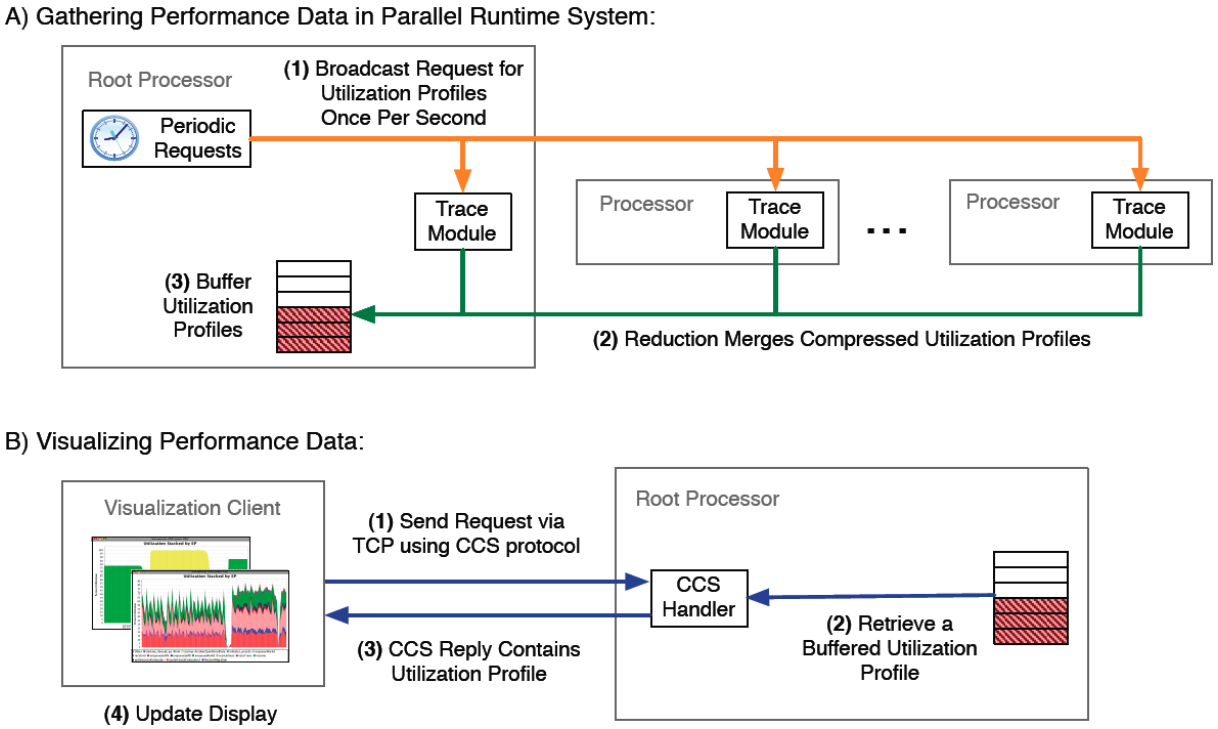


Figure 5.1: An overview of the Utilization Profile Tool. The tool is comprised of two separate mechanisms. The first mechanism (A) periodicially gathers performance data in the parallel runtime system. The second mechanism (B) allows a visualization client to retrieve the previously buffered performance data from the parallel program. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE*.

two independent mechanisms; one that comprise the Projections trace module which generates utilization profiles, with a performance statistics gathering portion in the parallel runtime system which compresses and merges these profiles through the CHARM++ reduction framework; and another that provides the mechanism for the visualization client to retrieve the statistics produced by the first mechanism. The details for each of these components are explained in the subsequent paragraphs.

On the parallel system, each processor creates and updates a utilization profile containing the amount of time spent in each entry method. To observe and record the utilization for each of the entry methods, the Projections summary trace module is enabled with a single link time option `-tracemode utilization`. This module is responsible for accumulating the time spent executing each entry method into the appropriate utilization profile bins. Each bin, representing 1ms of walltime, contains a double precision floating-point value for each of the entry method. Hooks in the trace module are called after each entry method completes its execution, at which point the execution time is accumulated into the one or more bins spanning the execution of the entry method.

In memory, the bins are allocated as a fixed-size contiguous array which is treated as a circular buffer [2]. We arbitrarily chose to use a circular buffer with $2^{15} = 32768$ bins which spans about 33 seconds. The circular buffer is not compressed, and hence its size can be large. In typical Charm++ programs there are hundreds of different entry methods, many of which are never executed, and many that are only called at startup. For example, in a run of the NAMD application as a case study with our trace module enabled, there are 371 distinct entry method. The size of the circular buffer allocated on each processor for this program would therefore be about 32768 bins $\times \frac{371 \text{ entry methods}}{\text{bin}} \times \frac{8 \text{ bytes}}{\text{entry method}} \approx 93$ MB. Although this buffer is somewhat large, the sparse data it contains is compressed before being merged across all the processors [3]. Section 5.5 shows that the actual cost of this approach is low when used with the

[2] Developed by Isaac Dooley.

[3] Compression scheme developed by Isaac Dooley.

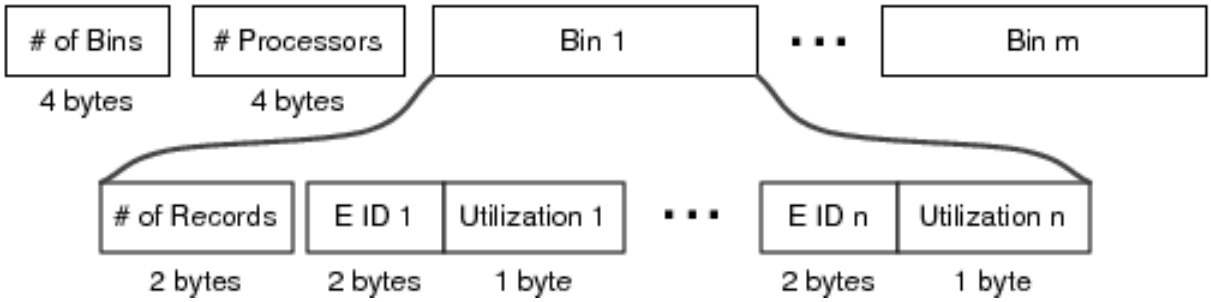


Figure 5.2: Compressed utilization profile format. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009)*, ©2009 IEEE.*

NAMD application.

If memory constraints are critical for a program, then the circular buffer could be reduced by a number of techniques. The number of bins could be reduced, the bins could cover larger amounts of execution time, and the set of entry methods could be reduced either by reducing the set of entry methods registered by the programs, or by reordering the entry methods and only recording information for the most important ones. A further way to shrink the memory requirement for this buffer would be to use single precision floating-point values instead of the 8-byte double precision values currently used.

Although each processor gathers its own utilization statistics in its trace module, the tool described will report the overall utilization across all processors. In order to efficiently combine the data from all processors, it is important to use an efficient data storage format for the communication intensive part of the data merging and transmission.

We created a compressed utilization profile format to use when transmitting the usage profiles between processors and to the visualization client [4]. This format, which is shown in figure 5.2, represents the utilization for many timeslice bins for one or more processors. The compressed format has a length that depends upon the number of bins and the number of entry

[4]Compression format developed by Isaac Dooley.

methods active during the timeslice represented by each of the bins. Within each bin, the utilization for each active entry method is stored using 3 bytes. One byte stores the utilization in a range from 0 to 250, a resolution that is reasonable for onscreen displays. Two bytes store the entry method index (E ID). The records within each bin are in sorted order by the entry method index. As we expect typical programs to have more than 256 entry methods, more than one byte is required to represent them all. The compressed format has a header which contains 4 bytes specifying the number of bins and 4 bytes specifying how many processors produced the utilization profile.

Because there will be some entry methods that contribute only tiny amounts to a bin, we decided to compress all such entry methods into a single *other* category which is stored just as any of the other entry methods in the bin but with a reserved entry method index. The entry method stored in this *other* category are any that fail to contribute more than a specified threshold to a bin. The threshold used is arbitrarily set at 10%. This merging of entry methods that contribute little to the result can reduce the sizes of the compressed utilization profiles significantly.

The compressed utilization profiles from all processors are periodically merged together and stored on processor zero, from which the visualization client can retrieve them. At startup, processor zero will instruct the runtime system to call a collection function once per second. This collection function in turn will broadcast a request to all processors. Each processor, upon receiving such a request, will compress a utilization profile of 1000 bins, and contribute the compressed profile to a reduction. The reduction is a standard Charm++ reduction with a custom reduction operation that merges any number of compressed utilization profiles into a single new compressed utilization profile. The reduction proceeds over a spanning tree, so it is scalable to large numbers of processors. The result of the reduction arrives at processor zero, at which point it is stored in a queue awaiting a request from a visualization client. The incoming reduction message containing the compressed utilization profile is itself saved, so no copying of the usage profile is required at the end of the reduction.

The custom reduction operation simply marches through all of the incoming compressed utilization profiles bin by bin, appending resulting merged bins to a new compressed utilization profile. Merging bins is simple because the entry method indices are already in a sorted order, so the minimal entry method index from all the incoming bins is selected and the utilization from any of the incoming bins for that entry method index are averaged. This average is weighted by the number of processors that have contributed to each of the respective incoming bins. The weighted average is important if the reduction tree is not a simple k-ary balanced tree.

The time-signal broadcast, the compression of the utilization profiles, and the subsequent reduction will interleave with the other work occurring in the program. This interleaving could potentially produce adverse performance problems for the running parallel program. We will show however in section 5.5 that the overhead is reasonably low.

Finally, the communication of the utilization profiles from the running parallel program to the visualization client is performed through CCS. When the CHARM++ application in question is first submitted for execution, a command line option is used to specify the TCP port to be used for the CCS server-side. The visualization client will connect to this port. In our implementation, the parallel program will simply reply to any incoming CCS request of the appropriate type with the oldest stored utilization profile in the queue on processor 0. If no stored utilization profile is available, then an empty message is sent back to the client. The client will periodically send requests to the running program. Once a compressed utilization profile is returned, the visualization windows of the client are updated. For convenience, the visualization client also supports the ability for the profiles to be written to a file for future analysis and the ability to play back the profiles saved in a file.

The visualization client created to work with the new Charm++ trace module is implemented in Java [5]. The client uses the CCS Java client-side library to create a `CcsThread` which is used to send requests to the parallel program. When the visualization client receives a CCS reply, it saves the message, and computes the data for a scrolling display. The client provides a both

[5] Visualization client developed by Isaac Dooley.

a graphical interface to connect to a running parallel program and some resulting graphical visualizations described below.

The first main display in the visualization client contains a scrolling stacked utilization plot of coarse grained averaged utilizations. This display shows 10 seconds worth of data at a time, scrolling as new data is added. The plot is composed of 100 bars in the x-dimension. Each bar displays the average of 100 bins, or 100ms of execution time. This view provides a high level view of the overall utilization of the program as it runs. A second more detailed view displays data at the finer grain 1ms resolution. The final display in the visualization client shows the sizes of the incoming compressed utilization profiles. This view is useful when determining the overhead of our system. Section 5.5 will show the visuals generated by this system for NAMD as a case study.

5.5 Case Study: Long-Running NAMD STMV Simulation

The goal of this case study is to demonstrate how we were able to use our implementation for live performance streaming to view, in real-time, a NAMD **stmv** 1 million atom simulation over a range of different performance characteristics in the simulation's lifetime. We quantify the overhead encountered in the use of our approach and show the bandwidth required by the streaming protocol over time.

Figures 5.3, 5.4, and 5.5 show the low-resolution utilization breakdown of the simulation over various key stages of its performance profile over time. Figure 5.3 shows the simulation's behavior just after initialization. Figure 5.4 shows a snapshot of the boundary between the initial timesteps used to collect object-load information, the application of the initial greedy load balancing strategy (evidenced by the deep and wide trough near the middle of the plot), and the application of a refinement load balancing strategy (the deep but sharp trough near the right edge of the plot). Figure 5.5 shows the simulation's steady-state behavior.

When seeking more specific performance structure in the streamed profile, such as sub-

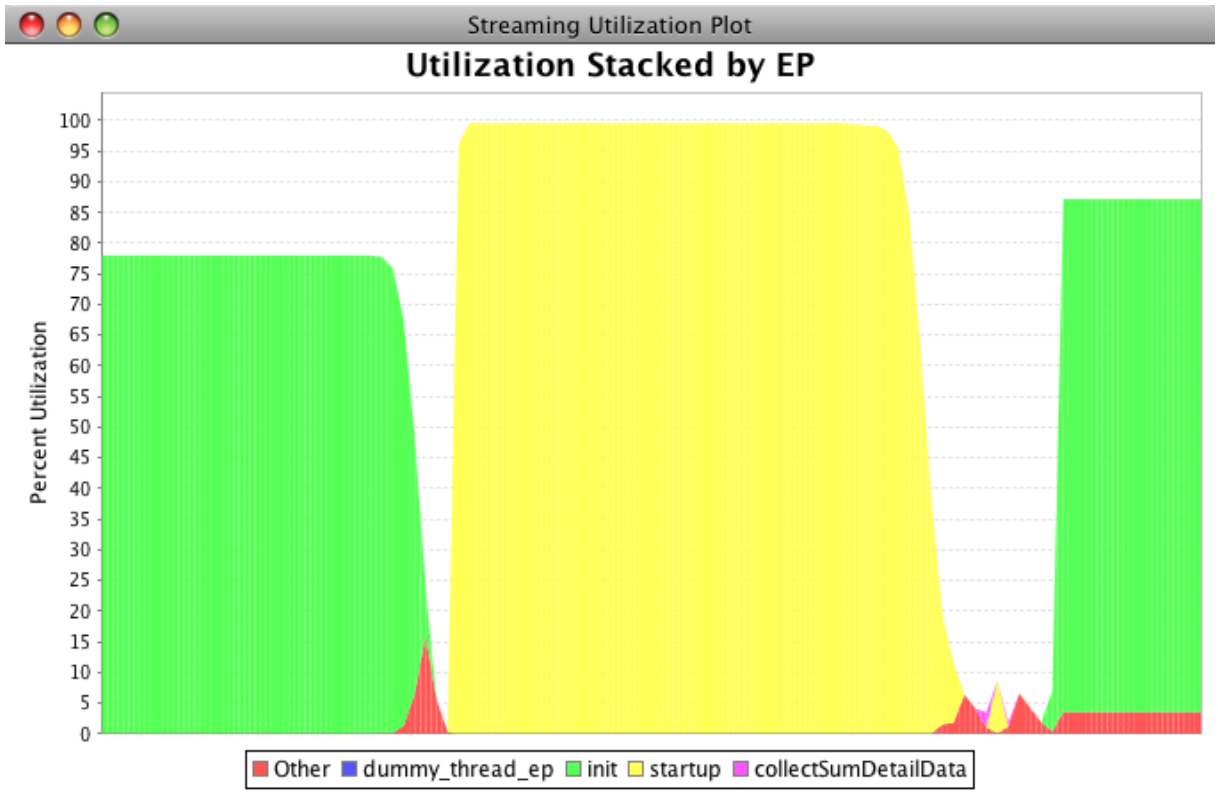


Figure 5.3: A screenshot of the streaming view in our tool. This view represents 10 seconds of execution during startup for NAMD on 1,024 processors running the STMV molecular system. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE*.

timestep structures, one can observe the simulation's behavior at higher resolutions. Figures 5.6 and 5.7 show snapshots of two such views. Figure 5.6 shows the detailed performance structure of simulation timesteps from before the application of the greedy load balancing strategy. Figure 5.7 shows the impact of the subsequent load balancing in detail.

