INTELLIGENT RUNTIME TUNING OF PARALLEL APPLICATIONS WITH CONTROL POINTS

BY

ISAAC J DOOLEY

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, 2010

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kale, Chair
Professor Michael Heath
Associate Professor Craig Zilles
Dr. David Jefferson, Lawrence Livermore National Laboratories

Abstract

The tuning of parallel programs on large distributed-memory machines today is usually a costly, and often extensive, manual process. Automatic tuning techniques can help reduce this manual burden. This dissertation investigates the utility of a new class of automatic tuning methods for large-scale parallel programs whereby each program exposes information about its behavior to the runtime system. This behavioral information enables a tuning framework to quickly find appropriate ways to reconfigure or steer the application towards better performance.

This dissertation describes both new automatic tuning mechanisms within a parallel runtime system, and a new framework that automatically reconfigures the behavior or structure of the program through one or more *control points*. Control points are a novel type of tunable parameter provided by an application wherein it exposes tunable knobs and information about the behavioral effects expected to occur as each knob is varied in each direction. This behavioral information associated with each control point allows tuning algorithms to identify the direction in which a control point should be adjusted to fix observed performance problems.

Multiple application case studies show that control points are useful mechanisms for dynamically reconfiguring applications to improve their performance. In these case studies, individual control points are examined to investigate how they can adjust diverse application behaviors including computational grain sizes, the amount of work offloaded to accelerators, the mapping of tasks to processors, the frequency of load balancing, and a communication throttling parameter.

To my parents Dr. Tom Dooley and Laura Dooley,
and to my wife Amanda Dooley,
for their love and support.

Acknowledgments

I would like to thank all the people that have helped me in this work. This work is not simply the product of one mind. Rather the work builds upon the ideas, existing applications, and infrastructure provide by many other talented individuals. Many of the ideas proposed in this thesis evolved through numerous discussions with my advisor Laxmikant Kalé, and with feedback from members of The Parallel Programming Lab and my dissertation committee.

My sincere thanks go out to my colleagues who have directly contributed to the ideas, text, and figures included within and to the numerous experiments performed for this thesis. Specifically I would like to thank Jonathan Lifflander, Chao Mei, Anshu Arya, David Kunzman, Aaron Becker, Yanhua Sun, Phil Miller, Eric Bohm, Ramprasad Venkataraman, Abhinav Bhatele, Filippo Gioachin, Forrest Iandola, Dr. Celso Mendes, and Dr. Gengbin Zheng.

This work has been made possible through the support of the wonderful DOE HPCS Fellowship program, and through funding by the NSF and DOE.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Existing parallel programming models and languages focus on the decomposition of data structures and control flow. The parallel portion of the application is often a library (e.g. MPI) used by the application. The library simply performs actions on behalf of the application such as parallel task creation and communication of data through messages or a shared namespace. In these existing parallel systems, the parallel runtime libraries do not influence or change the behavior of the application itself. Runtime systems, however, are uniquely poised to observe characteristics of a parallel program's execution as it runs in order to dynamically change the behavior of the application to increase its performance. Some existing parallel systems such as Charm++ already observe characteristics of a parallel program's execution in order to dynamically balance computation or communication load, but no mechanisms exist for the runtime system to control other behaviors of the application. This thesis evaluates a new mechanism through which applications expose information and the runtime system reconfigures the application. This approach extends the existing unidirectional flow of information and control to two directions (from the application to the runtime system and vice-versa). Specifically, this thesis investigates a new type of mechanism through which the runtime system can modify specific behaviors of an application in response to observed performance characteristics.

In this new approach, the runtime system will modify an application's behavior or structure through dynamically tunable parameters exposed by the application. These exposed parameters are called *control points*. An application will supply one or more control points while describing

their effects. This thesis examines various applications to find, evaluate, and classify useful control points that can be intelligently adjusted by an observant runtime system. The runtime system will be considered to perform intelligent tuning because it will use observed performance characteristics and knowledge about the effects of each control point when dynamically deciding how to modify, or steer, an application's behavior.

This work attempts to extend the philosophy of the Charm++ programming model. The traditional Charm++ philosophy includes the idea that an adaptive runtime system can instrument and adapt to the behavior of a parallel program. Prior to this thesis research, the adaptation only involved the application by requiring that the application provide a set of migratable objects that the runtime system can distribute dynamically. One benefit of this prior migratable object programming model is that most of the programming burden related to load balancing is eliminated. This approach has worked well when scaling some scientific applications to over 40,000 processors, and soon it will scale to over one petaflop/s of sustained application performance on hundreds of thousands of processors. Although this prior model has proven to be successful, much effort has still been spent manually tuning applications for each new parallel machine.

A key intuitive concept behind this work is that additional benefits ought to arise when an application provides a richer set of information to the runtime system about its behavior. New types of runtime system adaptations can be enabled when an application provides further information about its behavior. For example, if an application specifies how its various activities affect memory consumption, then an adaptive scheduler in the runtime system could schedule appropriate tasks when available memory is low. Or, if an application specifies that there are multiple paths of execution through its parallel activity graph, then an adaptive message prioritization scheme could use observed critical-path information to automatically prioritize the execution of the critical activities. These types of adaptation require the application programmer to expose small amounts of information about the program to the runtime system, enabling new mechanisms in the runtime system to affect scheduling or message delivery. No modification to the application is required beyond the minimal annotations that provide the information.

Unfortunately, certain application-specific behaviors cannot be modified

without more significant changes to an application. It is not possible for example for the runtime system to adjust an application's domain decomposition grain size without help from the application itself. Parallel applications do, however, frequently contain multiple alternative methods for decomposing a problem. The method by which the problem is decomposed is specific to the application, and hence an adaptive runtime system cannot modify this portion of an application's behavior without some mechanism exposed by the application. But, many different applications could expose their decomposition granularity parameters in a consistent uniform manner such that the runtime system could increase the application grain size if needed. A key observation is that many applications or modules in an application contain different internal methods for decomposing a problem, but the effects of changing the granularity are similar. Hence a single unified interface for adjusting the granularity or other such application-specific behaviors ought to be useful.

The tuning of an application at runtime by an intelligent runtime system will likely produce better performing applications than would be produced by a statically tuned program because more information is available about the system at runtime and programs may have dynamically evolving performance characteristics. The most flexible runtime optimizations are those that can be varied and adjusted within a single program run.

In this research, an API has been developed for exposing control points within applications and for exposing information about the effects of each control point. This API can be used by multiple program modules or libraries, potentially allowing the multiple modules to be co-optimized. Although co-optimization of many control points is likely to be useful, this thesis studies the adjusting of individual control points one at a time, developing various tuning strategies for each type of control point separately. This work differs from classical auto-tuning approaches because it dynamically tunes running parallel applications based upon observed characteristics of the program, feeding back information to the application through control points.

The approach taken in this dissertation follows a bottom-up approach, demonstrating the utility of multiple simple tuning schemes for individual control points. Such an approach results in an immediately useful infrastructure that can be used by other application developers. The goal is

not to build an all-powerful, or theoretically advanced tuning methodology, but rather to investigate solutions to practical problems. In the future the ideas presented in this dissertation will be incorporated with other ideas and solutions into complex, completely general, and fully automatic tuning frameworks for parallel programs.

CHAPTER 2

Methodology

The primary goal of this research is to identify and categorize useful places where instrumented performance characteristics can be analyzed at runtime to adapt the behavior of a parallel application using known information about the behavior of the application. To ensure that the resulting adaptation mechanisms are actually useful, the research will focus on identifying mechanisms that can improve the performance of real parallel applications.

The secondary goal of this work is to develop a general-purpose automatic tuning framework that can make intelligent tuning choices by utilizing both a knowledge base containing the effects of adjusting each available control point and the knowledge of past performance-related measurements. Mechanisms will be created to instrument and gather performance metrics that are related to the possible control point effects.

To achieve both of these goals, this research is composed of four phases. The first phase is to investigate cases whereby an application can provide a small amount of extra information about its behavior, and the runtime system can automatically adapt its own behavior to increase the application's performance. The second phase is to investigate a wide set of potential control points, creating a catalog of many conceivable types. The expected effects of each control point will be listed, along with potential application uses for each control point. The third phase consists of implementing the instrumentation mechanisms and creating a system that can modify application control point values after analyzing recent or past performance measurements. The fourth phase consists of adding control points to various applications, investigating the utility of different types of control points

individually.

The four phases in this project have overlapped in time with each other, as the overall progress has been driven by application case studies. Each application case study has advanced the understanding of the utility of one or more types of control points and has driven the creation of a new tuning technique or improvements to the API. The four phases are discussed in more detail in sections 2.1, 2.2, 2.3, and 2.4.

2.1 Phase 1: New Types of Adaptivity Within the Runtime System

This first phase of thesis research improves the Charm++ adaptive runtime system's ability to adapt its own behavior when behavioral information about an application is available. Here, performance can be improved through modifications to the activities performed by the runtime system, such as scheduling activities. For this phase, mechanisms have been developed that enable parallel programs to annotate behavioral information that enhances the types of adaptation that can be performed within the runtime system.

Chapter 3 describes a technique that allows an application to annotate which of its tasks are responsible for reducing the memory footprint of the application. The runtime system's scheduler can then intelligently choose to schedule tasks that reduce memory consumption when the amount of available memory is small.

Chapter 4 describes a technique that allows the parallel runtime system to automatically adapt an application's message priorities in response to observed critical-path profiles for the running application.

These two examples show that adaptive runtime systems are capable of previously unexamined types of adaptation. The existence of these new adaptive behaviors of the runtime system suggest that other new adaptive behaviors might also be found in the future. As further types of useful automatic adaptations are found, the importance of complex parallel runtime system such as Charm++ will increase. Perhaps these complex runtime systems will even displace the current widespread use of low level parallel libraries that do not provide many high-level features to an application.

2.2 Phase 2: Cataloging Control Points

This phase of thesis research involves identifying categories of control points, each with a well defined "knob" and a "language" for specifying its impacts on performance parameters, along with application examples for each category. The set of all known possible control points will then be separated into categories, forming a catalog. This catalog will list possible effects caused by adjusting each control point. The resulting catalog is provided in Chapter 5.

This information in the catalog about the effects of adjusting control points has driven the design of a programmable API for exposing that information by an application to the tuning framework.

2.3 Phase 3: Build Control Point Tuning Software Infrastructure

The third phase of thesis research involves building the software infrastructure necessary to tune control points. Building a real system ensures that the techniques and control points proposed in the previous phase are realistically implementable.