To determine the actual performance impact of our performance metric gathering scheme, we ran the parallel NAMD molecular dynamics application [60] on 512 up to 8,192 processors of the Cray XT5 system Kraken at the National Institute for Computational Sciences managed by the University of Tennessee. We compared the application performance of a baseline version of NAMD containing no tracing modules to a version using the tracing module that gathers

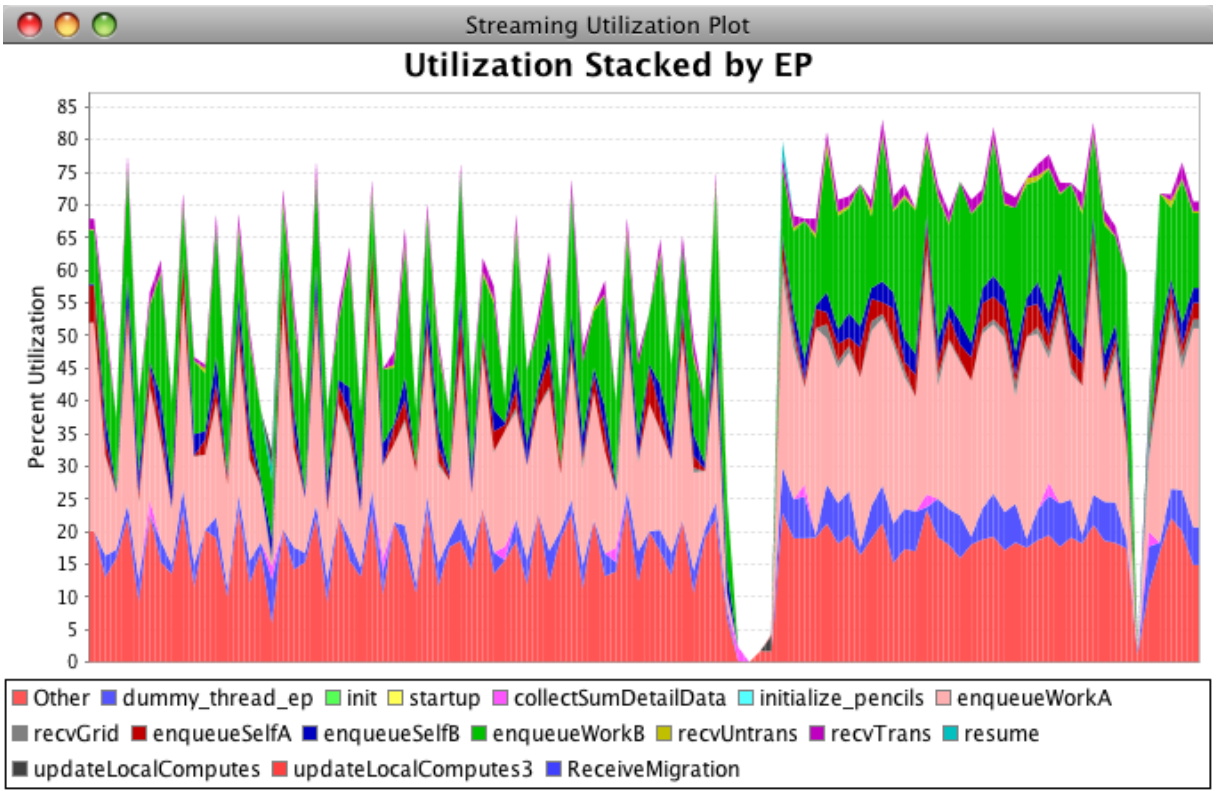


Figure 5.4: This streaming view represents 10 seconds of execution during the early steps when load balancing takes place. Two valleys, corresponding to the two load balancing steps, are clearly seen. This screenshot comes later in the execution of the same program run as in figure 5.3. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), *©2009 IEEE.*

only utilization profiles. We ran the baseline program once, and on the same allocation of processors we ran the version that gathers utilization profiles twice, once with a visualization client connected, and once without any visualization client connect. Timings were analyzed for 400 application timesteps well after the initial load balancing steps.

Our results show that for up to 8,192 processors, the overhead of recording the utilization profiles and sending them to a visualization client is at most 1.14%. The results appear to contain a slight amount of noise in that sometimes baseline version is slower than the version that gathers performance stastics. These variations are likely caused by intereference from other parallel jobs sharing the same interconnect and causing contention in the network.

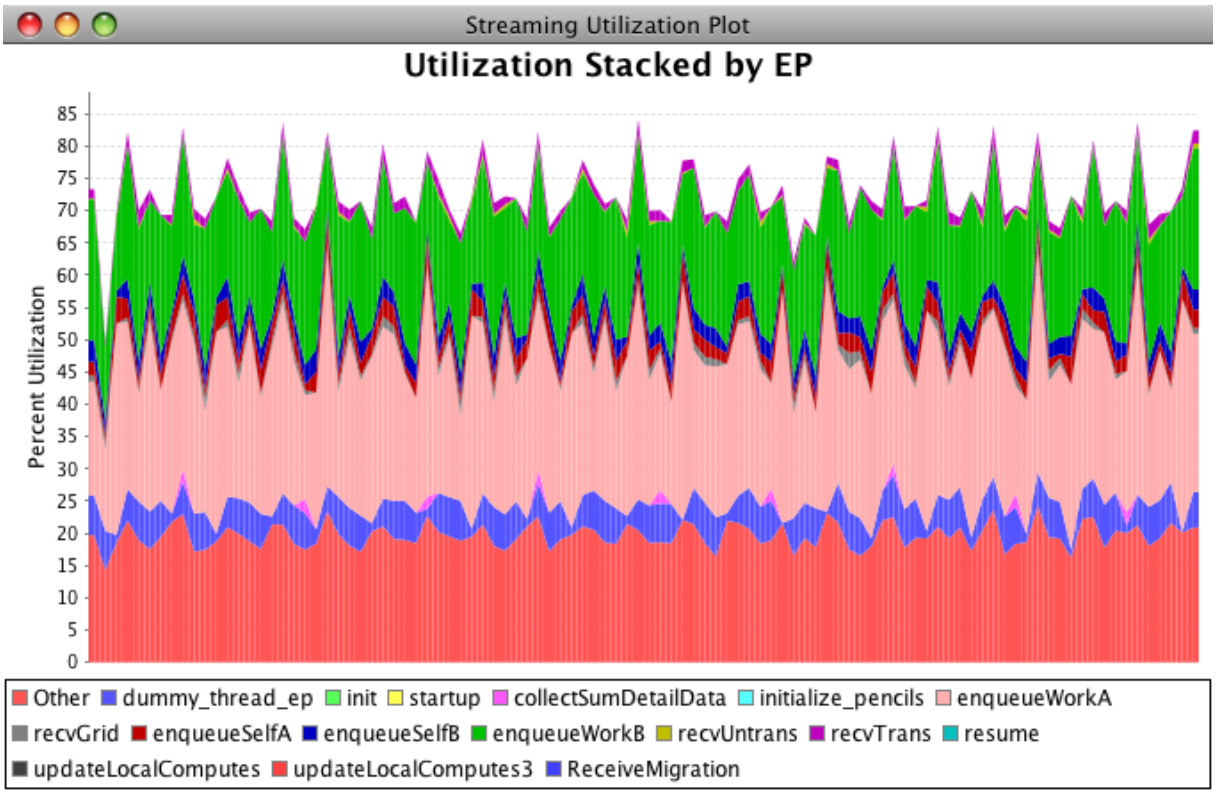


Figure 5.5: This streaming view represents 10 seconds of execution during the later simulation steps once a more uniform load balance has been achieved. Each bar represents the average utilization for 100ms of execution time. For this program, the timesteps are shorter than this duration and hence the utilization doesn't reach 100% in this plot. This screenshot comes later in the execution of the same program run as in figures 5.3 and 5.4. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE.*

We examined some projections trace logs for a 1,024 processor run of NAMD to determine the cost of creating the compressed utilization profiles. The time to create the compressed utilization profile from the uncompressed circular buffer was 9.3 ms. Because the uncompressed buffer is created once per second, we expect the overhead of our approach to be around $\frac{9.3ms}{1s} = 0.93\%$. The cost of the reductions was almost nonexistent in comparison to the compressing of the utilization profiles. The estimated 0.93% overhead seems to correspond well with the results actually obtained, modulo the noise.

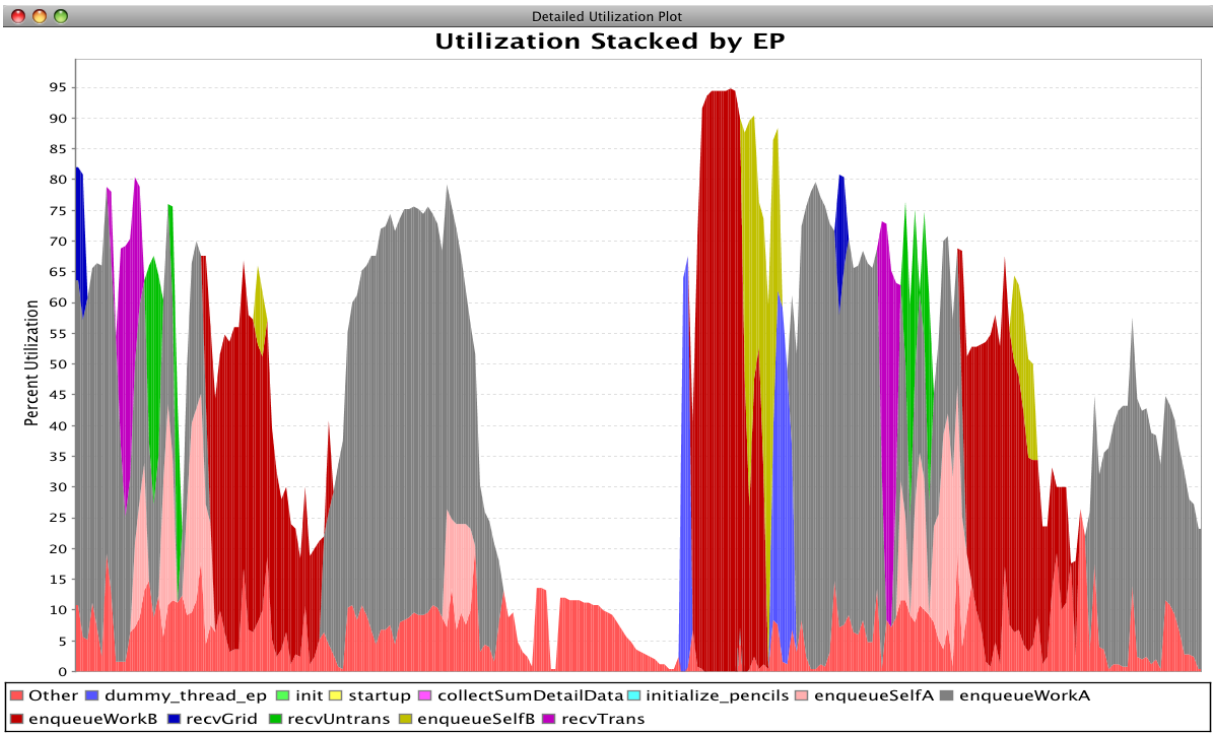


Figure 5.6: This detailed view plots the full 1ms resolution utilization profile for 0.25 seconds of execution time for a NAMD STMV run on 1,024 processors. This snapshot captures steps with poor load balance. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE.*

Processors	512	1024	2048	4096	8192
Overhead Without Visualization Client	0.94%	0.17%	-0.26%	0.16%	0.83%
Overhead With Visualization Client	0.58%	-0.17%	0.37%	1.14%	0.99%

Table 5.1: Overhead of collecting utilization profile instrumentation. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE.*

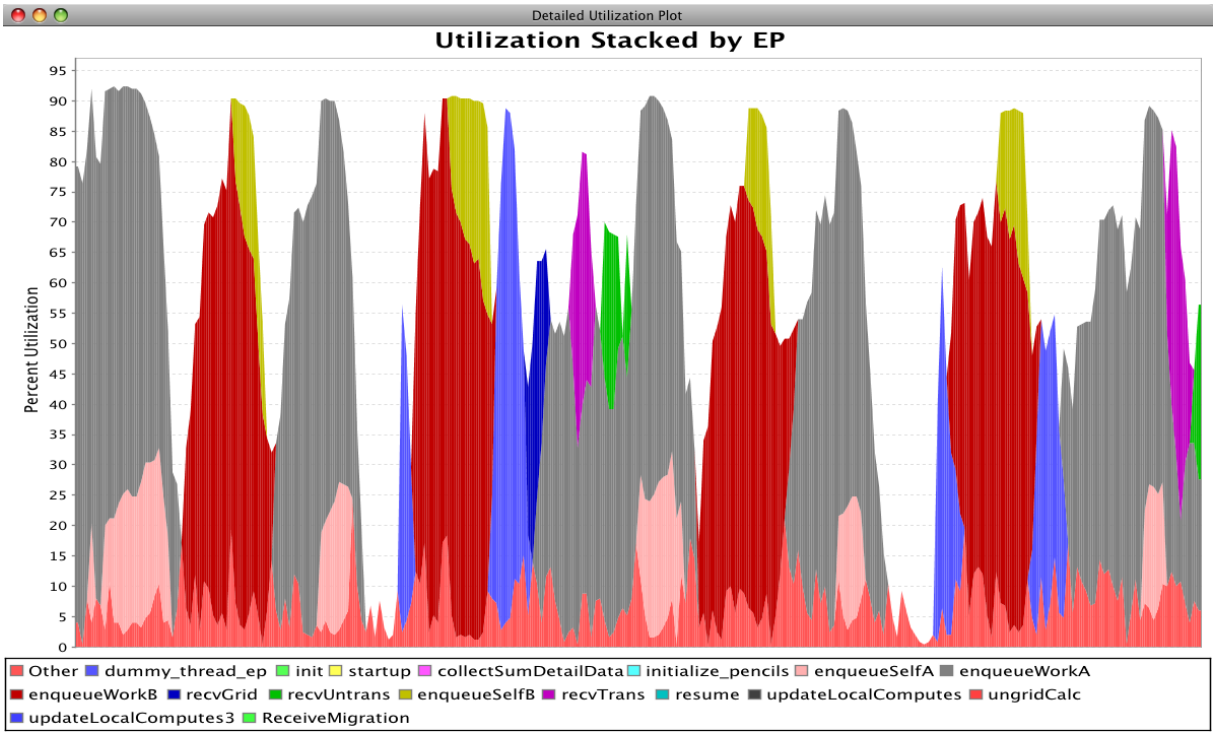


Figure 5.7: This detailed view shows a later plot from the same run shown in figure 5.6, after load balancing improves the performance by shortening the time per step. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE.*

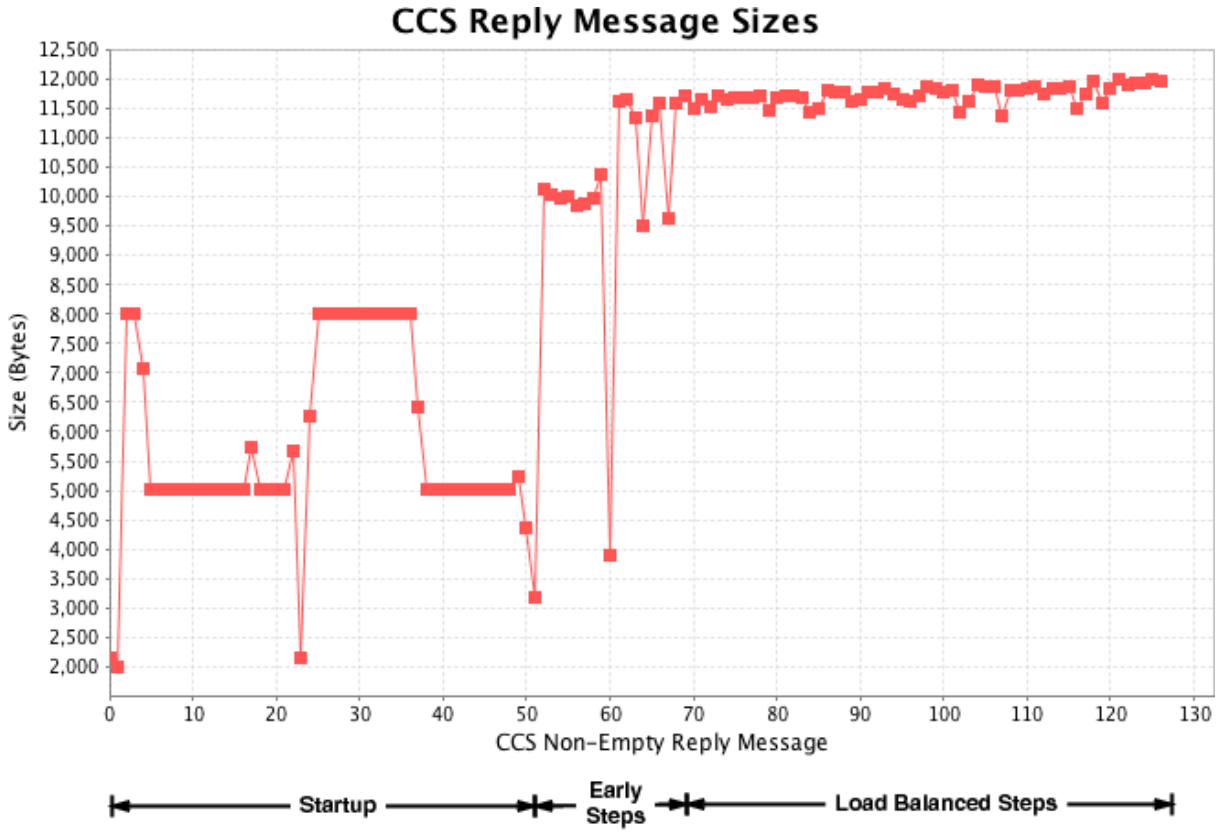


Figure 5.8: Sizes of the compressed utilization profiles received by the visualization client. The total bandwidth required to stream the data is thus under 12KB/second for this program, namely NAMD simulating the STMV system on 1,024 processors. *Reprinted, with permission, from " Continuous Performance Monitoring for Large-Scale Parallel Applications" by Isaac Dooley, Chee Wai Lee, and Laxmikant Kalé at the* 16^{th} Annual IEEE International Conference on High Performance Computing (HiPC 2009), ©*2009 IEEE.*

The messages sent along the reduction tree when combining the utilization profiles for all processors have sizes that range from 3.5KB up to almost 12KB. Figure 5.8 shows a plot of the resulting utilization profiles that were received by the visualization client when running NAMD on 1,024 processors. This figure shows that the sizes of the messages vary throughout different phases of the application. During startup, the computation involves few entry methods, and hence the message sizes are smaller. When the fine-grained timesteps are executing toward the end of the program, the utilization profile combined from all processors is approximately 12KB in size.

5.6 Future Work

The next step is to explore the ability to interactively change the type and detail of the performance information streamed to the remote visualization client.

With this ability, an analyst could ask the framework to present new information like the number of messages sent, instead of event computation time, per time interval. The analyst could do this over pre-defined phases of the computation, capturing many different sets of performance metrics over equivalent phases of a steady-state computation without sacrificing streaming bandwidth. This form of time-multiplexing is similar to the approach taken by the PAPI performance counter framework [10] to capture counter metrics that otherwise conflict and cannot be captured at exactly the same time by hardware design.

With the interactive functionality, the analyst can also decide if more detailed information is useful at some point in the application's execution. The analyst may be aware of a period of poor performance late into the application's steady-state he wishes to study in detail as it occurs. Also an unexpected dip in application performance could be observed while low resolution data is being streamed, prompting a desire to observe greater levels of detail in the performance stream.

Chapter 6

What-If Analysis Through Simulation

As discussed in section 1.2, the turn-around time required for studying applications at large processor core counts in the performance analysis process can be an impediment to effective problem solving. This can be particularly true for finding out the performance characteristics of an application when scaled to make use of an entire parallel machine system.

Reserved access to a full machine system can sometimes be arranged with machine administrators depending on the schedule. This mode of conducting performance analysis can be slow due to the tendency of the analysis process to be iterative. First, there are the time requirements for identifying possible performance problems. Then, in order to test if a solution to the identified problem works, the application would need to be executed again at the same scale. This means that if the preceeding two steps cannot be accomplished before the reservation time expires, the analyst would have to go through the administrative and scheduling process of reserving the machine. This could, of course, be mitigated by always reserving a second full machine access session at some reasonable point in time after the initial reservation. This second session can then provide the consolidated time to allow the analyst to validate multiple solution hypotheses, assuming the identified problems have non-trival solution spaces. Of course, if the second session fails to identify a good enough solution, the process must then be repeated.

If full machine reservations are not readily available as an option to an analyst, the only remaining option will be to attempt regular submissions to the machine's job queue. These submissions suffer from irregular wait times for full-machine or large-scale requests. Such

requests may involve high turn-around times of up to a week. Finally, even if turn-around time is reasonable, one needs to consider the additional computational resource costs of conducting performance benchmarking runs to test each hypothesis. Simply running a three-minute job on 20,000 processors will cost 1,000 CPU-hours of precious compute-time. For the study of the impact on application performance due to parametric trends (see the latency tolerance case study described in section 6.5), this resource consumption can add up very quickly.

In this chapter, we present a third alternative for conducting performance problem solving for applications at large processor core counts. This alternative avoids or mitigates the need for large-scale access to machines. The key principle of this approach is to conduct a large part of the performance analysis process on fewer processors than are normally necessary. We begin with a discussion of the related literature followed by a description of our simulation framework called BigSim. We explore BigSim's capabilities and the various forms of performance hypotheses testing we have developed either by exploiting or by extending those capabilities.

6.1 Related Work

There are a number of projects that aim to predict application performance at various levels of detail by employing trace-driven simulation like we do.

FASE [32] is a highly general prediction framework designed to support a wide variety of tools and simulation systems. An analyst would first use various tools to generate traces, profiles or even analytical models that could characterize an application. This characterization may require several iterative cycles but is eventually used by FASE as a "stimulus" input to a simulation module that also has to be modelled or constructed by the analyst for the target machine and configurations in mind. FASE works, in a number of ways, very similarly to the BigSim emulation and simulation framework (see Section 6.2). FASE can even be used at simulation time to generate performance event traces similar to the work described in this thesis for use with performance tools like Jumpshot. Its goals and capabilities are, however, differ-

ent. FASE is designed to provide predictions about an application when executed in different modelled hardware environments. Our work focuses on permitting effective performance hypothesis testing at the hardware (see Section 6.5) as well as the application level (see Section 6.6). A re-characterization of the application is required in FASE whenever a different performance hypothesis is to be applied. In contrast, under certain assumptions, our approach requires the application to be executed or emulated only once per base experiment.

Speedy [68, 67] was an old prototype termed as an "extrapolation tool" that share many similar characteristics with our work. Performance event traces are generated by Speedy's trace-driven simulation that can be used for detailed analysis like in the case of FASE and our work through BigSim. Their approach for generating the traces for simulation purposes are, however, somewhat more restrictive. Speedy does this by running an n-thread application on a single processor and abstracting the communication and computation properties for simulation purposes. When simulated, this produces a prediction of the application's performance on n processors given a model for the hardware execution environment for those n processors. Like FASE, it does not provide a more general approach to performance hypothesis testing.

Using the same ideas as Speedy, VPPB [9] executes a multithreaded program on a monitored uni-processor to record execution behavior. With the recorded information, VPPB then uses a simulator to predict the detailed performance of the multithreaded program on any number of actual processors.

Two important simulators of parallel systems are Dimemas [30] and PSINS [74]. Both simulators are driven by MPI traces obtained on a base system. The prediction involves constructing the application behavior on a different system of the same size, using the MPI traces and a scaling factor between machines for the sequential parts of the application. Because both systems depend on the availability of the traces at the same size, it becomes infeasible for these simulators to predict behavior on large machine configurations with more processors than currently available. Meanwhile, using our BigSim framework and its virtualization capability inherited from CHARM++, traces can be obtained for any number of target processors through

emulation.

Various studies have been conducted to predict performance of parallel applications using other approaches. Some of those studies [45, 34] employ analytical modeling techniques to characterize the behavior of an application on a certain system. By changing parameters in the models, one can predict performance for different program/machine combinations, similarly to our what-if functionality. However, building such an analytical model, in general, requires deep knowledge of the application and of the underlying system. In contrast, our simulation-based approach can capture application behavior via emulation, requiring little knowledge about the application.

6.2 The BigSim Simulation Framework

The BigSim [88, 89, 86] framework is comprised of two components: a generic emulator of applications on peta-FLOPS architectures, and a simulator that allows for prediction and analysis of the emulated program in a variety of situations, including in the presence of a detailed network contention model.

The components described above work together as shown in Figure 6.1. The application code is run on the emulator to produce the BigSim event dependency logs. These logs are then passed to the simulator, which can simulate the application with a latency-based messaging model, or a detailed network contention modeling. The simulator can also alter the execution times of each event according to some predictive models. The simulator can report basic performance predictions about execution time, or generate Projections event traces which can be used for detailed analysis of the predicted performance based on the simulated environment.

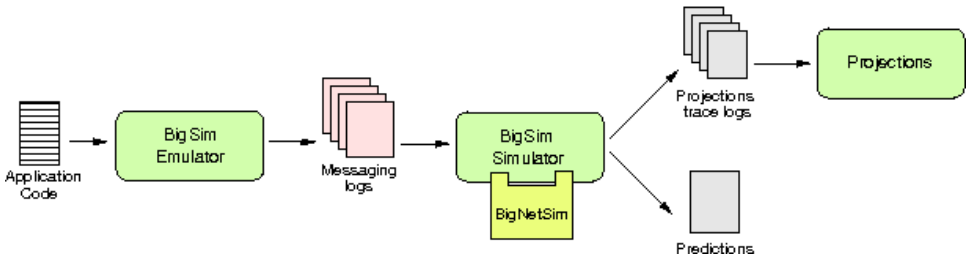


Figure 6.1: BigSim Structure and Output.

6.2.1 BigSim Emulator

The emulator first provides an environment in which users can run a parallel application as if they are running on the real target machine with its full set of processors. It achieves this by leveraging object-based virtualization provided by CHARM++. Suppose there are a million target processors for a normal MPI application. The emulator could start a million AMPI user-level threads on 10,000 available processors. Although this means 100 threads per processor, AMPI has been shown to correctly handle a large number of threads per processor [87]. The emulator produces event dependency traces at the granularity of individual messages and the computation triggered by these messages. These traces are subsequently used for simulation-based performance prediction.

The BigSim emulator also captures the dependencies between the *dependent execution blocks* (DEBs) of an application. Each DEB represents a segment of sequential execution of a thread or an entry method that executes without blocking for remote data. It is triggered by the receipt of one or more messages and the completion of 0 or more other DEBs. For example, in normal MPI use with blocking receives, each DEB depends on 1 message and 1 DEB. With irecv/waitall, it depends on many messages and one DEB. With CHARM++'s Structured Dagger [39], each DEB may depend on multiple DEBs and messages.

The time interval of a DEB is divided into *sequential execution blocks* (SEBs) by each message-spawn event. These SEBs can be further sub-divided by function call boundaries in cases where it is easy to trace and predict individual function call times.

The emulator stores DEB data in its event dependency logs. Figure 6.2 illustrates the structure of a DEB and the data stored for each one. Each DEB is assigned a unique global ID. The BigSim emulator logs which entity (such as an MPI thread, a virtual processor or a chare) corresponds to each DEB, the time duration of the DEB, a predecessor list (DEBs and messages) and a successor list (DEBs). In addition, it records a set of messages spawned by the DEB along with the time at which they were spawned, specified as an offset in time from the start of the DEB. Symbol table information such as entry method names, message tags, etc. are also stored for use with Projections. Each emulating processor produces a log file containing the DEBs for all entities emulated on that processor.

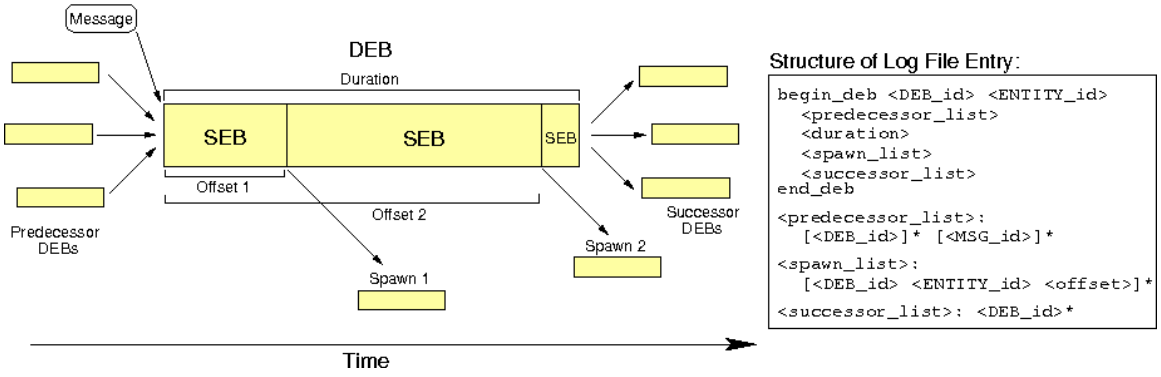


Figure 6.2: Dependent Execution Block.

It has been shown that for realistic models of computations (in either MPI/AMPI or CHARM++), this information is sufficient for the reconstruction of execution behavior even if messages are delivered in a different order [88].

6.2.2 BigSim Simulator

The BigSim simulator takes as its input the event dependency logs generated by the emulation of an application as described above. The simulator re-executes the communication and dependency behavior of the application according to a particular architectural specification, without re-executing the code of each dependent execution block. The simulator can simply scale the execution time recorded for each DEB according to speed ratios between the emulating and

target processors. It can also apply more sophisticated techniques based on target hardware architectures and additional hardware counter data captured at emulation time. This results in new absolute timings for all the DEBs and a new set of traces to be analyzed with Projections. The BigSim simulator is implemented in POSE [78], an optimistically-synchronized PDES environment.

The BigSim simulator is designed to be modular to allow for flexible specification of target architectures. For example, one could simply specify a latency for each message as a function of its size, run the simulator with that latency, and acquire a new set of trace files for Projections analysis. On the other end of the spectrum, we can plug in detailed models of network contention that are very specific to a target architecture via a module called BigNetSim [14].

BigNetSim is a highly flexible network simulator designed to take DEB data from the emulation traces of a real application and simulates the re-execution of the DEBs. BigNetSim preserves dependencies while modeling the transmission of messages for spawned DEBs at the packet-level as they are transmitted through an interconnection network with a detailed contention model. It simulates computational nodes and networks at the level of the switches, channels and NICs of which they are comprised. A wide variety of network configurations have been developed that can be plugged into this simulator. Various topologies and routing strategies exist as well as several other lower-level configurations. The BigSim simulator generates the receive times of messages and start times of all the DEBs as it receives the complete messages from BigNetSim, and generates a new set of predicted execution data that can be analyzed with Projections. BigNetSim is also capable of generating traffic patterns to test a network's performance.

Event trace logs from the simulator can be used with the Projections performance visualization and analysis tool to predict performance problems of an application running on the target machine.

6.3 Performance Analysis Using Fewer Processors

The BigSim simulation framework (see section 6.2) provides the necessary foundation for adopting the above approach. In its *emulation mode*, BigSim exploits the object-oriented, message-driven properties of a CHARM++ application to produce a set of *event dependency logs* that capture the runtime dependency structure and execution profile of a single execution instance of an application. These event dependency logs are then used to predict the application's performance and behavior with different runtime characteristics. The BigSim simulator was designed to handle changes in communication properties like network latency, as well as changes in computation properties like different processor clock speeds. These changes can be simple, like the addition of a fixed latency cost to all communication, or complex, like the determination of latency cost based on detailed networking hardware properties and topology. Finally, Projections-compatible event trace logs can be generated by the BigSim framework at simulation time, allowing an analyst to study the predicted performance visually using the same visualization tool used for regular CHARM++ application analysis.

The key capability we wish to develop and exploit via BigSim is its ability to produce a prediction about an application's performance on P processors using far fewer processors. This means we would not require unplanned and repeated access to the whole supercomputer to conduct performance optimization for the application at scale.

BigSim provides this capability at two levels. First, BigSim is capable of virtualizing processors at emulation time to emulate more than one application process on a physical processor. It does so by storing relative timestamps and dependencies for events and activities that occur instead of their absolute values. Second, at simulation time, the time and memory required to simulate the application's behavior using the captured event dependency data are disjoint from, and much smaller than, the time and memory requirements of the application itself. The former requirements are instead dependent on the total number of events and dependencies generated by the application.

6.3.1 Reducing Number Of Procesors Used At Emulation Time

The reduction of processor counts at the first level is strongly dependent on the nature of the application being emulated. The main limiting factor is memory. If physical memory runs out while the application is being emulated and the BigSim emulator is forced to perform page swaps, the timing information recorded for the current event or activity being emulated will be far from accurate. This cannot currently be corrected for. The other limiting factor is the increase in computation time as we emulate more application processes on a physical processor. If C is the computation time for the application on P processors without emulation, then the computation time is approximately increased by $E \times C + O$ where E is the number of processes emulated on each physical processor and O is the extra event scheduling and communication overhead. There are trade-offs to be considered. On the one hand, we consider the time taken to schedule a job for $(P \div E)$ processors plus the extra computation time taken to emulate E on each processor. On the other, we have to consider the wait time required for scheduling the job for P processors plus its computation time C. For most research scientists with limited priviledged access to supercomputing resources, we argue that the wait time and effort required for scheduling jobs at scale is the dominating factor.

The limiting factor for memory can be mitigated by the use of *memory aliasing* [1] where applicable. A memory aliasing API has been developed so that application programmers can mark memory segments used by the application as shared only for emulation purposes. With memory aliasing, multiple application processes share the same physical memory. The application programmer must mark memory segments such that shared access to those segments will not change the runtime dependency structure of the application, even though the computational correctness will obviously be affected. In the extreme case, if all memory used by an application could be aliased without affecting the runtime dependency structure, then one only needs to consider the trade-off factors for increased computation as described above.