Chapter 6 describes the software infrastructure that has been developed within the Charm++ Runtime System. The additions include mechanisms for measuring performance characteristics of a running parallel application (6.2), an API that allows applications to expose control points (6.1), and implementations of multiple algorithms for tuning control point values. These tuning algorithms include standard direct space-searching techniques such as simulated annealing, the Nelder-Mead simplex algorithm, and exhaustive searches (6.3), along with the novel intelligent steering techniques developed specifically for this thesis (6.4). The steering techniques incorporate performance measurements with the information about the expected effects of varying each control point knob to intelligently choose the future control point values.

2.4 Phase 4: Application Case Studies

The final phase of this research consists of a set of application case studies. The application case studies will be used to refine the API used to express control points, while also improving the techniques used for gathering performance measurements and the algorithms for steering the control point values. Using case studies of real applications to improve the tuning framework maximizes the utility of the resulting system. As each case study is performed, tuning mechanisms will be created or modified to effectively tune each application. As the application studies are performed, when possible, the costs associated with adapting each application will be measured to quantify the overhead required to tune each application.

Although the framework built in phase 3 is designed to be able to co-optimize many control points at once, the application case studies presented in this thesis only investigate how a single control point can be adjusted in isolation from any others.

Descriptions of the multiple application case studies comprise multiple chapters of this dissertation. Chapter 7 describes how the grain size of a dynamic-parallelism tree-based computation can be adjusted as the program runs. Chapter 8 describes for a finite element structural dynamics application how the amount of work offloaded to a computational accelerator device can be managed automatically based on the observed load on the standard CPU processors. Chapter 9 proposes and evaluates multiple methods for automatically and dynamically adjusting the load balancing period (or frequency) of a different structural dynamics application as it runs. Chapter 10 describes how three different aspects of a dense LU factorization program can be automatically adjusted, albeit independently: the matrix block decomposition, the block-to-processor mapping function, and a parameter that reduces the available parallelism when memory consumption is low.

These chapters contain descriptions of the applications, performance results over control point parameter spaces, analyses of costs of adaptation, analyses of the programmer burden, and descriptions of multiple tuning scheme variants where possible.

CHAPTER 3

Memory-Aware Schedulers

3.1 Introduction

This chapter presents a novel type of adaptation within the parallel runtime system that can improve the performance of an application when the application exposes information about its memory consumption patterns. The proposed adaptation is a simple, but powerful memory-aware scheduling mechanism that adaptively schedules tasks in a message-driven parallel program. The scheduler adapts its behavior whenever memory usage exceeds a threshold by scheduling tasks known to reduce memory usage. The usefulness of the scheduler and its low overhead are demonstrated in the context of an LU matrix factorization program. In the LU program, only a single additional line of code is required to make use of the new general-purpose memory-aware scheduling mechanism. Without memory-aware scheduling, the LU program can only factor small matrices, but with the new memory-aware scheduling, the program scales to larger problem sizes.

It is well known that some parallel algorithms require large quantities of memory. Unfortunately, parallel systems have limited amounts of memory, and hence parallel programs must use algorithms that do not exceed the available memory bounds.

This chapter describes a general-purpose memory-aware scheduling technique that can automatically restrict the memory usage for a class of parallel algorithms that would otherwise run out of memory. Because the scheduling

technique is included in a general-purpose parallel runtime system, the parallel program needs only minor changes to use the scheduler.

Often it is easier to implement a simple naïve algorithm instead of a more complicated explicitly memory-aware algorithm. Hence the productivity of a programmer will likely be higher if a simpler memory-oblivious algorithm can be written while allowing the runtime system's scheduler to automatically restrict memory consumption.

All scalable, parallel LU dense matrix factorization implementations frequently used today are written using algorithms that explicitly restrict the progress of subtasks comprising the algorithm to ensure that there always is enough memory available to make forward progress. Some algorithms, such as the one used in the High Performance Linpack implementation use a fixed parameter that statically controls the *lookahead depth*, or number of algorithm stages that can be executed ahead of the oldest currently-active algorithm stage [2]. When the amount of lookahead permitted is small, the degree of concurrency is small and the required memory buffer overhead is small. Conversely, if the amount of lookahead is high, the degree of concurrency is higher but the required memory footprint becomes larger. The memory footprint expands because more blocks of incoming data are stored on each processor before pairs of these blocks are consumed in *trailing update* operations.

Other LU implementations use dynamic lookahead so they can fully exploit as much concurrency as will fit in the available memory [3]. Memory buffers are reserved for specific tasks in a certain order while sending and receiving processors coordinate the accesses to the reserved buffers to ensure that deadlock will not occur if memory is exhausted for some processor. Such implementations use an application-specific scheduler with a user-level threading package to allow the program to proceed in a safe manner.

One key goal that all implementations share is to achieve high performance. This can be achieved by performing computation aggressively along the critical path so that the parallel machine achieves high utilization. A message-driven style of programming such as Charm++ [4] allows this pattern of computation to be expressed naturally. The case study presented in section 3.3, an LU implementation, was written in Charm++.

3.2 Memory-Aware Scheduling Implementation

This chapter describes a memory-aware scheduling technique which constrains the memory consumption of a class of naïve parallel algorithms that are oblivious to memory consumption. The memory usage is reduced by the scheduler as it chooses to schedule tasks known to reduce the memory footprint whenever available memory resources are low. An implementation of this new scheduling technique was created by modifying the existing scheduler in the Charm++ Runtime System. The new scheduler can therefore be used by any Charm++ program, and hence it is general-purpose. In order for the scheduler to know which tasks should be scheduled when memory resources are limited, the system requires only minor changes to the Charm++ program. The programmer simply needs to add an annotation for each of the tasks that reduce memory consumption. This section describes the existing Charm++ scheduling system and the modifications that result in a simple memory-aware scheduler.

3.2.1 Existing Charm++ Scheduler

The existing Charm++ Runtime System uses a flexible scheduling mechanism to execute tasks spawned locally and tasks associated with incoming messages from other processors. In a Charm++ program, the tasks are *Entry Method Invocations* on *Chare Objects*, *Chare Groups*, or *Chare Node Groups* [5]. The flow of control for a Charm++ parallel program proceeds as entry methods are invoked. These entry methods perform computations and asynchronously invoke other entry methods.

The existing scheduler in the Charm++ Runtime System, which runs on each processor, supports prioritized execution in both LIFO and FIFO modes. Priorities or LIFO/FIFO designations can be associated with each entry method invocation. If no priority is specified, a default medium priority is implicitly assumed. When an entry method is invoked, its designated queuing scheme is stored along with any parameters to the method inside a message. Each message is then delivered to the destination processor or processors. Each destination processor will enqueue the message using the queuing scheme specified in the message's header.

Although the primary Charm++ scheduler queue acts just like a priority

queue, it is actually composed of three separate data structures: a high priority heap, a default (or zero) priority queue, and a low priority heap. Charm++ entry method invocations awaiting execution are stored in messages recorded in one of these three data structures. The reason that three separate structures are used instead of a single priority queue is that the double ended queue used for the frequent default priority case, with $O(1)$ insertion and removal time, can be slightly faster than a more complicated heap data structure, with $O(log(n))$ insertion and removal time.

3.2.2 New Adaptive Charm++ Scheduler

The new adaptive scheduler is a simple variant of the existing scheduler. The new scheduler adapts its behavior whenever the current memory usage for the processor exceeds a threshold. The threshold can be specified at runtime as a command line argument.

As long as the current memory usage is below a threshold, the scheduler acts as it normally would, processing messages one at a time in prioritized order from the scheduler queue. When the current memory usage exceeds the specified threshold, certain types of tasks are scheduled immediately even though they might have priorities lower than other tasks in the queue. Specifically, tasks that potentially reduce memory usage will be scheduled ahead of all other tasks whenever a processor's memory usage exceeds the threshold.

To modify the behavior of the scheduler when the memory usage is high, a call is made to a function that modifies the scheduler queue just prior to determining which task ought to be executed next. The modification function simply performs a linear scan through the three priority queue data structures, searching for the first task known to reduce memory usage. Once such a task is found, the task is removed from the priority queue and is re-enqueued with maximum priority. Then the scheduler resumes its normal operations, resulting in that task being executed next.

Of course, the scheduler needs to know which types of tasks are candidates for rescheduling. The adaptive scheduler therefore contains a list of such task types. The list is populated at startup with tasks specified by the application programmer in the application's interface file. All Charm++

programs contain one or more simple interface files that specify the entry methods and other parallel constructs in the program. A simple translator parses the interface file and generates C++ code that is compiled into the program to support the specified entry methods and other constructs. A new tag called `[memcritical]` has been added to the interface file's grammar and parser. When this new tag is used as an annotation to any entry method, the entry method will be included in the scheduler's list. Hence any invocations of the entry method will become candidates for rescheduling.

In a Charm++ parallel program run on p total processor cores across n nodes, there are p separate schedulers, each of which adapts its behavior independently of any processors on other nodes. The decision of when to adapt is made purely on local information without use of any centralized or distributed information. In an SMP build of Charm++, up to $\frac{p}{n}$ schedulers within one node may execute in the same operating system level process and hence in the same memory address space. In such an SMP configuration, the memory consumption for the program will be visible to all of the schedulers within the process. Thus decisions in the SMP version will be made based on the memory consumption of multiple Charm++ PEs, but still without any knowledge of memory consumption on other nodes.

3.3 LU Case Study

To evaluate the usefulness of the memory-adaptive scheduler described in section 3.2.2, an LU program was modified to use the adaptive scheduler. This section describes the LU implementation as well as its performance characteristics both with and without the adaptive scheduler. The resulting memory consumption patterns for the program are analyzed to show that the memory-aware scheduling technique does indeed reduce memory usage in a useful manner. The section concludes with a set of insights gained from this case study.

3.3.1 Experimental Setup

All runs of the Charm++ LU implementation are performed on 64 nodes of an IBM Bluegene/P system at the Argonne Leadership Computing Facility.

Only one processor core per node is used, to maximize the range of memory footprints per core that could be studied. This allows each core to use from 0GB to 2GB, providing the maximal insight into the behavior of the program. If more cores per node were used, then the results presented in this chapter would be truncated. Each node contains four processor cores running at 850MHz and 2GB of memory. All visualizations of processor timelines are generated from actual application traces analyzed using the Projections performance analysis toolkit [6].

For the performance critical numerical kernels, when using an IBM Bluegene/P system, the Charm++ LU program uses the `dgemm` and `dtrsm` routines from the Engineering and Scientific Subroutine Library (ESSL).

For performance comparisons, the well-known High Performance Linpack Benchmark (HPL) version 2.0 was run on the same system with identical block sizes and matrix sizes.