[1] Developed for use in BigSim by Phil Miller.

As an example of the reduction factor in processor counts, we were able to effectively emulate the load imbalance benchmark (see section 6.6.1) at 256 processors using only 2 processors and still produce correct performance predictions and visually validate the performance behavior with Projections. The load imbalance benchmark only uses 48 kilobytes of user memory per processor at execution or emulation time.

6.3.2 Number Of Processors Used At Simulation Time

The memory and time requirements at simulation time differ greatly from the requirements at application execution or emulation time. This is because at simulation time, any actual computation, communication and memory access have become abstract. Memory requirements at simulation time depend on the number of outstanding events that will be scheduled for processing and any book-keeping requirements for parallel discrete event simulators. Similarly, the amount of computation required becomes dependent on the type of work a simulator have to perform in order to process each event as they are scheduled.

More importantly, for the purposes of performance analysis of the application via our approach, the performance of the simulator itself is now divorced from the predicted performance of the application. In the regular performance analysis process, any testing of a hypothesis solution to a performance problem requires the exact performance conditions on actual machine hardware. The simulator, on the other hand, will produce the same prediction results about the application's performance regardless of how the simulator itself performs. This means not only can the simulator be executed on fewer numbers of processors, they can also be executed on local laboratory-based clusters and be subject to local resource limitations rather than centrally managed supercomputing allocations. Predictions can even be made for non-existent machines. Predicted and locally-generated performance data has an additional advantage in that there is no need to transfer large volumes of data across the wide area network that large-scale applications inevitably generate.

All the simulations of benchmarks we have run to demonstrate performance analysis capabilities for this thesis were executed on a single processor core.

6.4 General Methodology For Hypothesis Testing

For the purposes of our subsequent discussion, we define a *baseline experiment* to be the execution or emulation of an application at some runtime configuration that necessitates the generation of a set of BigSim event dependency logs. Baseline experiments are conducted exactly once. In cases where we wish to study changes in application performance properties that cannot be modified by the simulator, specifically in cases where the object-decomposition of the application changes, we will have to make multiple baseline experiment runs. An example of this is the change in object grainsize described in section 6.5.

For each baseline experiment, we apply specific performance property changes through the simulator and generate Projections-compatible performance event logs. The analyst can then visualize and analyze these event traces as if the experiments were executed with those changes.

Sections 6.5 and 6.6 describe case studies that highlight the different performance hypothesis testing capabilities enabled by our approach. The first highlights a class of questions we can ask about the impact on an application's performance when the underlying hardware environment is changed. While the case study explores variation in latency, the idea can easily be extended to other concepts like processor power. The second highlights a class of questions we can ask about the impact on an application's performance when the application's software or code structure changes.

6.5 Analyzing Changes To Network Latency

For the analysis of the sensitivity of an application to communication latencies, it is useful to understand how latency-tolerant an application is. With adaptive overlap (See section 2.1.2),

any available work on other parallel objects can be scheduled by the runtime if an incoming message for one object is delayed. Latency tolerance can be observed by how an application's performance is affected in the face of delayed messages, either due to network contention issues or if the application is executed in a machine environment with a slower communication fabric.

As described in section 6.2.2, BigSim's simulation capabilities can be directly utilized to study the impact on predicted performance by varying communication latency. This is an example of the classic use of application simulation for detailed performance prediction on different hardware environments. We begin by executing the application on the real machine, capturing the application's BigSim event dependency log. At simulation time, the BigSim framework essentially modifies the time taken for a non-local message to be delivered to and scheduled on the target processor. There are two ways of doing this in BigSim. The first way is to simulate the application using the simple latency model where a fixed user-specified latency time is added to message delivery time whenever the message is sent to a remote processor. The second requires a detailed network contention model which takes the modeled network parameters into account when determining simulated latency.

6.5.1 Case Study: Seven-point Stencil Computation

A seven-point stencil code takes as input a three dimensional data array of values. At each iteration, each element of this array updates itself with the result of averaging the values of its six neighbors and itself. The iterative step for a single data element is illustrated in figure 6.3. The computation is normally stopped on convergence, which is usually defined to be the iteration when the difference between the values of the array at current iteration compared with the previous iteration is within a small, pre-determined bound over all data elements. For benchmarking purposes like ours, it is often more convenient to halt the computation at some pre-determined iteration step.

The parallelization strategy employed in our CHARM++ implementation of the seven-point

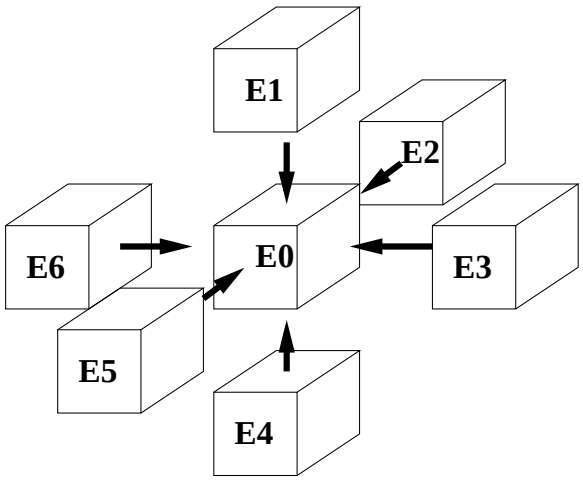


Figure 6.3: Computation interaction between data elements at each seven-point stencil iteration. At iteration $i + 1$, element $E0$ gets the value of $(E0 + E1 + E2 + E3 + E4 + E5 + E6) \div 7$.

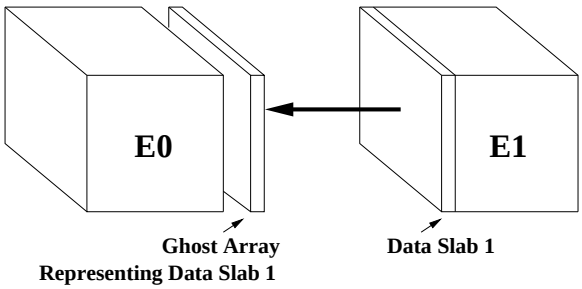


Figure 6.4: Parallel object E1 sends its data slab to object E0 to be kept as a ghost array for the iteration's computation.

stencil computation divides the data array into equally sized three dimensional *data blocks*. The dimensions for each of these blocks are determined by the user at the command line. With that, we create an array of CHARM++ parallel objects, one for each data block. We can now loosely define the *virtualization factor* to be the average number of parallel objects assigned to each processor. The virtualization factor enables an additional dimension for performance tuning via the principle of adaptive overlap (see Section 2.1.2) through object over-decomposition of the problem domain.

Communication between parallel objects under this parallelization strategy is staged at each iteration by the maintenance of two-dimensional "ghost" arrays on each parallel object. Each parallel object O keeps up to six "ghost" data arrays. Each of these arrays represents a one-

element deep slice of the neighboring data array which has a face in contact with the data array hosted by O. At the start of each iteration, each parallel object sends the current data values of the appropriate slices of its data blocks to its appropriate neighbors. These data slices are, as described above, hosted as the appropriate "ghost" arrays in neighboring parallel objects. Figure 6.4 illustrates a parallel object sending a data slice to a neighboring object. Computation for the iteration on each parallel object begins when it has received the "ghost" arrays from every one of its neighbors.

6.5.2 Validation With Seven-point Stencil Computation

Experiment Methodology

The goal in this section is to demonstrate, at a small scale, how we were able to quickly expose latency-tolerance trends for the 7-point stencil computation by changing latency values with many simulation experiments. We also show how detailed performance data based on the simulated application execution became available for visualization and how these were subsequently used to expose unanticipated performance problems at the time these experiments were conducted.

We observe latency tolerance trends for the 7-point stencil computation by varying the following parameters for the application for our baseline experiments:

1. The total problem size in the form of a x_{total} by y_{total} by z_{total} 3-dimensional array.

2. The amount of work assigned to each CHARM++ parallel object in the form of a x_{obj} by y_{obj} by z_{obj} block of the total array. Given a fixed processor count, this affects the virtualization factor described in section 6.5.1.

We studied two problem sizes, a small problem with a 64x64x48 data grid and a large problem with a 256x256x192 data grid. The small problem allowed us to observe the impact of latency changes to performance at small activity grainsizes. The large problem allowed

us to observe the same effects at the same virtualization ratios but with much larger activity grainsizes. Our experimental setup for the set of baseline experiments are summarized by table 6.1.

	Virtualization Factor				
	1	2	4	8	16
Small Problem (64x64x48)	16x16x16	16x8x16	16x8x8	8x8x8	
Large Problem (256x256x192)	64x64x64	64x32x64	64x32x32	32x32x32	32x16x32

Table 6.1: Experimental setup for baseline latency tolerance experiments executed on 48 processors.

Our data points for baseline experiments are gathered in two phases. The first involves data that cannot be modified at simulation time without changing the code's runtime dependencies. Changing the virtualization factor as described above also modifies the shape of the data array in each object and number of communicating objects. As a result, we needed to generate a BigSim event log for each data point. These logs were generated from jobs submitted for 48 processors of a Cray XT5 machine (Kraken). A total of 7 jobs were submitted for this purpose.

The second phase involved using the BigSim framework to simulate the application with varying latency values using the simple latency model. Latency was varied from 1 microsecond to 500 microseconds for the 64x64x48 experiments and from 1 microsecond to 10 milliseconds for the 256x256x192 experiments. These simulation experiments using BigSim were conducted on a single processor of a MacBook Pro, representing a low-end analysis workstation. A total of 108 Projections event trace datasets were collected to expose latency tolerance trends for the 64x64x48 case while 105 datasets were collected for the 256x256x192 case.

Observations and Results

Figures 6.5 and 6.6 show how the average iteration time is affected by increases in latency. The average iteration time is derived from the 10^{th} to 20^{th} iterations of a 25-iteration execution. There are no global barriers between each iteration.

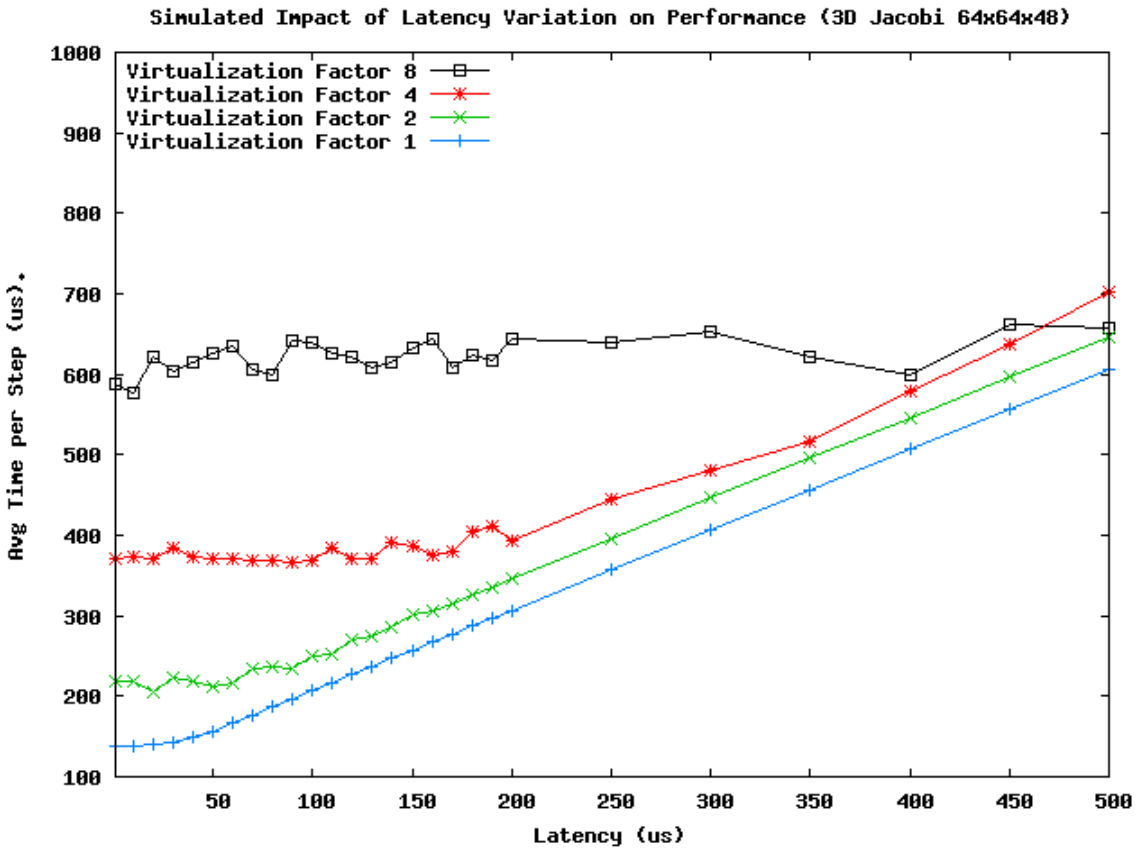


Figure 6.5: BigSim-predicted latency tolerance trends for a 64x64x48 7-point stencil computation code.

In figure 6.6 showing trends for large grainsize experiments, we can see latency hits early at virtualization factors 1 and 2. As we increase the virtualization factors, the application's performance stays reasonably unaffected at higher latencies. At virtualization factor 4, iteration time ramps up from latencies above 1.5 ms. At factor 8, the application tolerates up to 2 ms latencies and at factor 16, latency tolerance goes up to 5 ms. We see a similar trend for small grainsize experiments.

At the same time, we can see the overheads grow as we increase the virtualization factors. These can be expected to come from various sources:

1. scheduling overhead as a result of more, but finer grained, objects.

2. communication overhead as a result of more, but smaller, messages.

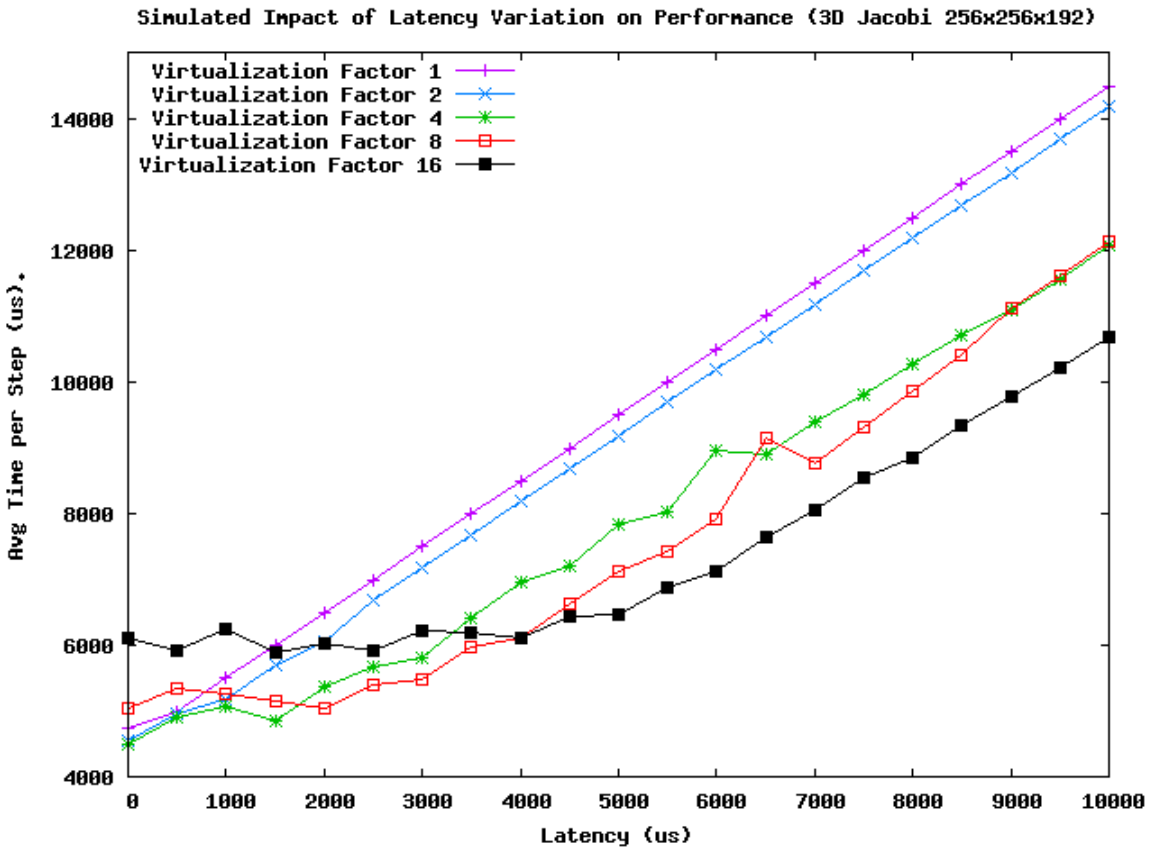


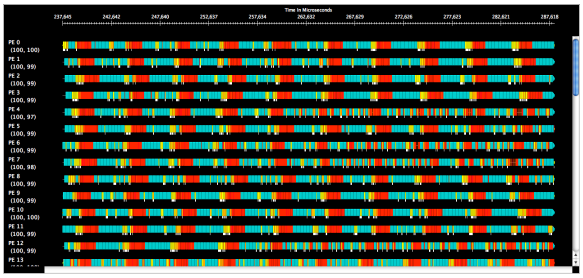
Figure 6.6: BigSim-predicted latency tolerance trends for the 256x256x192 7-point stencil computation code.