3.3.2 Charm++ LU Implementation

To write a dense LU algorithm, there are many implementation choices to be made. This section describes some of the design decisions made when developing a Charm++ implementation of dense square LU matrix factorization. The LU program was written as simply as possible, without any explicit memory-awareness in the parallel program's code. This implementation does not perform pivoting. Hence some numerical stability is lost, but the same number of floating point operations are still performed when compared to an LU program that implements pivoting [7].

The program uses a 2-D *chare array* to decompose a 2-D matrix into $b \times b$ square blocks. Each matrix block is stored in one of the chare array elements, while the mapping of chare array elements to processors is flexible. The default Charm++ mapping for 2-D chare arrays is a block mapping, but the program can easily specify other mappings, and for this LU program a custom one, called *balanced snake mapping*, was developed. Section 10.4 describes this new mapping scheme and its tradeoffs over the traditional block-cyclic mapping.

The main communication pattern that occurs throughout a blocked LU

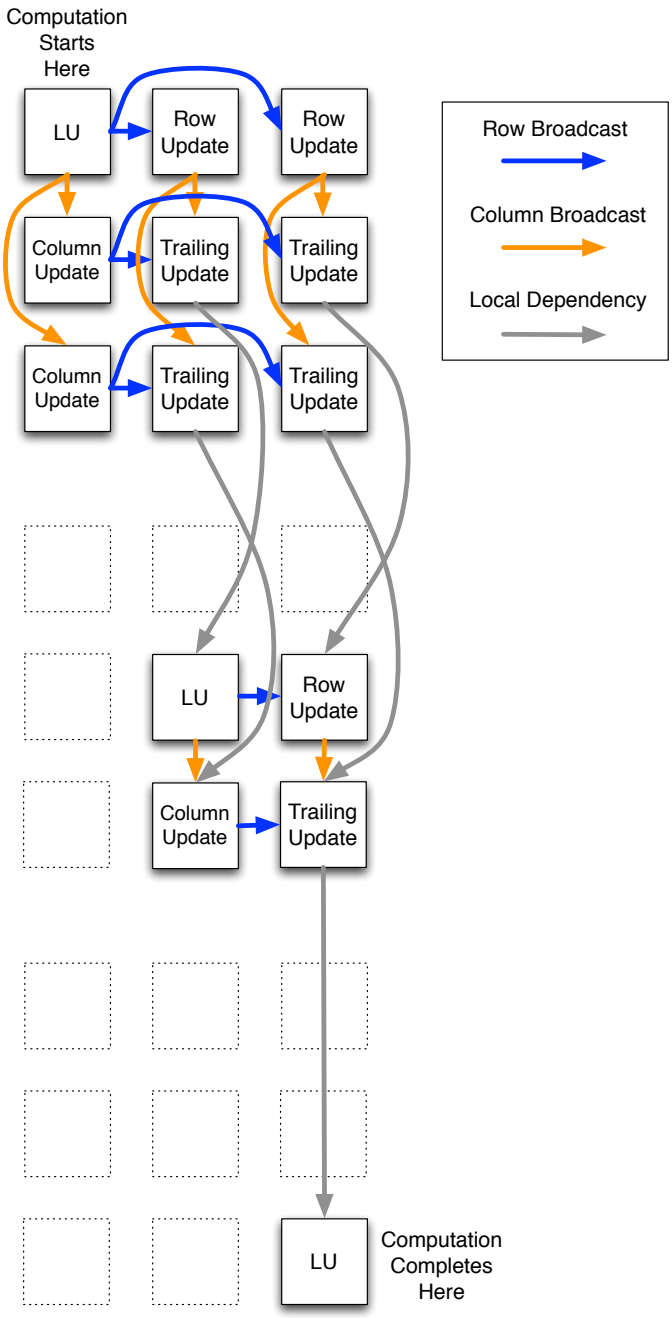


Figure 3.1: Structure of an LU factorization of a matrix decomposed into 3×3 blocks. Conceptually there are 3 phases, although overlapping work from multiple phases is acceptable as long as dependencies are preserved.

matrix factorization is a multicast[1] of a data block from a source block to all subsequent blocks in the same row, and a downward multicast of a data block from its source to all blocks below it in the same column. Figure 3.1 shows the structure of the program, including dependencies and the flow of data in the blocked algorithm for a coarsely decomposed matrix. As the program proceeds, the upper and leftmost blocks complete while the final result is produced only after the bottom rightmost block performs its own LU factorization.

The Charm++ language natively supports *chare array section sends*, which are mechanisms for sending a single message to a set of destination chare array elements. The programmer can choose one of many predefined algorithms for each of these communication operations [5]. The Charm++ LU program can therefore concisely express the pattern of communication that needs to occur. The multicast algorithm that appears to perform well for the cases described below uses a simple processor spanning tree of degree four.

The main computations performed in a dense LU algorithm are matrix-matrix multiplications that update the values in a block. This update operation is referred to as a *trailing update*. For block (i, j), the block LU algorithm performs $min\,(i, j)$ trailing updates. The closer a block is to the bottom right corner of the overall matrix, the more computation is performed for it. Other computationally intensive portions of the algorithm involve local single-block LU factorizations to be performed for blocks along the diagonal, and updates along the topmost active row and leftmost active column.

To factorize an $n \times n$ matrix, approximately $\frac{2n^3}{3}$ floating point operations are required. Assuming the matrix is decomposed into $b \times b$ square blocks, the fraction of the floating point operations spent inside the matrix-matrix multiply operation approaches $1 - \frac{1}{b^2}$ as b increases [7]. Thus for large LU factorizations, almost all floating point operations occur within the context of matrix multiplication. Therefore, a performance of a good LU implementation should approach the performance achieved by the double precision matrix-matrix multiply operation.

[1] Also called a broadcast

3.3.3 Priority Based Dynamic Lookahead

One general goal when writing parallel programs is to expose as much concurrency as possible to provide for the greatest opportunities to fully exploit the available processors and obtain high application performance. In a parallel LU factorization, there are important tasks along the critical path of the computation, namely the block LU factorizations and the following topmost active row and leftmost active column block updates. Scheduling such tasks as early as possible results in greater exposed concurrency earlier in the program. The other tasks, namely block trailing updates, can sometimes be delayed relative to the other tasks. If the trailing updates are executed with high priority, the program will not expose enough concurrency to keep all processors busy because the other critical path tasks are delayed in time. Alternatively, if the trailing updates are executed with low priority, then the critical path tasks will execute sooner, causing an avalanche of enqueued block trailing updates across all processors. The enqueued block trailing updates necessitate the buffering of two incoming data blocks. These blocks will occupy space in memory, and an increase in delayed trailing updates will directly relate to an increase in memory usage.

When writing an LU program, there are a few options regarding how much lookahead to support. High degrees of lookahead cause more trailing updates to be delayed, increasing memory usage. Low degrees of lookahead ensure that trailing updates cannot be buffered for too long, and hence the memory usage will not be as high.

The simplest LU implementations ignore the issue of lookahead and allow the program to proceed without regard to how far ahead one processor can compute relative to tasks buffered on itself or other processors [8]. Such an unlimited lookahead scheme is not scalable because memory usage can grow as the problem size is scaled up. At some point the program cannot run because memory is exhausted and the program will deadlock. Other algorithms, such as the one used in the High Performance Linpack implementation include a static parameter specifying the allowed degree of lookahead [2]. Yet other implementations support dynamic lookahead, but restrict some tasks so that deadlock will not occur when memory is exhausted [3].

Dynamic lookahead is important because better performance can be

achieved when there is a greater amount of available concurrency than in a static lookahead algorithm. Hence a dynamic lookahead implementation exhibited better performance than the traditional static lookahead implementations [3]. This dynamic lookahead implementation, however, contains application-specific code that explicitly coordinates between sending and receiving processors to ensure memory is not exhausted [3].

The Charm++ LU implementation described in this chapter is written to provide unlimited lookahead, with no code attempting to reduce concurrency. Priorities are assigned to tasks with higher priorities for block LU operations occurring in the upper-leftmost active blocks and lower priorities for the trailing updates, with priorities decreasing for each type of event from top left to bottom right. The priority scheme should provide as much concurrency as is available at any point in time.

This section shows that although the LU program itself is written with unlimited lookahead and hence a high level of available concurrency, a general-purpose memory-aware scheduling technique provides a sufficient mechanism to reduce the memory consumption of the simple LU program. This scheduling technique will dynamically vary the lookahead in the case of LU, but could also be used to control the memory usage patterns of other Charm++ programs.

3.3.4 Enabling Memory-Aware Scheduling

To enable the new memory-aware Charm++ scheduler, an application developer is only required to modify the Charm++ interface file (.ci file) for the program by adding one annotation to each entry method that could be used for reducing memory usage. The reason this current implementation uses annotations is that the user has knowledge of the program behavior, particularly which entry methods will decrease memory usage. In the LU implementation, the trailing update entry method is the sole method that is annotated for possible rescheduling when the memory threshold is reached.