3. computation overhead.

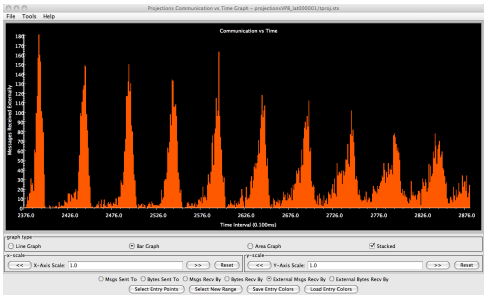
What figures 6.5 and 6.6 do not show us are the precise reasons for the trends observed. In particular, one may ask how latency is being tolerated and why, in spite the overheads, the large grainsize experiment at virtualization factor 16 outperforms the one at virtualization factor 8 at the latency of 5 ms. One may also ask if the magnitudes of the overheads make sense. For example, in the small grainsize experiments, the average time per iteration take relatively huge hits as the virtualization factors are increased from 1 to 8.

This is where BigSim's ability to generate Projections event traces, at simulation time, comes in. Using the trends shown in figures 6.5 and 6.6 as a guideline, we picked several key data points to be simulated again, but this time with Projections logs produced. Note that

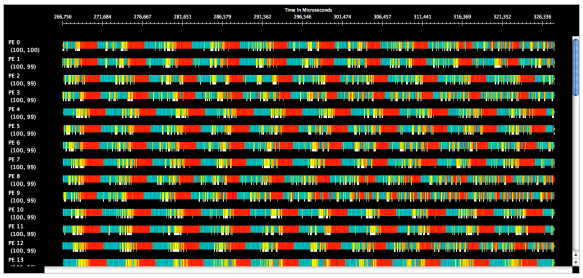
no new runs on the parallel machine were needed.



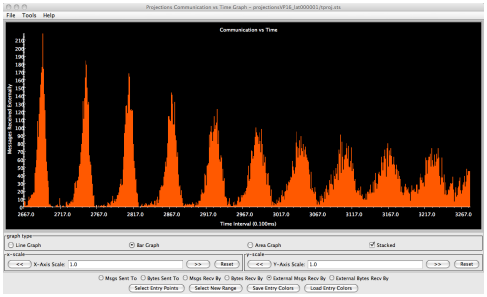
(a) Timeline showing events and activities over 10 iterations at virtualization factor 8.



(b) Number of external messages received over 10 iterations at virtualization factor 8.



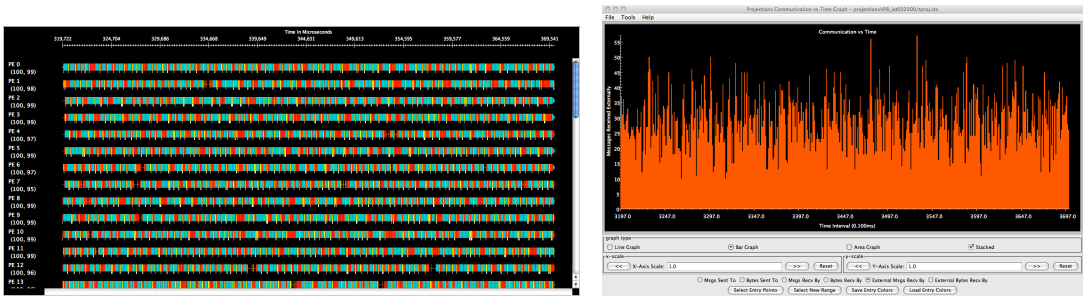
(c) Timeline showing events and activities over 10 iterations at virtualization factor 16.



(d) Number of external messages received over 10 iterations at virtualization factor 16.

Figure 6.7: Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation with a simulated latency of 1 us.

First, we look at how figure 6.7 compares the detailed visualization of two performance properties between virtualization factors 8 and 16 of the large grainsize experiment at 1 microsecond latencies. This is done over the 10 iterations from which we had derived the average iteration times earlier. The timelines show that the activities in both cases are compact. Red activity represent work performed for ghost-region interaction, teal activities represent computation work performed within an array block and yellow activities represent some miscellaneous work that are required at the start of each iteration. The timelines also show in detail how some processors host objects that operate at the array boundaries where fewer and smaller activities associated with ghost-region communication is performed. Meanwhile, the charts showing external message properties over time indicate that message receipt is tightly bound to iteration boundaries.



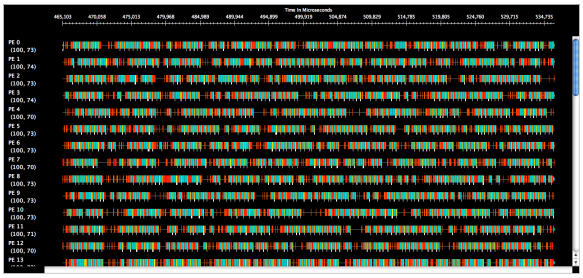
(a) Timeline showing events and activities over 10 iterations.

(b) Number of external messages received over 10 iterations.

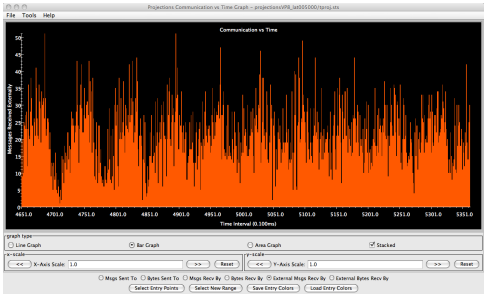
Figure 6.8: Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation at virtualization factor 8 with a simulated latency of 2 ms.

We then look at the details of what happened to the experiment for the virtualization factor of 8 at 2 ms latencies, which is the knee of the curve for virtualization factor 8. The equivalent Projections visualization charts are shown in figure 6.8. The timeline shows that the activities are still compact. However, the nature of the communication structure has changed as adaptive overlap in the CHARM++ runtime kicks in to tolerate the increased latency. Activities are no longer clearly delineated by iteration boundaries. Figure 6.8(b) shows message arrivals becoming spread out.

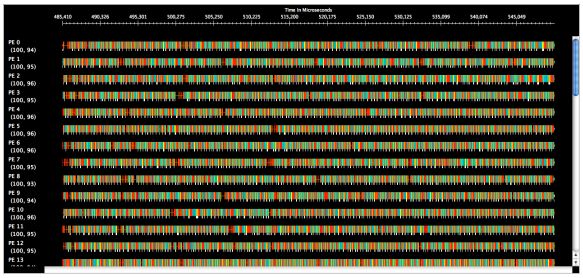
Keeping the above visualization effects in mind, we can now look at what happens at 5 ms latencies which is at the knee of the curve for virtualization factor of 16. At the same time, because the performance of the experiment at virtualization factor 8 is worse than at the same latency for virtualization factor 16, it is now useful to compare the same visualization for a better understanding of the phenomenon. Figure 6.9 illustrates this. At virtualization factor 8, gaps have now shown up on the timeline where the latency increase had crossed the threshold where the application was able to schedule work while waiting for the messages. At the same time, the latencies of 5 ms is now higher than the original average iteration time of about 4.5 ms. As a result, we can see iteration boundaries sharpening again as illustrated by the communication chart 6.9(b). At virtualization factor of 16, however, one sees pretty much the same performance properties as those found at virtualization factor 8 with a latency of 2 ms as illustrated in figures



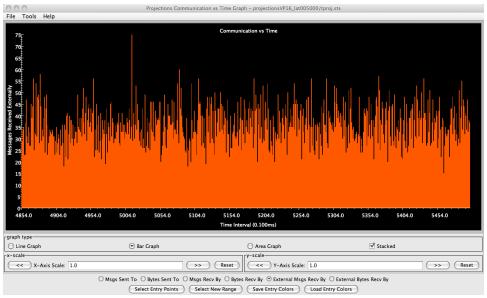
(a) Timeline showing events and activities over 10 iterations at virtualization factor 8.



(b) Number of external messages received over 10 iterations at virtualization factor 8.



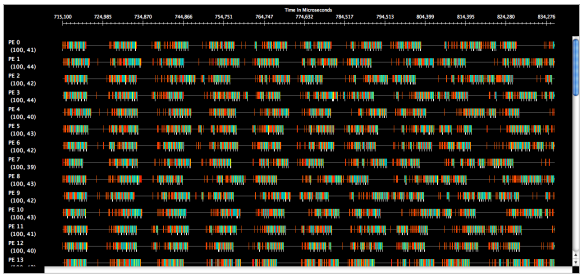
(c) Timeline showing events and activities over 10 iterations at virtualization factor 16.



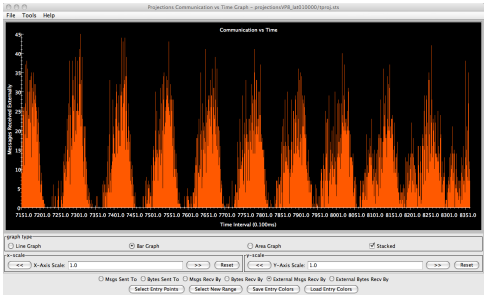
(d) Number of external messages received over 10 iterations at virtualization factor 16.

Figure 6.9: Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation with a simulated latency of 5 ms.

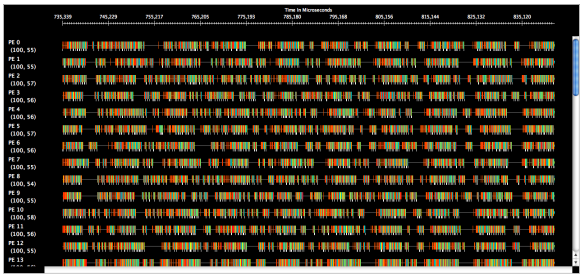
6.8(a) and 6.8(b).



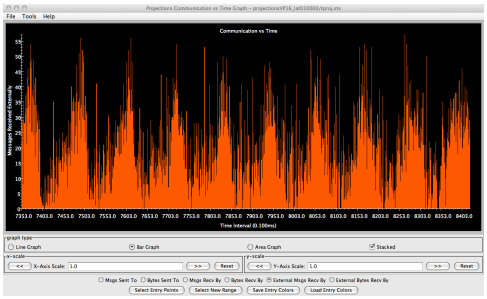
(a) Timeline showing events and activities over 10 iterations at virtualization factor 8.



(b) Number of external messages received over 10 iterations at virtualization factor 8.



(c) Timeline showing events and activities over 10 iterations at virtualization factor 16.

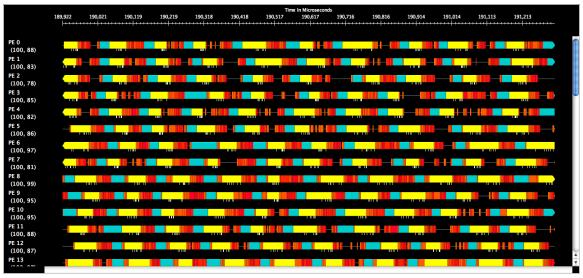


(d) Number of external messages received over 10 iterations at virtualization factor 16.

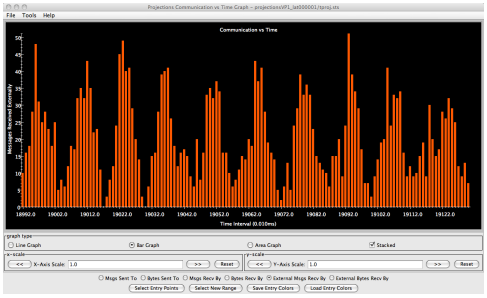
Figure 6.10: Visualization of timeline and communication behavior for the 256x256x192 7-point stencil computation at with a simulated latency of 10 ms.

To complete the picture for the large grainsize experiments, we look at the Projections performance charts at 10 ms latency for both virtualization factor 8 and virtualization factor 16. These are shown in figure 6.10. At virtualization factor 8, it is clear that latencies of 10 ms have completely dominated the performance profile of the application. Iteration boundaries are sharp and punctuated by long time periods where nothing happens. A similar pattern can now be seen at virtualization factor 16, although less stark.

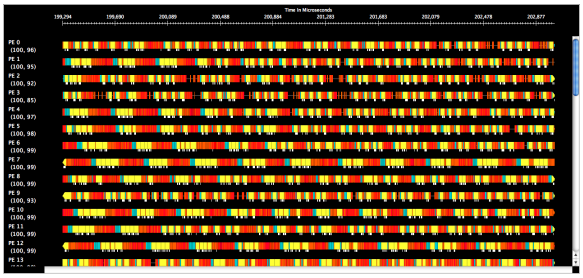
Finally, we conducted the same visualization analysis for the small grainsize experiments, to try to observe the same effects of latency on performance. At the same time, we wanted to figure out if we could see in detail what could have caused overhead to be so high for these experiments. Figures 6.11 and 6.12 demonstrate that the latency tolerance properties of the 7-point stencil computation for large grainsize is preserved for the small grainsize experiments.



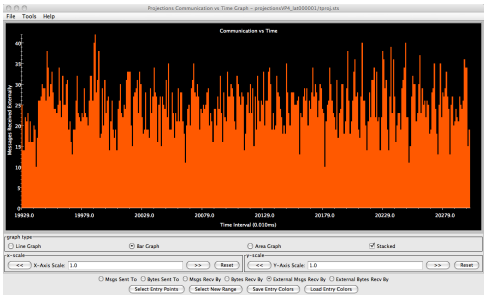
(a) Timeline showing events and activities over 10 iterations at virtualization factor 1.



(b) Number of external messages received over 10 iterations at virtualization factor 1.

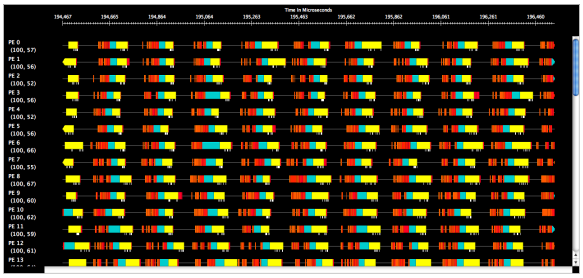


(c) Timeline showing events and activities over 10 iterations at virtualization factor 4.

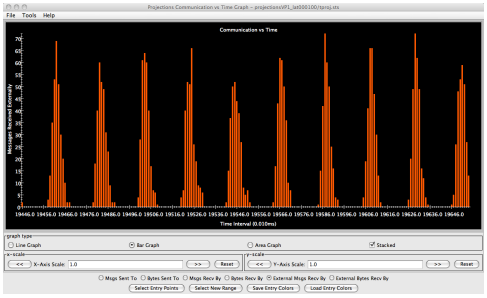


(d) Number of external messages received over 10 iterations at virtualization factor 4.

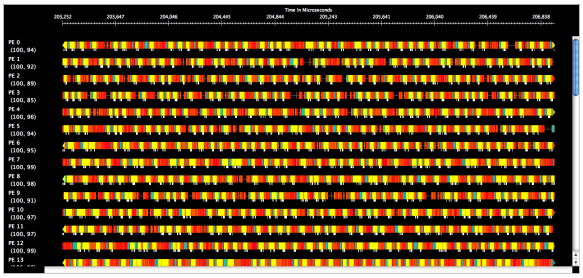
Figure 6.11: Visualization of timeline and communication behavior for the 64x64x48 7-point stencil computation at a latency of 1 us.



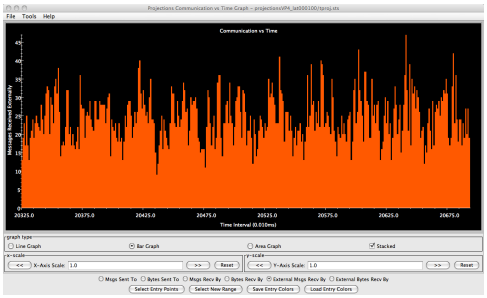
(a) Timeline showing events and activities over 10 iterations at virtualization factor 1.



(b) Number of external messages received over 10 iterations at virtualization factor 1.



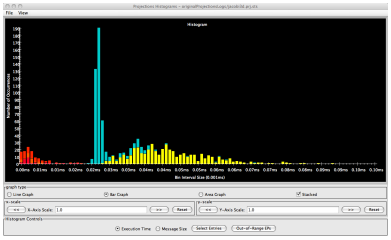
(c) Timeline showing events and activities over 10 iterations at virtualization factor 4.



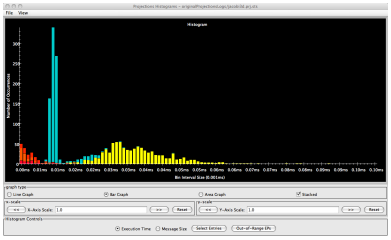
(d) Number of external messages received over 10 iterations at virtualization factor 4.

Figure 6.12: Visualization of timeline and communication behavior for the 64x64x48 7-point stencil computation at a latency of 100 us.

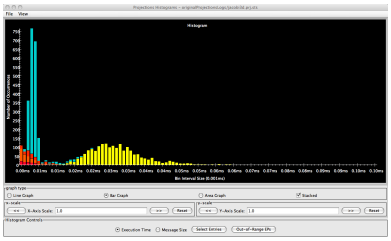
The figures also reveal the possible main source of overhead when attempting to increase the virtualization factor at small grainsizes. We noticed that the actual work done in an array block, as represented by teal-colored activities, seemed to become dominated by miscellaneous work done at the start of each iteration, represented by the yellow-colored activities.



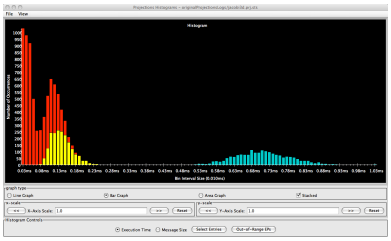
(a) Histogram of activity counts over 10 iterations for 64x64x48 7-point stencil computation at virtualization factor 1.



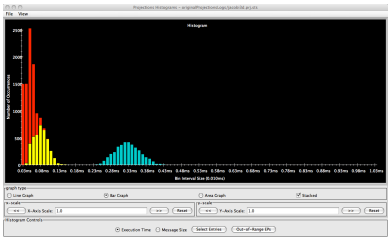
(b) Histogram of activity counts over 10 iterations for 64x64x48 7-point stencil computation at virtualization factor 2.



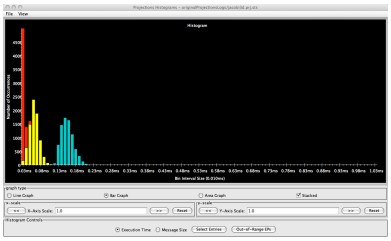
(c) Histogram of activity counts over 10 iterations for 64x64x48 7-point stencil computation at virtualization factor 4.



(d) Histogram of activity counts over 10 iterations for 256x256x192 7-point stencil computation at virtualization factor 4.



(e) Histogram of activity counts over 10 iterations for 256x256x192 7-point stencil computation at virtualization factor 8.



(f) Histogram of activity counts over 10 iterations for 256x256x192 7-point stencil computation at virtualization factor 16.

Figure 6.13: Histogram visualization of activity grain sizes for the 64x64x48 and the 256x256x192 7-point stencil computation experiments.

To bring clarity to the issue, we made use of the Histogram view of Projections to see how activity grain sizes change as we increased our virtualization factors. The trends for large grain-

size experiments from virtualization factors 4 to 16 as well as for small grainsize experiments from virtualization factors 1 to 4 are shown in figure 6.13. The bars colored yellow represent activities associated with some work conducted at the start of each iteration. As we can see, the size of the computation overhead for these yellow-colored activities limit any benefit to grainsize reduction for the computation work associated with the array blocks. Any performance improvements to the code as far as grainsize of work is concerned will have to take these activities into account.

Conclusion

This case study clearly demonstrates the wealth of performance information from new event traces that we were able to create through simulation. Using just 7 actual runs on the parallel machine, we were able to predict the performance trends for over 213 different configurations without the need to consume resources on the same parallel machine. Additionally, we were able to selectively produce detailed performance event traces for specific configurations after the trends were revealed. In classical approaches to performance analysis, an analyst would have two choices: generate performance event traces for each of the 213 configurations; or execute the 213 configurations to acquire timing data for the trend and then run the desired configurations again to acquire the detailed event traces. At very large processor scales, this can potentially cost a lot of computing resources.

6.6 Analyzing Hypothetical Load Balancing Changes

Load Balance (see section 2.1.3) in an executing CHARM++ application is achieved by migrating CHARM++ objects from a more heavily loaded processor to a more lightly loaded processor at execution time. The measurement based load balancing framework maintains computation and communication load information involving the application's object graph and passes this information to a chosen load balancing strategy. The strategy module then makes a decision

on object migration. The framework supports a number of pre-written load balancing strategies while also providing an application developer a means to develop their own strategy that is most appropriate for their application.

Figuring out how load balancing strategies impact the performance of a non-trivial application like NAMD is an important aspect of performance analysis work. Load balancing strategies can impact application performance significantly. Kalé et. al. [43] demonstrated how a two-step approach of initially applying a greedy load balancing strategy to NAMD followed by a gradual series of refinement strategies was partially responsible for allowing the simulation to scale to 3,000 processors. Bhatele [5] showed the importance of incorporating physical processor topology information into load balancing strategies for good application performance at large scale. To understand the impact different load balancing strategies have on application performance, one would normally have to re-execute the application in order to observe the changes. Application re-execution at large-scales for testing the impact of different load balancing strategies can be extremely time-consuming, both from the job scheduling perspective and the fact that execution has to reach the appropriate load balancing point at runtime. This could be mitigated if the application's state can be checkpointed to disk just prior to the load balancing decision point. If the object-load information can also be captured and written to disk, it would then be possible to short-cut the process to start execution and record performance information from that checkpoint.

What-if experiments involving the application of different object load balancing strategies require the BigSim simulator to be capable of simulating the execution of an activity associated with a CHARM++ object on a possibly different processor than the one the activity was recorded with. This capability is currently not available on BigSim. As a result, we have developed a tool to manipulate the event dependency logs. The idea behind this tool is to modify the event dependency logs such that event records associated with objects that would be mapped to a different processor by a modified load balancing strategy would be moved from the source processor's event dependency log to the target processor's. This allows the modified logs to be

simulated by BigSim to achieve an effect as if the application were emulated with the modified load balancing strategy.

To effectively generate modified logs as described above requires some automated way of acquiring object-to-processor maps for different load balancing strategies given some fixed object-load information. Manually determining such object-to-processor maps are simply infeasible for applications with very large numbers of migratable objects. Fortunately, per-object load is available from the load balancing framework at any time in the application's execution. When invoking the centralized load balancer, this information can be written to disk. The load balancing framework has the capability of reading per-object load information from the disk and using that, along with the desired number of processors, as input to any load balancing strategy module to produce object-to-processor mapping decisions. This capability is independent of the application that produced the object load information in the first place.

6.6.1 Case Study: Simple Load Imbalance Benchmark

We will refer to this program as *SimpleImbalance*. The purpose of *SimpleImbalance* is to provide a baseline model for validating and quantifying the correctness and accuracy of the techniques described in the thesis. As we will describe later, it has a clearly defined model of execution which facilitates a clear understanding of performance expectations and performance problem-injection.

SimpleImbalance is a CHARM++ application. In its most basic form, at the start of the application, it will place 2 migratable computation objects on each processor. The number of migratable computation objects used can be changed at runtime. The idea is to generate a deliberate load imbalance by initially assigning a single unit of work to each object on half the processors and 2 units of work to each object on the other half of the set of processors. The assumption is *SimpleImbalance* will always be executed on an even number of processors. The computation then proceeds for a user-specified k number of iterations, each ending with

a synchronization barrier. At the $k/2$th iteration, the code an opportunity for the runtime to invoke load balancing based on the load balancing strategy selected at the start of the execution. Figure 6.14 illustrates the basic execution structure of the benchmark as a time line with four migratable objects per processor.

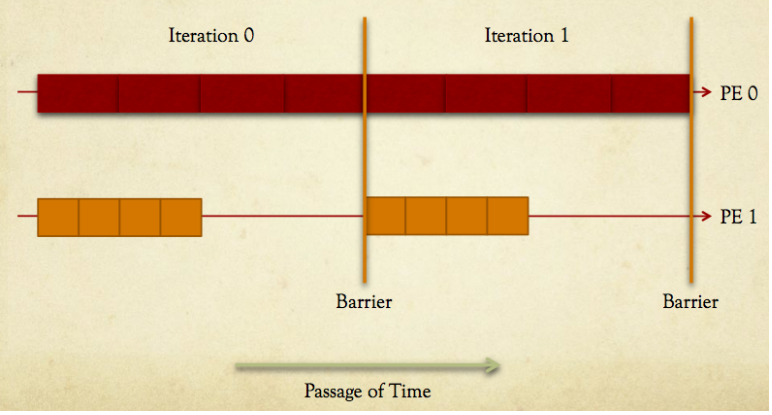


Figure 6.14: Illustration of object placement and execution of the basic *SimpleImbalance* application.

When the application is executed with a "perfect" load balancing strategy like the *GreedyLB* module distributed with the CHARM++ runtime system, the application is expected to demonstrate a performance profile where the last half of the iterations would execute in three quarters the time of the first half of the iterations. The performance profile would also be expected to show much lower idle time as no processor would complete an iteration in half the time and be forced to wait at a barrier for other processors. This profile is illustrated in figure 6.15.

Similarly, when the application is executed with a useless load balancing strategy, one expects a performance profile that is no different from when no load balancing strategy was employed. An example of such a load balancing strategy is simply moving objects from processor P to processor $P+1$ modulo the number of processors. Such a strategy exists in the form of the *RotateLB* module also distrubted with the runtime for test purposes. This serves as validation in the sense that objects can observed as moved by the load balancing strategy but the overall performance effect can be perfectly predicted.

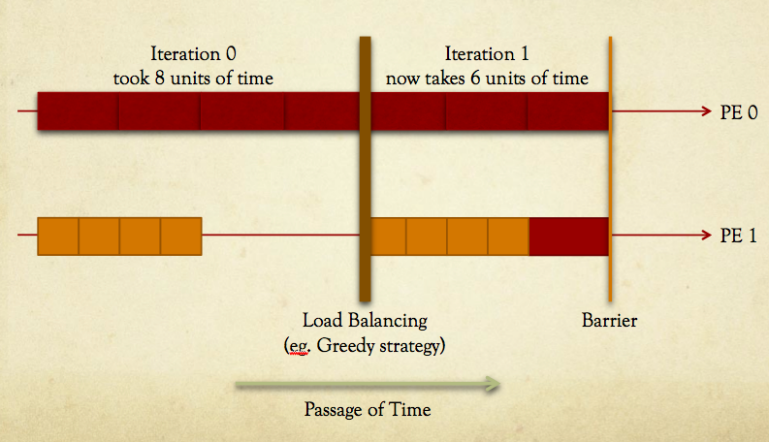


Figure 6.15: Illustration of object placement and execution of the *SimpleImbalance* application with the *GreedyLB* load balancing strategy applied.

6.6.2 Validation With Simple Load Imbalance Benchmark

Methodology

We validated our technique on a small benchmark which shows simple imbalance properties over a number of iterations k with every iteration separated by global barriers (see section 6.6.1). This imbalance property is simply that all processors hosts 2 objects each but the objects on half the processors will be made to perform twice the work at each iteration. This initial assignment of load to objects will not change throughout the benchmark's execution. Load balancing is invoked halfway at iteration $k \div 2$. This baseline experiment helps us validate if the object-to-processor maps produced by the load balancer strategy really does what it was suppose to do. For example, the Greedy load balancing strategy is certain to balance the objects such that the load across all processors become even. Meanwhile, a Rotate load balancing strategy will simply migrate objects to the next processor. Either result can be observed easily from the Projections-compatible event logs generated at simulation time.

Results

Figures 6.16, 6.17 and 6.18 show the results with the baseline experiment emulated for 12 application processors with an emulation factor of one application processor to one physical processor. Figure 6.16 shows the Projections timeline of the baseline experiment, displaying all the performance characteristics described above. Figure 6.17 shows the Projections timeline after applying the object-to-processor map generated by a Greedy load balancing strategy. One clearly sees the intended result of applying the load balancing strategy. What is less clear is that the results are based on real load information at the application's emulation time. While one can, given one's understanding of this benchmark's behavior, simply swap objects between pairs of processors, the final result shown takes into account all minor variations in object load at runtime. Figure 6.18 shows the Projections timeline after applying the object-to-processor map generated by the toy Rotate load balancing strategy.

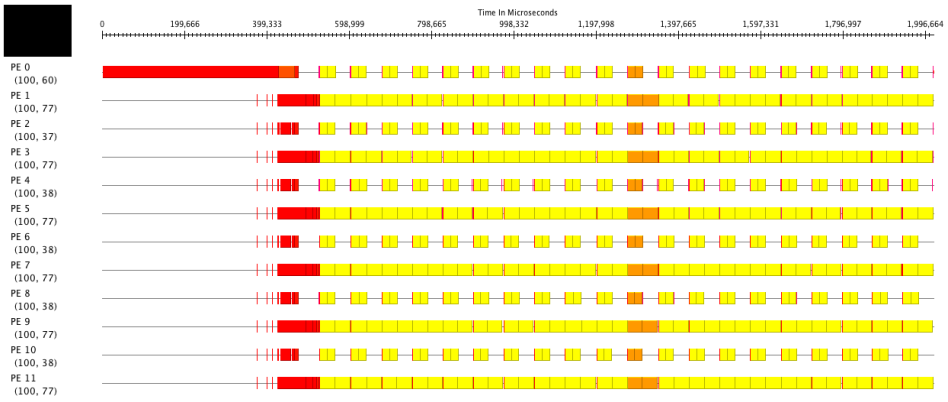


Figure 6.16: Projections timeline showing the baseline performance profile of the benchmark with deliberate load imbalance.

In addition to a visual inspection of the effects of the application of hypothetical load balancing strategies to the benchmark, we have also validated the effects on the measured time of iterations. We conducted this validation using 100 Cray XT5 processors of Kraken. The experiments are all run over 10 iterations with the load balancing strategy invoked just before

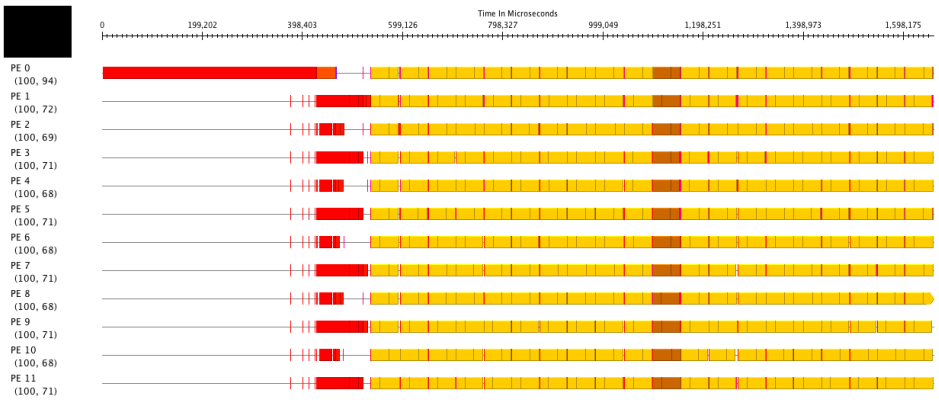


Figure 6.17: Projections timeline showing what happens if the Greedy load balancing strategy is applied.

the 6^{th} iteration.

BigSim event dependency logs are recorded from emulated runs of two baseline benchmark experiments using a dummy load balancing strategy: one with 2 objects per processor and the second with 20 objects per processor. For each of these baseline experiments, we ran the following experiment configurations:

1. Plain CHARM++ execution using the dummy load balancing strategy;

2. BigSim emulation using the greedy load balancing strategy *without recording* BigSim event dependency logs;

3. Plain CHARM++ execution using the greedy load balancing strategy;

4. BigSim simulation with the dummy load balancing strategy using event dependency logs recorded from the baseline experiment; and

5. BigSim simulation with the greedy load balancing strategy using the *re-mapped event dependency logs* derived from the event dependency logs recorded from the baseline experiment.