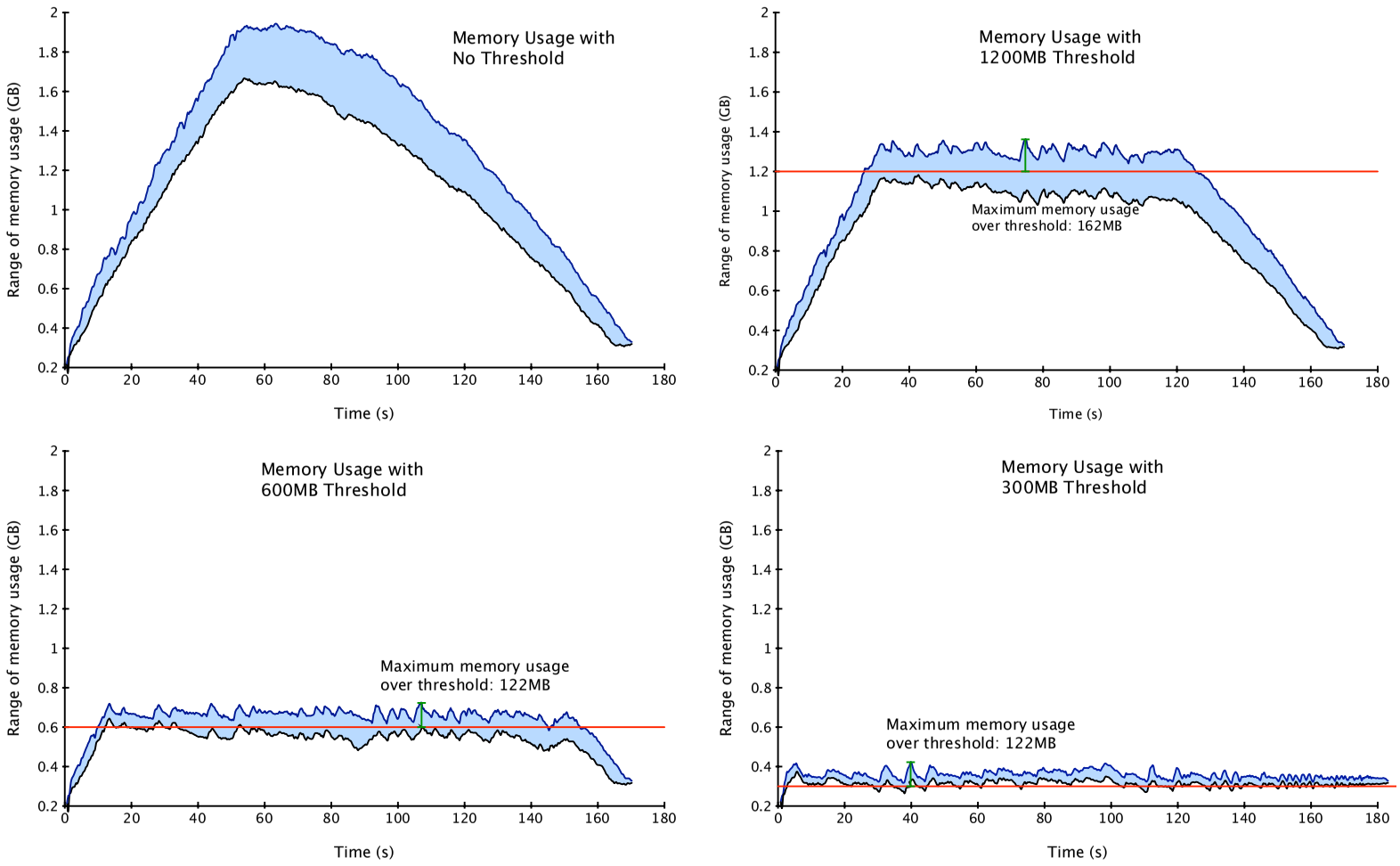


Figure 3.2: Ranges of maximal memory utilization across all processors over time for different thresholds. The adapting scheduler causes the memory usage to remain close to the threshold for this LU factorization.

Figure 3.2 was created by Jonathan Lifflander.

3.3.5 Analysis of Resulting LU Memory Patterns

To analyze the effects of the memory-aware scheduler, the LU program was run with various thresholds. Figure 3.2 displays the measured memory utilization over time for the program for various scheduler thresholds, for an $N = 32768$ sized matrix with 512×512 sized blocks. The horizontal red line displays the corresponding memory threshold for each run. This figure shows that adapting the scheduler queue does constrain the memory that is used on each processor. It appears that 300MB was the minimum effective threshold for this problem size, which is evidenced in figure 3.2 where the actual memory usage for all the processors is mostly above memory threshold. In the runs where the threshold is higher (600MB and 1200MB), the range of memory footprints for all processors mostly straddles the threshold. In all three cases where a threshold is applied, the memory usage is reduced from the original version where no adaptation was performed in the scheduler.

3.3.6 Analysis of Performance

As expected when testing the performance of the LU program, the higher memory usage configurations achieve higher performance than the more restrictive low memory threshold configurations. Figure 3.3 shows the performance of the program for various chosen memory thresholds[2]. In the figure two performance regimes are visible. The two regimes meet at the knee in the plotted curve. The first regime exhibits decreasing performance when lower thresholds are used, while the second regime is a large constant-performance plateau of sufficiently large thresholds.

When running with the N=32768 matrix problem size and 512×512 block size, the Charm++ LU implementation using the balanced snake mapping performs at 138 GFlop/s. The same implementation using a block-cyclic mapping performs at 131 GFlop/s. Both of these configurations perform better than HPL, a standard reference implementation of the Linpack Benchmark [2]. Figure 3.4 shows the resulting performance of 93 different configurations for HPL[3]. All of these configurations use the same N=32768 matrix problem size and 512×512 block size, but the other configurable

[2]Figure 3.3 was created by Jonathan Lifflander.
[3]Figure 3.4 was created by Jonathan Lifflander.

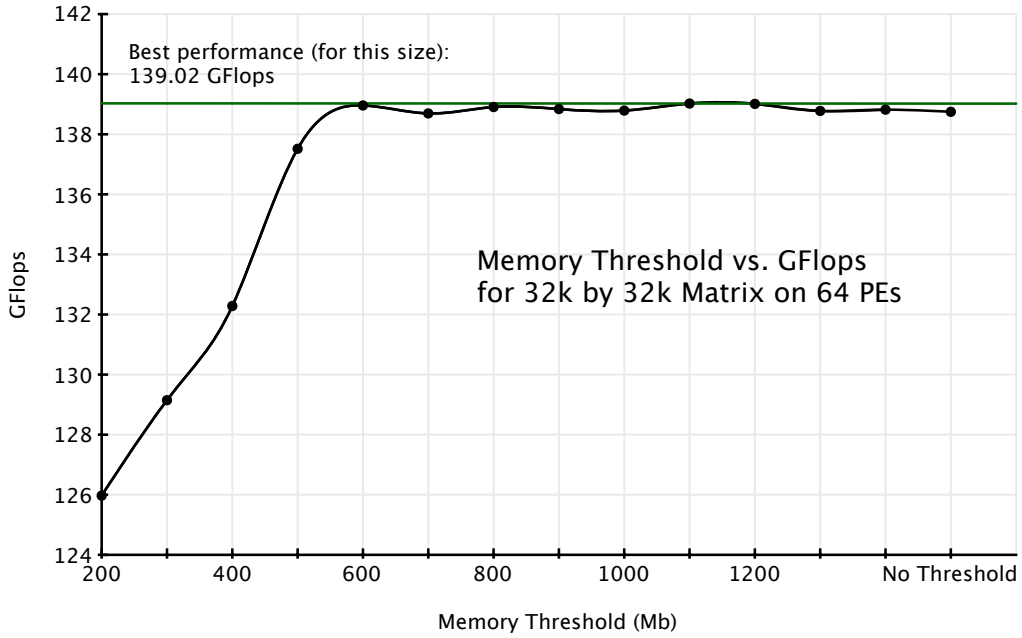


Figure 3.3: Performance of LU program for various memory thresholds. The problem is a factorization of an $N = 32768$ sized matrix with 512×512 sized blocks run on 64 processor cores of BG/P.

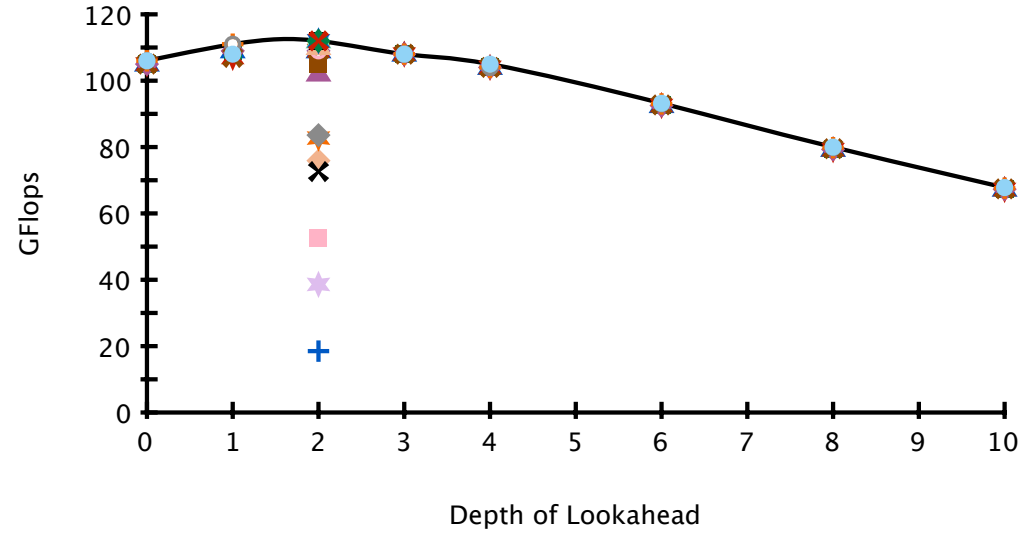


Figure 3.4: HPL performance on 64 processors for 93 different configurations for a $N = 32768$ sized matrix with a 512×512 block size. The configurations were tested in two phases. The first phase varied some parameters to find a good lookahead value. Then the best lookahead depth of 2 was fixed and more configurations were evaluated. The best observed HPL performance is 111 GFlop/s

parameters are varied. The broadcast method, processor grid arrangement, depth of lookahead, panels in recursion, and recursive stopping criterion were all varied. The maximal observed performance for HPL among these 93 different configurations is only 111 GFlop/s.

3.3.7 Costs of Modifying the Scheduler Queue

The overhead of adapting the scheduler queue for the LU factorization program is small. To measure the overhead, timer calls were added around the code that adapts the scheduler queue. Included in this code is the function that determines the current memory usage and compares it to a threshold. When the LU program is run with an $N = 32768$ sized matrix and a 512×512 block size, the average time spent in the scheduler modification code on each of the 64 processors was 0.0239 seconds while the whole LU factorization takes 168.4 seconds. This corresponds to a negligible overhead of 0.014%.

3.3.8 Insights Gained from the LU Implementation

The naively written LU program exhibits a simple memory usage pattern: memory usage changes slowly, and is relatively uniform across processors at each point in time. The memory usage generally grows to a single maximum value on each processor and then shrinks back down to the minimum required to store the matrix. The memory patterns are different however when a memory-aware adaptive scheduler is used, or when lookahead is restricted by other means. Hence, using memory-adaptive scheduling on each processor can constrain the memory usage in a useful manner.

The performance of the LU program over a range of memory thresholds shows two performance regimes. The first exhibits decreasing performance when lower thresholds are used, while the second regime is a large plateau of sufficiently large thresholds. Figure 3.3 shows that these two performance regimes meet at some point, namely the knee in the plotted curve knee in the curve.

A simple straightforward implementation of LU in the Charm++ language can achieve reasonable performance, while remaining flexible and not

requiring complicated application-specific schedulers or static limitations on lookahead. Charm++ makes it easy to specify the mapping of blocks to processors and to specify the priorities of each task. When developing the LU program, we found that a non-standard mapping outperformed the traditional block cyclic mapping, at least for some input matrix sizes.

Finally, the new adaptive scheduling technique enables larger LU factorizations to be performed, even ones that previously would have failed by depleting all available memory. Figure 3.5 shows a timeline visualization of one such larger factorization of an $N = 51200$ matrix size.