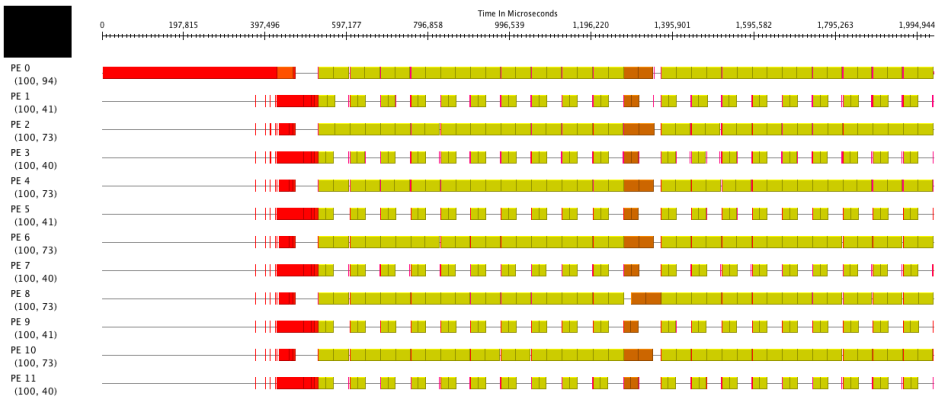


Figure 6.18: Projections timeline showing what happens if the Rotate load balancing strategy is applied.

% Difference from	2 objects per processor	20 objects per processor
BigSim Emulated Dummy Strategy	0.01%	0.00%
CHARM++ **Dummy Strategy**	0.63%	0.88%

Table 6.2: Comparison of the *SimpleImbalance* benchmark average iteration times with a dummy load balancing strategy on 100 processors.

We then compared the average iteration times after load balancing invocation between the simulated and non-simulated runs. We also compared the average iteration times between the BigSim emulated runs against the plain CHARM++ executions. Tables 6.2 and 6.3 summarize the results for the use of a dummy load balancing strategy (no load balancing) and the use of a greedy load balancing strategy respectively. The simulated runs predicted iteration times indistinguishable from the corresponding emulated runs. Meanwhile, the iteration times recorded by the emulated runs are within 1% of the times recorded for plain CHARM++ executions. We

% Difference from	2 objects per processor	20 objects per processor
BigSim Emulated Greedy Strategy	0.06%	-0.37%
CHARM++ **Greedy Strategy**	0.57%	0.63%

Table 6.3: Comparison of the *SimpleImbalance* benchmark average iteration times with a greedy load balancing strategy on 100 processors.

found some small perturbations in the timing of the iteration immediately after load balancing. This may be because of the way the BigSim emulation generates dependency information and will require further investigation.

Conclusions

This example employed a simple and easily verifiable benchmark that demonstrates the correctness of our approach to testing different load balancing strategies for CHARM++ applications through simulation. We were able to visually and numerically confirm the expected effects of the changes through the event traces generated by the BigSim simulator. No analytical modeling of the application itself was required. All dependency and object abstractions were encapsulated by the CHARM++ language and runtime.

6.7 Future Work: Variations In Object-to-Processor Mapping

The technique described in the preceeding section on load balancing strategy changes (see section 6.6) can be extended to apply to events associated with objects that do not necessarily move throughout the application's execution but whose permanent object-to-processor placement plays an important role in overall application performance. Kumar et. al. [49] described just such a scenario in NAMD where significant performance improvement was gained if objects associated with the spatial decomposition component ("patches") in NAMD were placed in a manner that matches the Bluegene/L torus topology as much as possible.

We can also extend the technique to study what happens to an application's performance when it is executed on a different number of processors. There are three possibilities:

1. performance when running on fewer processors. This is useful when an analyst wants to see if code or simulation parameter changes meant to enable better performance for

higher processor counts would negatively impact performance at lower processor counts. For the purposes of these what-if experiments, we assume the object decomposition does not change when the application is run on fewer processors. Based on this assumption, the number of objects in the recorded event dependency logs do not change. As a result, some method of re-mapping them to a lower number of processors is needed. This could be done by extracting a generalization of the algorithm used to initially place objects for the application for a certain number of processors or the objects could be re-mapped by the object-to-processor mapping produced by some load balancing strategy (see section 6.6) using the lower processor count.

2. performance when running on more processors. These experiments would assume the object decomposition will not change for the application when run on more processors. Similar to the case where we study the application's performance on fewer processors, some method for re-mapping the objects in the event dependency logs will have to be used.

3. performance when running on an "infinite" number of processors. This a simple but useful way to study the effects of the main computational critical path given a fixed object decomposition of an application simulation. We then gain the ability to alter both computation and communication parameters in the context of how the CHARM++ runtime schedules events and observe the effects on the critical path. To achieve this effect, each object is mapped on its own dedicated processor by the object-to-processor map.

Chapter 7

Extending Scalability Techniques To Third Party Tools

The scalability techniques presented in previous chapters allow effective performance analysis to be employed for large system configurations, on scenarios that are typical in modern parallel machines. Although they are based on the CHARM++ infrastructure, these techniques can also benefit other tools that are able to import performance data made available by our instrumentation capabilities. In this chapter, we show how the integration with external tools is made possible in our environment, via an open interface in our data collection mechanism [1]. We illustrate that capability with a concrete example, describing the relevant features of an integration between CHARM++ instrumentation and the TAU performance system. This example clearly demonstrates the extensibility of our infrastructure and its applicability to other scalable performance tools.

7.1 Performance Call-back (Event) Interface

How a parallel language system operationalizes events is critical to building an effective performance framework. We use the term *performance event* to represent the execution of instrumentation code associated with any of the above points for the purpose of making a performance measurement. The CHARM++ performance framework implements performance events using a *call-back* mechanism [2], whereby instrumentation in the runtime system's code invokes any

[1] This chapter is written with portions reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at *The 38th International Conference on Parallel Processing (ICPP-2009)*, ©2009 IEEE

[2] Implemented by Gengbin Zheng

performance module (performance client) registered in the framework interested in the event. The framework forwards basic default event information, such as the event ID and time of occurrence, as well as other event-specific information, to clients.

A client, as a performance module, will also have access to internal runtime routines and meta-data not normally available to user code. These runtime routines and meta-data might then be used to derive more pertinent event-related information. Multiple clients can be used on a given execution, according to the selections made in the command line.

The framework exposes the set of key runtime events as a base C++ class (Figure 7.1). A new performance module like TAU (Figure 7.3) could inherit from this base class and implement methods for the interpretation and storage of information derived from individual runtime events (we show a concrete example in the next section). The client module need not handle every runtime event, merely the ones that are of interest to the module. The initialization of the performance framework invokes client module initialization through statically determined methods.

The call-back approach utilized by CHARM++ has several advantages. Foremost, it separates concerns for performance event visibility from performance measurement. The call-back mechanism defines a performance event interface, but does not mandate how measurements are made. The ability to register different modules allows measurements to be configured for the desired performance experiment. It also allows the available measurement capabilities to be extended. Registering multiple modules allows measurement techniques to be simultaneously applied.

It is interesting to note that a new measurement module can introduce additional requirements for event information. A call-back-based performance interface also allows CHARM++ developers to update the call-back implementation and/or runtime "performance support" library, when needed, without affecting the other measurement modules.

```
// Base class of all tracing strategies.
class Trace {
  // creation of message(s)
  virtual void creation(envelope *, int epIdx,
int num=1) {}
  virtual void creationMulticast(envelope *, int epIdx,
 int num=1,
 int *pelist=NULL) {}
  virtual void creationDone(int num=1) {}
  virtual void beginExecute(envelope *) {}
  virtual void beginExecute(CmiObjId *tid) {}
  virtual void beginExecute(
    int event,   // event type defined in trace-common.h
    int msgType, // message type
    int ep,      // Charm++ entry point
    int srcPe,   // Which PE originated the call
    int ml,      // message size
    CmiObjId* idx)    // index
  { }
  virtual void endExecute(void) {}
  virtual void beginIdle(double curWallTime) {}
  virtual void endIdle(double curWallTime) {}
  virtual void beginComputation(void) {}
  virtual void endComputation(void) {}
};
```

Figure 7.1: Simplified fragment of framework base class. *Reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at* The 38^{th} International Conference on Parallel Processing (ICPP-2009)*,* ©*2009 IEEE.*

7.2 Integrating An External Performance Module

To demonstrate our capability of integration to external tools, we present a new instrumentation module created in CHARM++ for the TAU performance system [70]. TAU is an integrated parallel performance system providing support for instrumentation, measurement, analysis, and visualization for scalable parallel applications. The measurement system is cross-platform and provides both profiling and tracing support. In this particular experiment, the interest was in exploring TAU's parallel profiling feature.

The TAU's profiling model is based on the notion that every "thread of execution" in the parallel computation has an *event stack* that records the dynamic nesting of performance events marking the begin/end of interesting execution regions for measurement. Performance data is measured for events to reflect the *exclusive* performance (e.g., time) when the event was active (e.g., time spent in a method). The event stack allows *inclusive* performance data to also be kept (e.g., time spent in a method including time spent in nested method calls). For each event, TAU can collect performance data for execution time, hardware counters, and other metrics.

7.2.1 TAU Profiling Integration

The problem of integrating TAU profiling can be seen as how to map the TAU profiling model onto the existing CHARM++ runtime performance framework. TAU associates for each "thread of execution" an event stack (logically an *event tree*), with a top-level event (root) which encloses all other events. In normal TAU use, the top-level event can be thought of as the *main* routine of the program. In CHARM++, each parallel process executes the scheduler as the top-level routine and the methods are called within this process. Given the performance events in the CHARM++ performance framework, we might then see Figure 7.2 as a logical event transition diagram which could be profiled in TAU.

Following this approach, the TAU performance module was patterned on the Projections module to capture all method events. However, TAU requires a slightly different initialization

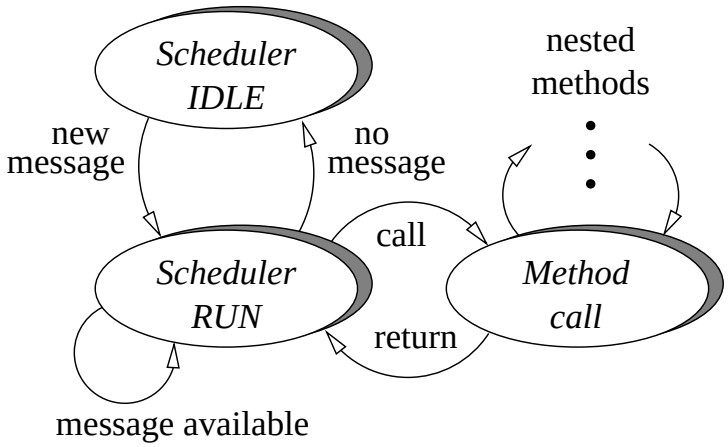


Figure 7.2: CHARM++ runtime event transition. *Reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at* The 38^{th} International Conference on Parallel Processing (ICPP-2009), ©*2009 IEEE.*

to establish the scheduler creation as the top-level event. The state when the scheduler is active and processing messages ("Scheduler RUN" in Figure 7.2) can be associated to the *Main* event in the TAU profile. When no methods are executing and there are no messages to process ("Scheduler IDLE"), performance data is associated with the *Idle* event. Method events appear naturally nested under *Main* in a TAU profile.

The integration to the CHARM++ instrumentation model can be done by using the call-back interface described in the previous section. The new performance module created for TAU follows the scheme depicted in Figure 7.3. As expected, it inherits from the base *Trace* class and provides the additional functionality needed for profiling.

7.2.2 Instrumentation Overhead Assessment

An important concern when using multiple instrumentation modules is the potential impact on application performance. Ideally, the perturbation inserted by each module should be negligible. This effect can be assessed with a simple benchmark program that measures module overhead under different instrumentation conditions. Both CHARM++ and TAU have mechanisms to control the degree of enabled instrumentation and it is important to evaluate how the overhead

```
// TAU implements a performance module class
// inheriting from the base framework class.
class TraceTau : public Trace {
// Statically-determined method which is invoked by the
// framework at runtime initialization.
void _createTraceTau(char **argv)
{
  // TAU initializes the events buffer to hold
//5000 different events ...
  bzero(events, sizeof(void *)*5000);
  // ...

  // TAU creates a new performance module instance and
  // attaches itself to the Charm++ tracing framework
  CkpvInitialize(TraceTau*, _trace);
  CkpvAccess(_trace) = new TraceTau(argv);
  CkpvAccess(_traces)->addTrace(CkpvAccess(_trace));
}

// Performance module constructor.
TraceTau::TraceTau(char **argv)
{
  if (CkpvAccess(traceOnPe) == 0) return;
  // TAU does more initialization, processes
  // commandline arguments ...
    // ...
    TAU_PROFILER_CREATE(main, "Main", "", TAU_DEFAULT);
    TAU_PROFILER_CREATE(idle, "Idle", "", TAU_DEFAULT);
    // ...
}

// TAU interprets Charm++ runtime events
// which are of interest to TAU.
void TraceTau::beginExecute(envelope *e)
{
  // ...
    startEntryEvent(e->getEpIdx());
  // ...
}
```

Figure 7.3: TAU integration with CHARM++ framework. *Reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at* The 38^{th} International Conference on Parallel Processing (ICPP-2009)*,* ©*2009 IEEE.*

No measurement module	
CHARM++ fully optimized	0.09
Null trace module loaded	0.44
TAU module	
with *[notrace] option*	0.55
with selective instrumentation	0.74
with fastest available timers	1.03
with *get_time_of_day() timers*	1.21
Projections module	
with *[notrace] option*	0.49
with fastest available timers	1.99
TAU and Projections modules	
with fastest available timers	2.52

Table 7.1: Overhead of performance modules (microseconds per event). *Reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at* The 38^{th} International Conference on Parallel Processing (ICPP-2009), *©2009 IEEE*.

changes with greater measurement accuracy.

Table 7.1 shows the results from the overhead tests on a Linux x86 cluster, using different time bases and varied instrumentation level. CHARM++ allows the exclusion of entry events from the tracing system by use of the `[notrace]` entry method attribute. TAU's selective instrumentation works by runtime selection. In either case, the performance framework callback routines are executed and they incur a small amount of overhead, as shown by the *Null trace module* for the overhead when tracing framework is active but no events are actually being measured. The table also shows the overhead when both the Projections and TAU modules are enabled, which is accomplished by having each performance module called individually, and thus results a greater amount of overhead. Projections and TAU have comparable overheads and these are relatively small. The availability of event selection options can also have beneficial effects in reducing overhead.

7.2.3 Experimental Results On Current Machines

The newly integrated TAU module was used in a performance analysis of NAMD, as described in [7]. In that study, conducted on two large existing machines and employing two distinct molecular datasets, detailed performance data got collected by both Projections and TAU modules. It was possible to gain insight into application's behavior via the overview, timeline and time-profile views in Projections, as well as to produce profiles in TAU with breakdowns of the execution time across the various code sections.

The study with NAMD also included a quantification of the overhead due to instrumentation. For executions on TACC's Ranger from 64 to 4096 processors, Figure 7.4 shows the impact of the overhead for NAMD using both the TAU module and the Projections + summary module (in each case the Projections trace buffer is set so that no overflow would occur while NAMD was running). At large scale (4096 processors), where the granularity of work becomes small enough, the modules incur a nontrivial amount of overhead. While 10% overhead is within the range of most performance experimentation, further research could target overhead reduction in the context of this framework.

7.3 Scalability Benefits Of Integration

Performance tool integration conducted in the manner described in this Chapter enables various interesting possibilities. The first is the natural extension of any techniques employed by the third-party tool to handle performance data from CHARM++ applications scalably. An example would be the option to use Supermon daemons or an MRNet setup at job scheduling time for scalable delivery of online data through the integrated TAU performance component instead of routing the data through our own adaptive runtime as described in Chapter 5.