3.4 Automatically Finding an Optimal Memory Threshold

Although the scheduling scheme can reduce memory consumption for a certain class of programs, the memory aware scheduling scheme does not provide hard upper limits on the amount of memory used by a program. Thus a reasonable threshold needs to be chosen for a run of the program. The simplest scheme would be to set the threshold to a fixed fraction of the system's memory. A safer, and better solution is to automatically find the threshold that yields the best performance. This section describes an automatic scheme that slowly increases the threshold while observing memory consumption measurements across all processors.

The proposed scheme, which was implemented, is simple. The memory threshold is initially set to a safe low value, but it is automatically increased when previously observed memory usage measurements are low enough. After the threshold has been increased to a level where further increases are likely to exceed the desirable limits, the tuning framework [5] scans through its recorded history to find the best known configuration. The best known configuration can then be used for all future factorizations. This automatic tuning system can find a configuration providing good performance while restraining the actual memory consumption even when it exceeds the specified threshold. Figure 3.6 displays the actual memory usage over successive LU factorizations for a program using the automatic threshold determination scheme described in this section.

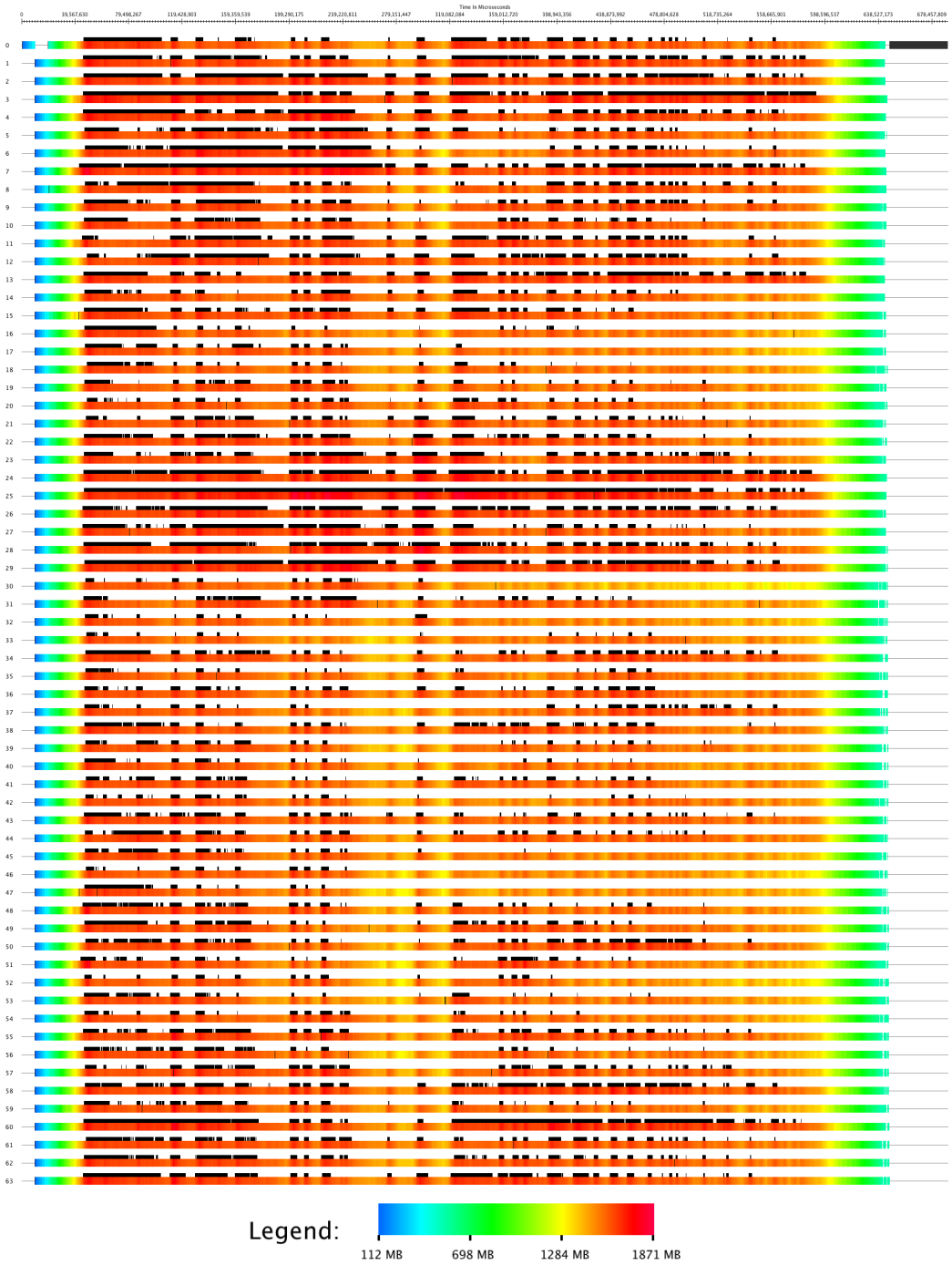


Figure 3.5: A timeline view of an execution of LU on 64 processors for a larger matrix N=51,200 using the adaptive scheduler. This same program dies when it runs out of memory when not using the adaptive scheduler. Each row in the figure corresponds to one of the processors, with colors indicating memory usage. Black tick marks on the top of each row indicate a point where a trailing update is immediately executed because the memory usage is over the specified threshold.

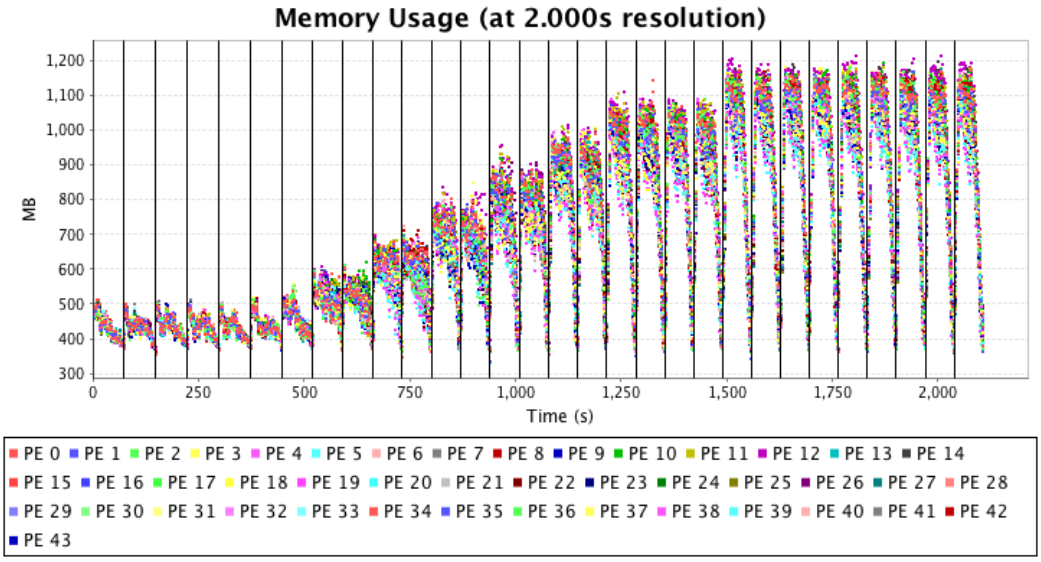


Figure 3.6: Actual memory usage for each of 44 processors while the LU program performs 30 successive factorizations. The memory threshold is increased by an automatic tuning mechanism whenever memory usage measurements from previous factorizations are still low.

3.5 Summary

A new method has been introduced for constraining memory usage dynamically over the lifetime of an application. This chapter showed that this method can be utilized by a programmer who simply annotates methods that reduce memory usage. Furthermore, the utility of this new scheduling mechanism was demonstrated by showing that an LU factorization algorithm can be scaled beyond the $N = 32768$ problem size, without any other modifications to the program. Typically, there is a tradeoff between implementing dynamic lookahead, which introduces many problems and increases the complexity of the program significantly, and using static lookahead, which constrains the concurrency. It was also shown that the best of these extremes can be realized in Charm++ using a simple LU factorization program, which implicitly allows for infinite lookahead but is constrained by the memory-aware scheduler so it can scale to large problem sizes.

A future direction for further research would be to automatically select tasks to be rescheduled to eliminate the use of annotations by the programmer.

CHAPTER 4

Adapting Message Priorities

Just as chapter 3 proposed a new type of adaptivity within a parallel runtime system that was enabled when an application expressed further behavioral information, this chapter too proposes a novel type of adaptation within a parallel runtime system. Specifically, the new type of adaptivity is the ability to automatically adjust message priorities based upon observed critical paths for a running program. This work, first presented in [9], is the first to use critical paths online, and to observe critical paths at runtime for message-driven parallel programs.

This chapter describes both how critical paths can be recorded for message-driven parallel programs, and then how the resulting critical-path profiles can be used to adjust message priorities.

4.1 Introduction

Detecting critical paths in parallel programs is useful for online automatic performance tuning. In the literature today, critical paths have not yet been used for online performance tuning. This chapter provides a discussion of how critical paths can be recorded efficiently for message-driven parallel programs. Three initial implementations cover three parallel languages: *Charm++*, *Charisma*, and *Structured Dagger*.

The work described in this chapter has been published in a paper showing that for the first time, in any parallel program, the critical-path profiles

recorded at runtime are used at runtime to automatically tune a parallel program [9]. By automatically adjusting message priorities based on the knowledge of which tasks occur along a critical path, a quantum chemistry application called OpenAtom realizes a performance gain of 10.2%. The use of critical paths for data reduction in performance analysis, application phase detection, and enhancing manual post-mortem performance analysis is also discussed. The costs of the proposed mechanisms are measured.

Critical paths are important paths through the execution of a parallel program. In the past, critical-path detection schemes were developed for some typical message passing models such as PVM [10] and MPI [11]. These approaches record a distributed Program Activity Graph (PAG) as a program executes by storing local portions of the PAG on each processor while augmenting each message sent between processors with information about the critical path leading up to the message send. The critical path can be extracted through a backwards traversal of the distributed PAG.

Although researchers have developed methods for detecting critical paths within the message-passing models of parallel computation, they have not previously detected critical paths at runtime within a message-driven execution model of parallel computing. In this work, we implement an efficient critical-path detection mechanism inside the Charm++ message-driven distributed object system. The Charm++ programming model has fundamental differences from the more widely used approach of programming at the level of communicating processors. These differences require revisiting and adapting the known algorithms for critical-path profiling, but the differences also provide fertile new ground for novel uses of the resulting critical-path profiles. This chapter describes both the implementation of critical-path detection for the message-driven programs and how the resulting critical paths can be successfully used for online automatic performance tuning and for other tasks.

We show that the critical-path profiles obtained online by our implementation can be used to automatically tune the performance of a complicated real-world quantum chemistry application, improving its performance by 10.2%.

4.2 Message-Driven Parallel Programs

The most widely used parallel programming model for distributed memory systems is the message passing model which has become standardized in MPI [12]. An alternative model is the message-driven execution model. In this model, the programmer does not write programs explicitly for a set of processors as is done in MPI, but rather the programmer describes the computation as a set of tasks whose computation is driven by messages sent by other tasks. The tasks may be mapped onto the computational resources dynamically by the runtime system.

The message-driven execution model is a paradigm that has proven to be successful for parallel programming. Scientific simulation codes such as NAMD [13] and OpenAtom [14] are written using this model. The message-driven approach could also be called *data driven* because tasks are dynamically scheduled when the prerequisite data (usually in the form of messages) is available. Directed acyclic graphs (DAGs) can be used to describe a message-driven program's pattern of computation and communication, with the edges in the graph representing the dependencies between all the computation tasks in the parallel program. Tasks can be scheduled in any order as long as all the dependencies for each task have been satisfied before the task is executed. The parallel runtime system can record the DAG when the tasks in the program are executed.

A dynamic message-driven program must include a scheduler responsible for executing tasks once dependencies have been fulfilled. This work focuses on the *Charm++* language [4] and two languages that each extend it: *Charisma* and *Structured Dagger*. The scheduler in these three languages is a general-purpose scheduler that is part of the Charm++ runtime system.

Writing parallel programs containing algorithms with complex dependencies in languages such as UPC or MPI will inevitably result in a program containing some sort of scheduling mechanism that can execute tasks in a smart manner once its dependencies are met. One such recent example of a program containing a specialized task scheduler is a UPC implementation of LU matrix factorization. It performs better than a traditional LU algorithm implemented in MPI [15]. The ideas in this chapter could also be adapted to these new complex parallel programs as well.

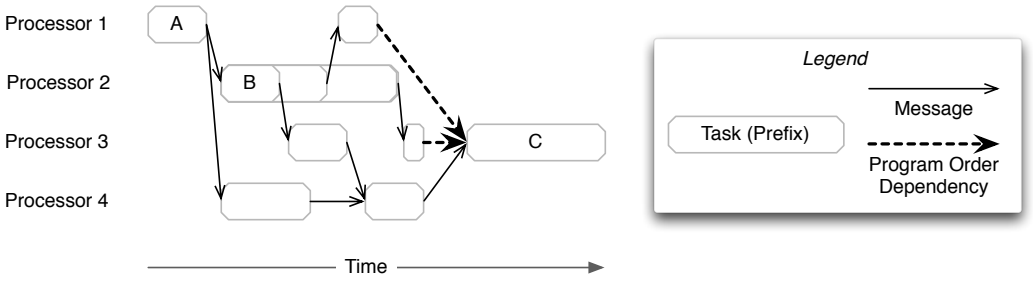


Figure 4.1: Example timeline view of a parallel program activity graph. Task A represents an initial task that multicasts a message to two other processors. Task prefix B can execute once its message from A arrives. Then task prefix B sends a message. Task C can only execute after three preceding tasks have completed.

4.3 Program Activity Graph Terminology

As a parallel message-driven program executes, its execution can be represented by a directed acyclic program activity graph (PAG), composed of tasks and their dependencies. Figure 4.1 shows a small example PAG composed of eight tasks that ran on four processors.

Because a message-driven parallel programming paradigm is different from the more commonly used message passing model, the existing definitions of critical paths used in the literature are not directly applicable. Thus in this section, definitions are provided to precisely define the critical path and the program activity graph for a message-driven program's execution.

Task: After its prerequisite dependencies have been fulfilled, a task is executed by a scheduler on a single processor. Each task may send messages that fulfill dependencies for other tasks. [1]

Task Prefix: A task prefix is the portion of a task from its beginning to the point where a message is sent or the task ends. There exist $m + 1$ task prefixes for each task which sends m messages. The weight of each task prefix is the the execution time for the task prefix.

Initial Task: Each program starts with the execution of a single initial task.

Terminal Task: A terminal task causes the parallel program to terminate.

Message Edge: A message edge represents a dependency from a sending task prefix to the execution of a task spawned by the message.

[1] In Charm++, each task corresponds to an *entry method invocation*.

Program Order Edge: Other dependencies due to sequencing requirements in the program are represented by program order edges. In a message-driven system, these sequencing requirements could be implemented as messages if spanning different processors.

Program Activity Graph (PAG): A PAG represents an execution of a parallel program, with one vertex for each task prefix and a set of edges comprising the dependencies between the task prefixes. The PAG is therefore a directed acyclic vertex-weighted graph.

Task In-Degree: Each task is executed once a set of messages have arrived and all other order dependencies have been satisfied. The in-degree of a task is the number of incoming message edges and program order edges to the task. The in-degree for all non-initial tasks is ≥ 1.

Task Out-Degree: Each task prefix either results in the sending of a message, or the completion of the task. Each message send or task completion can be the start of a message edge or program order edge in the program activity graph. The out-degree of each task is the number of messages sent by the task, plus the number of program order edges produced by the end of the task. All non-terminal tasks have out-degree ≥ 1.

Path: A path is an alternating sequence of task prefixes and edges in the PAG beginning with some task prefix and ending with another task prefix, where each task prefix is incident to both the edge that precedes it and the edge that follows it in the sequence.

Path Duration: The path duration is the sum of the node weights (task prefix execution durations) along the path. The path duration represents the minimum possible execution time of the path, with unlimited processors and an infinitely fast network. [2]

Critical Path (t): For each task t in the PAG, its critical path is the path of maximal path duration which ends at t and starts at the initial task.

The path duration of the critical path for a phase of an application represents a lower bound on the execution time for the application phase. It does not include any communication times along the path or the computation times for other unrelated concurrent tasks.

Critical-Path Profile (t): The critical-path profile for any task t is the critical path for t augmented with useful information about the task prefixes

[2] An idealized message duration could be included, but in Charm++ the costs of a message can often be overlapped with other work, and are hence of minimal importance.

comprising the path.

4.4 Algorithm for Determining a Critical Path

To determine the critical path for a program execution, we use an approach similar to the approach described in [16]. In both approaches, a distributed PAG is constructed at runtime, but the exact details of what is stored in the table is different. In our approach, each processor maintains a table of all task prefixes that have executed locally. Figure 4.2 shows an example of a PAG and the local information stored on each processor. To store the necessary information, an entry is added to the local processor's table each time a message is sent or when a task completes. Information that uniquely identifies the sending task prefix is appended to each message. Specifically, each message is augmented with two values, the first contains the duration of the path that led to the message send and the second uniquely identifies the sender-task-prefix in the sending processor's table. Messages must also contain a field specifying the index of the sending processor, but this field already exists in all Charm++ messages.

The critical path is determined for each task prior to its execution once all incoming dependencies have been satisfied. The path descriptors contained in all incoming messages or dependency edges are merged by selecting the one with maximal duration. All non-maximal incoming paths are ignored. Keeping the maximum incoming path maintains the invariant that each critical path extended along a dependency edge is maximal (critical). Figure 4.3 shows an example of three incoming message dependencies for a task. The incoming path with maximum cumulative path duration is stored in the processor table to be able to trace back any critical path that includes the task.

When a path is propagated forward via a message or other dependency, the duration of the extended path is found by adding the time spent executing the task prefix to the duration of its maximal incoming path. The new value representing the whole path duration is then stored in the newly prepared message.

When the critical-path profile is required for a task t, a backward traversal through the distributed PAG is performed. At each step in the traversal, the

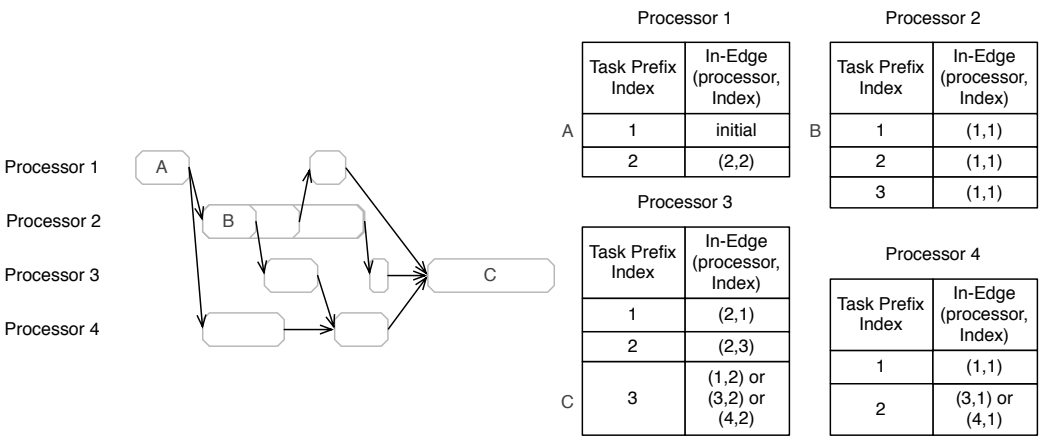


Figure 4.2: The tables created on each of four processors represent an example PAG. The specific entries for task prefixes with in-degree greater than zero depend upon the actual program execution, but here all multiple possibilities are shown.

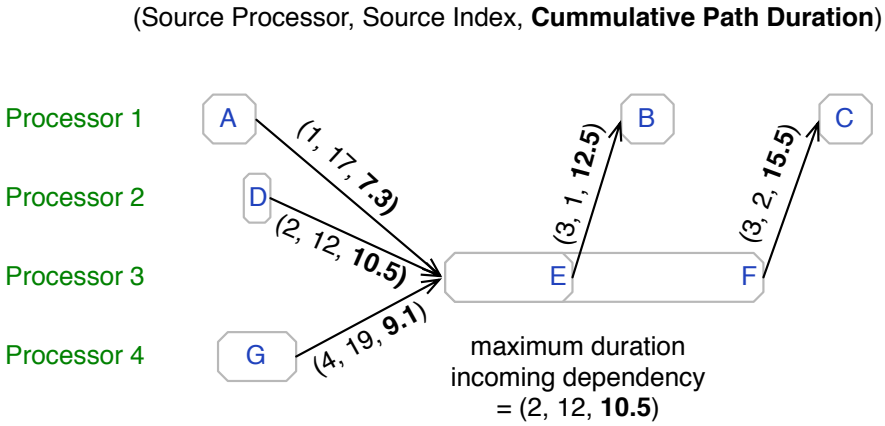


Figure 4.3: Illustration of how three incoming message dependencies are merged by recording only the one with maximal path duration. Each message contains information pointing back to a table entry for its sending task, as well as the critical path's duration.

information about the task prefix is retrieved and then its maximal incoming dependency edge is followed backward.