The second possibility is the application of any of the scalability techniques described in this thesis to integrated third-party tools. Since the tools have access to the same parallel runtime

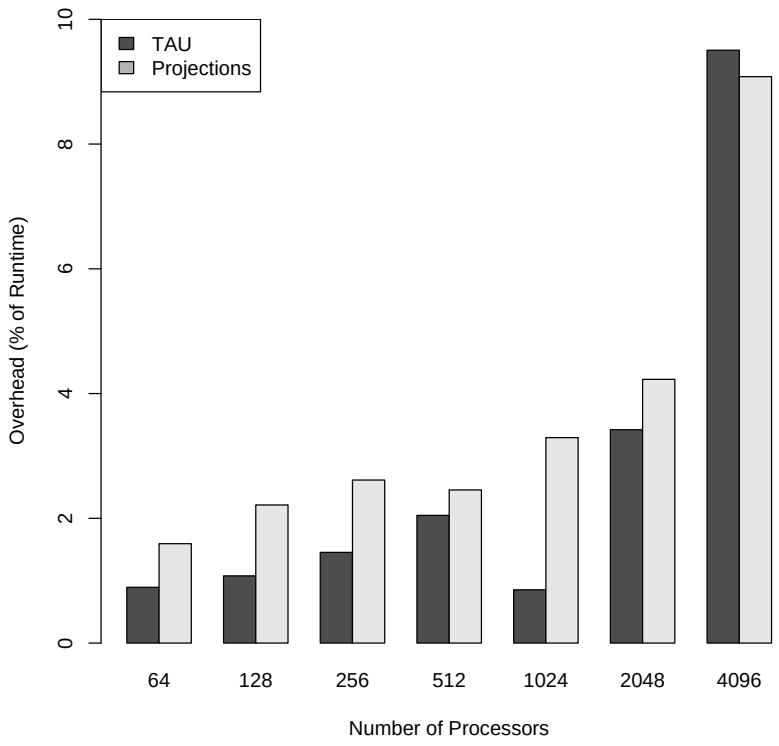


Figure 7.4: Instrumentation overhead for TAU and Projections with NAMD running on Ranger. *Reprinted, with permission, from "Integrated Performance Views in Charm ++: Projections Meets TAU" by Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé at* The 38^{th} International Conference on Parallel Processing (ICPP-2009), ©*2009 IEEE*.

infrastructure, information and data structures, they can choose to implement or use any of the techniques described in previous chapters.

An additional possibility is the creation of a richer performance analysis scenario, comprising the union of the capabilities from the individual modules. In a scenario such as the one in our example with TAU integration, there are many complementary features from Projections and TAU that one can choose to assess application performance. Some of these features may be impossible to explore with a simple transfer of trace files between the different tools after the execution is done (e.g. it would be unfeasible to apply TAU's dynamic event throttling mechanisms without having access to performance data in real time).

Chapter 8

Conclusions

In this thesis, we have studied the various factors that affect the effectiveness of performance tools when used to study applications scaled to large numbers of processors. We proceeded to develop novel techniques to address and overcome some of these factors.

We developed a variety of tools and features to support new idioms for scalable performance analysis. These tools include histograms to support grainsize distribution analysis, time profiles and the Extrema Tool, which supports scalability in many recurring analysis idioms that required analysts to pick extreme or interesting processors for detailed examination.

The principles behind the ideas developed in the Extrema Tool for scalable analysis idioms were then applied to performance data volume reduction. When exercising those scalable analysis idioms that require detailed performance data, the need for high detail involved only a small subset of processors. This subset included the interesting or extreme processors along with the necessary context for each analysis idiom. As such, we developed an approach for the reduction of detailed performance event traces through the use of the k-Means clustering algorithm to find equivalence classes of processor behavior. This required the identification and pruning of dimensions of performance data that have low utility in characterizing processors, so that the clustering algorithm can be effective. Without a priori information regarding the types of performance problems we may encounter, the selection of the outlier and exemplar processors appeared to be the best general option. We placed a strong emphasis on the use of detailed data from outlier processors. We investigated the relative advantages of applying this approach online, just after the application has completed its work but before the performance data is written

to disk. We found that an environment that allowed interaction with the running application allowed a more flexible approach to determining optimal parameters for the k-Means algorithm. A low-overhead parallel implementation of the k-Means algorithm was developed to exploit the available parallel machine scalably, in both memory and time.

We showed that online live analysis presented a different class of powerful idioms for scalable performance analysis. We developed a novel and scalable mechanism to support such idioms. The adaptive runtime system of CHARM++ itself was deployed as a substrate for the processing and delivery of live performance profile data from a running parallel application to a remote client. We demonstrated that the use of this approached enabled online live analysis capabilities at very low overhead costs up to 8,192 processors. We expect the approach to scale to hundreds of thousands of processors with relative ease as the per-processor overhead of this method is constant.

For performance analysis idioms requiring repeated large scale access to parallel machines, we introduced simulation techniques to address the issues of time and resource consumption. We developed a technique for repeated performance hypotheses testing via simulation to produce predicted but detailed performance information using far fewer processors. Thanks to the general parallel messaging and computation dependency model implemented by the CHARM++ runtime system, we showed our ability to conduct hypotheses testing on both hardware and software properties under the same framework. Other systems require separate hardware and application-specific analytical models to achieve the same effect. We demonstrated these capabilities through examples that varied messaging latency as well as object-based load balancing strategies.

Finally, we showed the importance of a performance framework that allowed easy integration with third-party tools through a callback system. Through collaborative effort with the developers of TAU, we demonstrated how scalable analysis idioms in TAU could be applied directly to CHARM++ applications like NAMD. At the same time, we showed the potential for TAU to exploit scalability features adopted in the Projections framework.

The techniques developed in this work are already in use in tuning several production codes on machines with tens of thousands of processors. They are directly applicable to MPI programs via the adaptive MPI implementation that uses the same runtime infrastructure used in this work. Further, most of the ideas can be applied to other MPI implementations as well, except those that require the message-driven substrate, as in the case of live analysis. Undoubtedly, the techniques need to be further extended as one gains experience with an ever-increasing number of processors, running more complex, sophisticated and adaptive applications of the future. We believe that a foundation for such scalable analysis has been provided in this work.

References

[1] Mpi: A message passing interface standard. In *M. P. I. Forum*, 1994.

[2] G. Aguilera, P.J. Teller, M. Taufer, and F. Wolf. A systematic multi-step methodology for performance analysis of communication traces of distributed applications based on hierarchical clustering. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[3] D.H. Ahn and J.S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. *Supercomputing, ACM/IEEE 2002 Conference*, pages 3–3, 16-22 Nov. 2002.

[4] Katharina Benkert, Edgar Gabriel, and Michael M. Resch. Outlier Detection in Performance Data of Parallel Applications. *PDSEC*, April 2008.

[5] Abhinav Bhatele. Application specific topology aware mapping and load balancing for three dimensional torus topologies. Master's thesis, Dept. of Computer Science, University of Illinois, 2007. http://charm.cs.uiuc.edu/papers/BhateleMSThesis07.shtml.

[6] Nikhil Bhatia, Fengguang Song, Felix Wolf, Jack Dongarra, Bernd Mohr, and Shirley Moore. Automatic experimental analysis of communication patterns in virtual topologies. *icpp*, 00:465–472, 2005.

[7] Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kale. Integrated Performance Views in Charm ++: Projections Meets TAU. In *To appear in the Proceedings of The 38th International Conference on Parallel Processing (ICPP-2009)*, Vienna, Austria, September 2009.

[8] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[9] Magnus Broberg, Lars Lundberg, and Hakan Grahn. VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs. In *in Proc. 12th International Parallel Processing Symposium*, pages 770–776, 1998.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[11] Holger Brunst and Wolfgang E. Nagel. Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. *Proceedings of the Parallel Computing Conference (ParCo) 2003*, 2003.

[12] Marc Casas, Rosa M. Badia, and Jesus Labarta. Automatic Structure Extraction from MPI Application Tracefiles. *Lecture Notes in Computer Science*, 4641:3–12, August 2007.

[13] C. Catlett and et al. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In Lucio Grandinetti, editor, *HPC and Grids in Action*, Amsterdam, 2007. IOS Press.

[14] Nilesh Choudhury, Yogesh Mehta, Terry L. Wilmarth, Eric J. Bohm, and Laxmikant V. Kalé. Scaling an optimistic parallel simulation of large-scale interconnection networks. In *Proceedings of the Winter Simulation Conference*, 2005.

[15] I.-H. Chung and J.K. Hollingsworth. A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 45–56, 0-0 2006.

[16] I-Hsin Chung, Robert E. Walkup, Hui-Fang Wen, and Hao Yu. MPI tools and performance studies—MPI performance analysis tools on Blue Gene/L. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 123, New York, NY, USA, 2006. ACM Press.

[17] Bronis R. de Supinski, Rob Fowler, Todd Gamblin, Frank Mueller, Prasun Ratn, and Martin Schulz. An Open Infrastructure for Scalable, Reconfigurable Analysis. *International Workshop on Scalable Tools for High-End Computing (STHEC)*, June 2008.

[18] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.

[19] Luiz DeRose, Bill Homer, Dean Johnson, and Steve Kaufmann. Performance Tuning and Optimization with CrayPat and Cray Apprentice2. In *Cray User Group Meeting 2006*, Lugano, Switzerland, 2006.

[20] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray Performance Analysis Tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, June 2008.

[21] Markus Dittrich, Shigehiko Hayashi, and Klaus Schulten. On the mechanism of ATP hydrolysis in F1-ATPase. *Biophysical Journal*, 85:2253–2266, 2003.

[22] Isaac Dooley and Laxmikant Kalé. Detecting and Using Critical Paths at Runtime in Message Driven Parallel Programs. 2009.

[23] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. 14:437–449, 2006.

[24] Felix Freitag, Jordi Caubet, and Jesus Labarta. On the scalability of tracing mechanisms. In *Euro-Par 2002 Parallel Processing*, pages 55–77. Springer Berlin, 2002.

[25] Todd Gamblin, Rob Fowler, and Daniel A. Reed. Scalable Methods for Monitoring and Detecting Behavioral Equivalence Classes in Scientific Codes. *IPDPS*, April 2008.

[26] M. Geimer, F. Wolf, A. Knupfer, B. Mohr, and B. J. N. Wylie. A Parallel Trace-Data Interface for Scalable Performance Analysis. In *Proceedings of the Workshop on State-of-the-art in Scientific and Parallel Computing (PARA), Minisymposium Tools for Parallel Performance Analysis*, Umea, Sweden, June 2006.

[27] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, Bonn, Germany, September 2006. Springer LNCS.

[28] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, Bonn, Germany, September 2006. Springer LNCS.

[29] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[30] S. Girona, J. Labarta, and R. Badia. Validation of Dimemas Communication Model for MPI Collective Operations. *Proceedings of the European Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000.

[31] S. L . Graham, P. B. Kessler, and M. K. McKusick. GPROF: a call graph execution profiler. *SIGPLAN 1982 Symposium on Compiler Construction*, pages 120–126, June 1982.

[32] Eric Grobelny, David Bueno, Ian Troxel, Alan D. George, and Jeffrey S. Vetter. FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications. *SIMULATION*, 83(10):721–745, 2007.

[33] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Softw.*, 8(5):29–39, 1991.

[34] Adolfy Hoisie, Greg Johnson, Darren J. Kerbyson, Michael Lang, and Scott Pakin. A performance comparison through benchmarking and modeling of three leading supercomputers: Blue Gene/L, Red Storm, and Purple. *SC 2006*, November 2006.

[35] Chao Huang, Gengbin Zheng, and Laxmikant V. Kalé. Supporting adaptivity in mpi for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[36] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. June 1994.

[37] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[38] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[39] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[40] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[41] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[42] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[43] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[44] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[45] Darren J. Kerbyson, Adolfy Hoisie, and Harvey J. Wasserman. Use of Predictive Performance Modeling During Large-Scale System Installation. *Parallel processing letters*, 15(4):387–395, 2005.

[46] A Knupfer, H Brunst, and W E. Nagel. High performance event trace visualization. *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 258–263, 2005.

[47] Andreas Knupfer and Wolfgang E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. *icpp*, 00:165–172, 2005.

[48] Andreas Knupfer and Wolfgang E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. *icpp*, 00:165–172, 2005.

[49] Sameer Kumar, Chao Huang, Gheorghe Almasi, and Laxmikant V. Kalé. Achieving strong scaling with NAMD on Blue Gene/L. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[50] Sameer Kumar, Chao Huang, Gengbin Zheng, Eric Bohm, Abhinav Bhatele, James C. Phillips, Hao Yu, and Laxmikant V. Kalé. Scalable Molecular Dynamics with NAMD on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):177–187, 2008.

[51] Allen D. Malony, Sameer Shende, and Robert Bell. Online Performance Observation of Large-Scale Parallel Applications. In *Proc. Parco 2003 Symposium, Elsevier B.V*, volume 13, pages 761–768, 2004.

[52] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[53] Kathryn Mohror and Karen L. Karavanic. Evaluating Similarity-based Trace Reduction Techniques for Scalable Performance Analysis. In *Proc. ACM/IEEE SC09 Conference (Portland, OR, USA)*, November 2009.

[54] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A Scalable Approach to MPI Application Performance Analysis. *LNCS*, 3666:309–316, 2005.

[55] Alan Morris, Wyatt Spear, Allen D. Malony, and Sameer Shende. Observing Performance Dynamics Using Parallel Profile Snapshots. *Lecture Notes in Computer Science*, 5168:162–171, August 2008.

[56] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[57] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet. *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, June 2008.

[58] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen D. Malony, and Sameer Shende. TAUoverSupermon: Low-Overhead Online Parallel Performance Monitoring. *Lecture Notes in Computer Science*, 4641:85–96, August 2007.

[59] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.

[60] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[61] James C. Phillips, Willy Wriggers, Zhigang Li, Ana Jonas, and Klaus Schulten. Predicting the structure of apolipoprotein A-I in reconstituted high density lipoprotein disks. *Biophysical Journal*, 73:2337–2346, 1997.

[62] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[63] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[64] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable performance analysis : The pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[65] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2003. IEEE Computer Society.

[66] Philip C. Roth and Barton P. Miller. On-line automated performance diagnosis on thousands of processes. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 69–80, New York, NY, USA, 2006. ACM Press.

[67] K. Shanmugam and A. Malony. Performance Extrapolation of Parallel Programs. In C. Polychronopoulos, editor, *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 117–120. CRC Press, August 1995.

[68] K. Shanmugam, A. Malony, and B. Mohr. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems - Proceedings of the Joint Conference Modelling Techniques and Tools for Computer Performance Evaluation (Performance Tools '95) and Measuring, Modelling and Evaluating Computing and Communication Systems (MMB '95)*, number 977 in Lecture Notes in Computer Science, pages 254–68. Springer–Verlag, September 1995.

[69] S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[70] Sameer Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[71] Amitabh Sinha and L. V. Kale. Towards Automatic Peformance Analysis. In *Proceedings of International Conference on Parallel Processing*, volume III, pages 53–60, August 1996.

[72] M. J. Sottile and R. G. Minnich. Supermon: a high-speed cluster monitoring system. pages 39–46, 2002.

[73] Zoltán Szebenyi, Felix Wolf, and Brian J.N. Wylie. Space-Efficient Time-Series Call-Path Profiling of Parallel Applications. In *Proc. ACM/IEEE SC09 Conference (Portland, OR, USA)*, November 2009.

[74] Mustafa M Tikir, Michael A Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. *Proceedings of EuroPar'2009*, August 2009.

[75] Ramkumar V. Vadali, Yan Shi, Sameer Kumar, L. V. Kale, Mark E. Tuckerman, and Glenn J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[76] Jeffrey Vetter and Daniel Reed. Managing performance analysis with dynamic statistical projection pursuit. *sc*, 00:44, 1999.

[77] N. Karthikeyani Visalakshi and K. Thangavel. Impact of Normalization in Distributed K-Means Clustering. *International Journal of Soft Computing*, 4(4):168–172, 2009.

[78] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[79] F. Wolf, F. Freitag, B. Mohr, S. Moore, and B. Wylie. Large Event Traces in Parallel Performance Analysis. In *Proceedings of the 8th Workshop Ṕarallel Systems and Algorithms(ṔASA)*, Lecture Notes in Informatics, Gesellschaft fr Informatik, Frankfurt/Main, Germany, March 2006.

[80] F. Wolf and B. Mohr. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304, Klagenfurt, Austria, August 2003. Springer. Demonstrations of Parallel and Distributed Computing.

[81] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture*, 49(10-11):421 – 439, 2003. Evolutions in parallel distributed and network-based processing.

[82] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. *sc*, 00:50, 2000.

[83] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf. Holistic Hardware Counter Performance Analysis of Parallel Programs. *Parallel Computing: Current & Future Issues of High-End Computing*, 33:187–194, 2006.

[84] Brian J. N. Wylie, Felix Wolf, Bernd Mohr, and Markus Geimer. Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. *Lecture Notes in Computer Science*, 4699:460–469, June 2006.

[85] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.

[86] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.

[87] Gengbin Zheng, Orion Sky Lawlor, and Laxmikant V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.

[88] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

[89] Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, David Padua, and Philippe Geubelle. Performance modeling and programming environments for petaflops computers and the blue gene machine. In *NSF Next Generation Systems Program Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS)*, page 197, Santa Fe, New Mexico, April 2004. IEEE Press.