4.5 Implementations

We have implemented the critical-path detection algorithm inside the Charm++ runtime system. This implementation supports standard Charm++ programs as well as those written using the Structured Dagger or Charisma languages.

To implement the critical-path profiling algorithm, the following portions of the Charm++ runtime system were modified[3]:

- A new module was created with startup routines on each processor that create a table to hold the local portion of the PAG.
- The envelope used for all Charm++ messages was expanded to hold the critical-path duration and a reference to the sender's table entry.
- The message send functions were modified to fill in the envelope fields with the information about the currently executing task, after creating a new table entry.
- Methods for performing the backwards traversal over the PAG were created within the new module.
- Macros were created to simplify the storing of the maximal known incoming edge and the comparing of it with each new incoming dependency.
- Instances of the macros were added to the Charm++ reduction methods.

To use the new critical-path detection capabilities, a Charm++ program must be modified to add macros at each point where the program in-degree is greater than one. Charisma or Structured Dagger programs, however, do not require any modification because their compilers have been adapted to

[3]This implementation can be found in the publicly available development version of Charm++. The new module is located in the src/ck-cp directory.

automatically insert the macro instances wherever required. Sections 4.5.2, 4.5.3, and 4.5.4 provide examples of places in each of the three languages where the in-degree for a task could be greater than one.

4.5.1 Merging Multiple Dependencies For One Task

For any task with an in-degree greater than one, the longest path from the multiple incoming dependencies must be selected in order to correctly propagate forward the critical path. A set of three simple C++ macros have been provided to make it easy to select the maximal incoming path. The definitions of these three macros are provided in figure 4.4. The first, `MERGE_PATH_DECLARE` declares a variable that can be used to store information about the maximal path seen so far for a task. One macro definition should be added for each task with in-degree greater than one. The second macro, `MERGE_PATH_RESET` can be used to reset the values in the variable once all dependencies have been satisfied. The values ought to be reset at the end of one phase of a program prior to their use in a subsequent phase. Finally, the third macro, `MERGE_PATH_MAX` , is used to merge the previously longest path with a newly arriving message (i.e. dependency), by selecting the maximum of these two.

Each macro takes a parameter that distinguishes between multiple sets of dependencies that could be declared within the same scope in the source code. For example, if a single class has two tasks, each with multiple incoming dependencies, then `MERGE_PATH_DECLARE(A)` and `MERGE_PATH_DECLARE(B)` could be used to create data structures that store the two different incoming maximal paths, A and B.

The `MergeablePathHistory` class, which is used in these macros to store the information from an incoming message dependency, contains variables that store the originating processor, an index in that processor's table, and the path duration.

A second set macros are also provided for use in loosely synchronized iterative programs. In such a program, multiple iterations might occur concurrently. These three macros perform similar functions to those described earlier, except that they provide separate instances of the underlying variables for each iteration. The implementation of these variables

```
/// Wrappers for Charm++ programs to use to annotate their
    program dependencies

/// Declare a MergeablePathHistory variable, whose name is
    mangled with the supplied parameter
#define MERGE_PATH_DECLARE(x) MergeablePathHistory
    merge_path_##x

/// Reset the merge_path variable
#define MERGE_PATH_RESET(x) merge_path_##x.reset()

/// Take the maximal path from the stored merge_path variable
     and the currently executing path. Put the result in
    currently executing path.
#define MERGE_PATH_MAX(x) merge_path_##x.updateMax(CkpvAccess
    (currentlyExecutingPath)); CkpvAccess(
    currentlyExecutingPath) = merge_path_##x;
```

Figure 4.4: Definition of macros that can be used to merge multiple incoming dependencies for a task using an `MergeablePathHistory` object which contains information about the incoming path's duration, originating processor, and index to the originating processor's table.

uses an STL map data structure to store multiple `MergeablePathHistory` instances for different iterations. Figure 4.5 lists these alternative set of macros.

4.5.2 Charm++ Programming Model

The first language supported by the new critical-path detection scheme is Charm++. All Charm++ programs are written mostly in C++, with a small interface portion that is parsed by a very simple translator that generates C++ code.

In the Charm++ language, there are two places where in-degree is greater than one. In one of these two places, the user must augment their code with simple annotations specifying that multiple incoming messages are dependencies for a certain task.

1. Reductions from multiple objects to a single destination entry method result in an in-degree greater than one. The reduction framework in Charm++ has been modified to correctly compute the maximal incoming paths along any reduction tree, so the user does not need

```
/// Declare a dynamic MergeablePathHistory variable. Each
    object can have many merge points stored in this single
    DECLARE.
#define MERGE_PATH_DECLARE_D(x) std::map<int,
    MergeablePathHistory> merge_path_D_##x

/// Reset the merge_path variable
#define MERGE_PATH_RESET_D(x,n) merge_path_D_##x[n].reset()

/// Delete the merge_path variable
#define MERGE_PATH_DELETE_D(x,n) merge_path_D_##x.erase(n)

/// Delete all entries in the merge_path variable
#define MERGE_PATH_DELETE_ALL_D(x) merge_path_D_##x.clear()

/// Take the maximal path from the stored merge_path variable
     and the currently executing path. Put the result in
    currently executing path.
#define MERGE_PATH_MAX_D(x,n) merge_path_D_##x[n].updateMax(
    CkpvAccess(currentlyExecutingPath)); CkpvAccess(
    currentlyExecutingPath) = merge_path_D_##x[n];
```

Figure 4.5: Definition of macros that can be used to merge multiple incoming dependencies for multiple iterations of a loosely synchronized application.

to modify an application to handle the dependencies that arise due to reductions.

2. The user can buffer incoming messages explicitly until all necessary messages have arrived, at which point the execution of some task is performed. Each time the user buffers a message, a new implicit dependency is created and the in-degree of the task increases. Figure 4.6 shows an example of such explicit buffering. To correctly handle the critical paths, the user must augment the Charm++ program with macros specifying the existence of multiple incoming message dependencies. To do this, the user must add macros as described in section 4.5.1.

4.5.3 Structured Dagger Programming Language

The Structured Dagger language is an extension to Charm++. It allows a programmer to express a complex control flow with various dependency patterns easily. In Structured Dagger programs, messages are buffered

```
class myClass: public
    CBase_myClass {
  ...
  MERGE_PATH_DECLARE(A);
  ...
  void recvGhost(msg *m) {
    buffer_msg(m);
    MERGE_PATH_MAX(A);
    if(received_all_msg()){
      MERGE_PATH_RESET(A);
      subsequent_task();
    }
  }
};
```

Figure 4.6: Multiple dependencies occur when buffering messages prior to executing a task.

automatically until all input dependencies for an object have arrived, at which point the object's entry method is invoked. The dataflow patterns and all associated dependencies in the program are clearly expressed in the language, so the programmer does not need to add extra annotations to the program.

For clarity and to help implementers of similar languages, described below are the types of dependencies that must be handled for languages similar to Structured Dagger. Structured Dagger provides language constructs that impose ordering restrictions between the ends of some tasks and the beginnings of other tasks. Structured Dagger also provides language constructs for message sending and receiving.

1. All concurrent tasks specified inside an `overlap` block must complete before any subsequent task begins. Thus there are program order dependencies from the ends of the overlapped tasks to the beginning of the subsequent task. Figure 4.7 shows an example of this pattern.

2. Each `when` clause requires that one or more messages have been delivered prior to executing the following statement. Additionally, it requires that the preceding statement has also finished executing. Figure 4.8 shows an example of this pattern with two message dependencies and one program order dependency.

```
overlap {
  when recvLeft(msg *l)
    atomic { processLeft(l); }
  when recvRight(msg *r)
    atomic { processRight(r); }
}
atomic { subsequentTask(); }
```

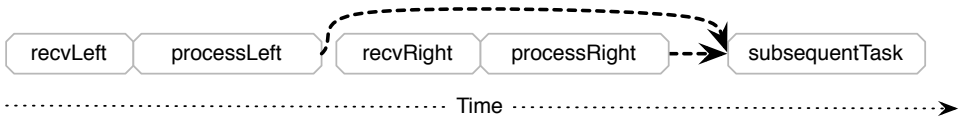


Figure 4.7: In a Structured Dagger program, the task following an overlap block will depend upon program order edges produced by each of the overlapped tasks.

```
atomic { previousTask(); }
when recvLeft(msg *l),
     recvRight(msg *r)
  atomic { process(l,r); }
```

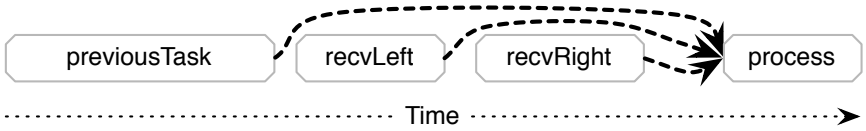


Figure 4.8: In this Structured Dagger example, the `process` task depends upon two messages as well as the program order dependency from the task preceding the `when` statement.

4.5.4 Charisma Programming Model

The Charisma language [17] is built upon Charm++. It allows a programmer to express various static dataflow and producer-consumer patterns easily. The dataflow patterns and all associated dependencies in the program are clearly expressed in the language, so programmers do not need to modify their programs for use with the critical-path detection scheme.

For clarity and to help implementers of similar languages, we will describe the types of dependencies that must be handled for languages such as Charisma. Charisma provides language constructs that impose ordering restrictions between the ends of some tasks and the beginnings of other tasks. Charisma also provides language constructs for producing and consuming

messages.

There are two places in Charisma programs where tasks can have in-degrees exceeding one. The Charisma compiler has been modified to record the proper critical-path information for these types of dependencies.

1. A statement can consume multiple input parameters:
   `workers[i].compute(lb[i+1], rb[i-1]);`

2. A reduction results in multiple dependencies flowing into a single task:
   `(+error) <- workers[i].getData();`

4.6 Overhead

The overhead of using the critical-path detection mechanism is small. Figure 4.9 plots the overhead for two simple benchmark programs. The benchmark programs were created to measure the costs associated with recording the table entries on each processor and the increased size of each message. Two versions of each benchmark program are run, and their results are compared to determine the overhead. In the first version, the critical-path functionality is entirely disabled. The envelopes are not augmented with critical-path table references, nor are the critical-path tables allocated. The second version is compiled against a version of Charm++ containing the critical-path mechanisms described in section 4.5. These experiments were performed on the Cray XT5 system named Kraken at NICS.

The first benchmark program sends a small message around a ring of Charm++ objects. Each object, upon receipt of the message, performs some amount of CPU work, and then sends a copy of the message to the next object in the ring. The amount of work performed by each object is varied to simulate various computational grain sizes. If the amount of work is small, the ring proceeds faster, while if the amount of work is greater, the ring proceeds more slowly. Each of the ring executions is timed, and the granularity of the program is calculated: $tasks\,per\,second = \frac{number\,of\,hops\,in\,ring}{ring\,execution\,time}$. The benchmark results shown in Figure 4.9 correspond to an execution of the benchmark with 40,000 hops around the ring for each granularity sample to amortize away perturbations. The second benchmark program exhibits more complicated communication patterns than the first

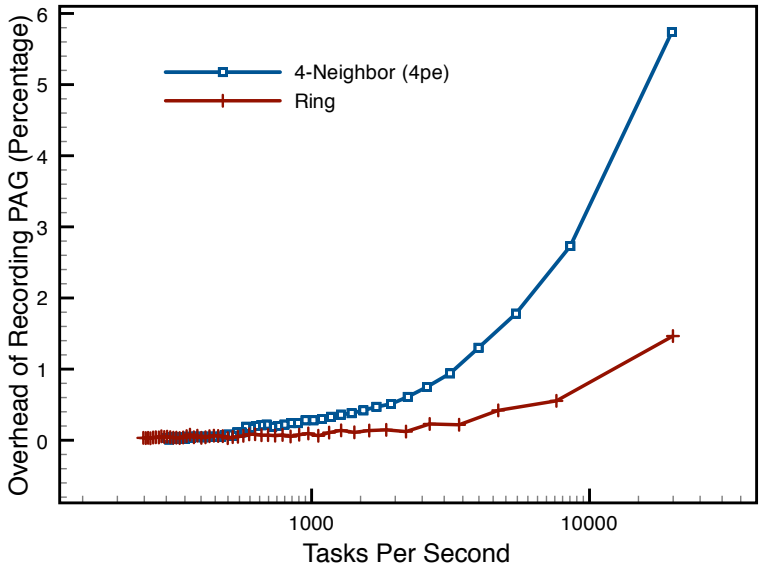


Figure 4.9: Overhead of recording critical-path information for varying computational grain sizes in both a ring benchmark program and a 2-D grid program where each processor communicates with four neighbors during each step.

ring benchmark. Specifically, a 2-D grid of chares is created, with each chare communicating with four nearby neighbors each step. Again, varying amounts of computation are performed each step to simulate various grain sizes.

Ultimately, the overhead for the ring benchmark executing 10,000 task prefixes per second on each processor will incur an overhead of about 1%. The overhead is caused by the recording of the information necessary to reconstruct a critical-path profile. That is, each task executes for about $100\mu s$ with an overhead of $1\mu s$. The overhead for the four-neighbor program is higher, as expected, because each task sends four times as many messages per step than the ring benchmark.

An experiment was performed to measure how much of the overhead was attributable to the two main mechanisms used to record a critical path or PAG. In this experiment, a portion of the critical-path measurement system, namely the code that allocates and updates a table on each processor, is removed. Then the four-neighbor benchmark program was run to measure the overhead when the critical-path duration is propagated in the message envelopes, but the tables are not created.

In this experiment, the baseline four-neighbor benchmark executes 3769 tasks per second. The first version of the program propagates critical-path information through all messages, but does not record table entries and hence would not be able to perform a backwards traversal of the path. The second version does include the full critical-path profiling system described in this paper, including the recording of entries in tables on all processors. The first version incurs an overhead of 0.19%. The second version incurs an overhead of 0.31%. Thus about 61% of the costs incurred by the critical-path monitoring are caused by the adding 12 bytes to each message envelope, the timer calls measuring the lengths of each task, and the propagating of information found in the messages any time a new message is sent. The remaining 39% of the costs are caused by the recording of information in the distributed PAG tables across the processors.

4.7 Using Critical-Path Profiles

In all the related work we have examined [18, 10, 16, 19, 20, 11], critical-path profiles for distributed memory parallel programs were gathered in online or semi-online manners, but the resulting critical-path profiles were only used for offline performance analysis. One novel contribution of this work is to show that critical-path profiles can be used both for automatic online performance tuning and for offline manual performance analysis.

There are already multiple uses of the new critical-path detection mechanisms in the Charm++ runtime system. Section 4.7.1 describes a simple online automatic task prioritization scheme that improves a complicated real-world application's performance by 10.2%. Section 4.7.2 describes some initial work in using critical paths at runtime to reduce the volume of performance trace data that is gathered for use in offline post-mortem performance analysis tools. Finally, sections 4.7.3 and 4.7.4 describe the uses of the critical paths offline in a more traditional manner to guide manual performance analysis.

4.7.1 Automatically Tuning Task Priorities

In typical Charm++ programs, there are frequently many messages available for processing (and hence enqueued in the scheduler) at a given time. One of the most obvious uses of critical-path profiles at runtime is to automatically adjust the scheduling priorities for tasks, so that the best task is chosen to run mong the available tasks. In many Charm++ programs, message priorities are hand tuned using both the programmer's intuition and experimental runs testing different configurations. Such manual tuning is time consuming and not portable for all applications.

An automatic message prioritization scheme was created in the Charm++ runtime system. The scheme extracts a list of types of tasks from a critical-path profile. The autoprioritization mechanism then modifies message priorities when outgoing messages are prepared by the runtime system just prior to being sent. In the current implementation, only messages allocated with priority bits are modified. The priorities on the messages are set based on whether or not the destination task type is found within a critical-path profile. Messages destined for critical-path task types are given a high priority while messages destined for non-critical-path task types are given a low priority. All other messages will retain a default medium priority. Using this simple autoprioritization scheme, speedups can be observed in real applications.

The developers of the OpenAtom quantum chemistry application [21] have found that by manually tuning message priorities, the application performance varies by about 10%. Unfortunately, the manually chosen priority configurations that work well on one parallel machine and input problem do not work well on other machines or with other input problems. Thus if the priorities could be automatically tuned, the performance of the application would improve for many of its users and the effort required to manually tune the program would be eliminated. Our test shows that indeed an automatic message prioritization scheme is useful.

To test the effectiveness of the automatic message prioritization scheme, we made two minor sets of modifications to the OpenAtom source code. The first modification was to add macros to mark the multiple incoming dependencies for certain tasks. These changes required adding the `MERGE_PATH_DECLARE` and corresponding `MERGE_PATH_MAX` and `MERGE_PATH_RESET` macros in

6 locations within 3 different classes in the source code. The second modification was to add a call to `useThisCriticalPathForPriorities()` after a specific iteration to start traversing the critical path and request that it be used for message prioritization. All of these changes were easy to make.

To run the program, we used the *water 32 70* system and ran the program on 64 processors of the Cray XT5 machine, Kraken, at NICS. The configuration files were modified to enable the prioritization of three types of messages, of which only two are enabled by default. We did not modify any of the specific message priority coefficients. Then the application was run for 40 application iterations. The first half of the iterations used the default message priorities while the second half used the automatic message prioritization scheme based upon the critical path gathered at iteration 20. The iteration timings output by the program were analyzed to determine the benefits of the automatic prioritization scheme relative to the default message priorities. Specifically, two startup iterations and the two fastest and slowest iterations for each case were ignored, resulting in 15 remaining iteration times for each case.

The resulting performance of the automatically prioritized portion of the application's execution was 10.2% faster than the other portion that used the default priorities. In both portions of the execution, the critical-path algorithm is enabled, so any overheads associated with the critical-path detection are present in both portions.

4.7.2 Performance Analysis Data Reduction

A second use of critical-path profiles is to reduce the volume of data produced for post-mortem performance analysis at runtime. The critical-path profile itself can be used online for filtering trace data produced by parallel programs run on many processors. At the end of the program, the critical-path profile is produced and broadcast to all processors. The processors use this resulting critical-path profile to determine how to filter their local performance trace logs before writing them to disk. The volume of performance data that needs to be analyzed offline is much smaller. Such savings are significant for large program runs on hundreds of thousands of processors.

To use this feature in the current implementation, the parallel program

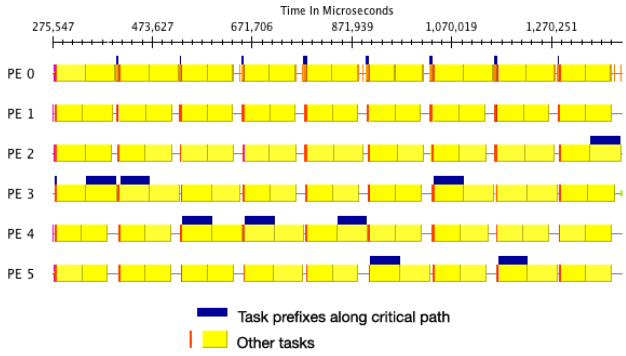


Figure 4.10: The critical paths can enhance visualizations of performance analysis in post-mortem analysis tools. This shows a timeline view of 6 processors of a Charm++ program simulating the 2-D wave equation. The task prefixes along the critical path are displayed as dark blue bars above on top of each processors' tasks.

can simply make a call to `traceCriticalPathBack()` near the end of its computation. This call will trace the critical path back to its origin at which point the entire path will be broadcast to all processors. The recipients of the broadcast will instruct the performance log tracing framework to not output the log files to disk if the processor is not found along the critical-path tasks. Thus any uninteresting processors, in the sense of not being on the critical path, will not write to disk their potentially large trace logs. The exact savings in data volume are program dependent.

4.7.3 Post-Mortem Performance Analysis

Critical paths can be displayed in a post-mortem performance analysis tool to help visually identify features of the program's execution in a timeline view. Our system provides the ability to generate the necessary trace log data from a critical-path profile. Figure 4.10 shows such a performance analysis visualization of a Charm++ program that solves the wave equation over a 2-d grid. Each processor contains 2 partitions of the 2-D problem domain. Thus about half of the program's execution time is spent along the critical path. There is a neighbor communication of ghost values each step, and each worker task is augmented with code that merges the incoming paths as described in section 4.5.2.

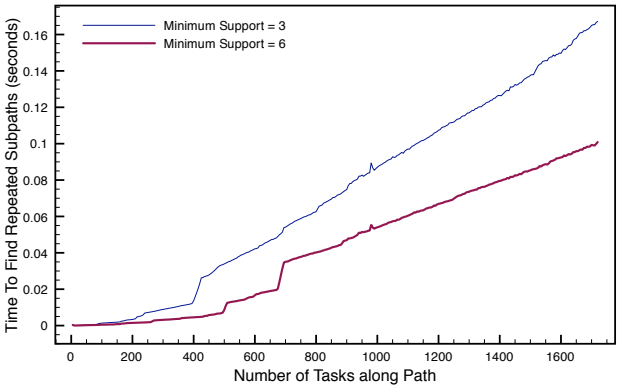


Figure 4.11: Time to compute frequently repeated sub-paths for varying lengths of input critical paths. The minimum support level is the number of times a sub-path must appear for it to be considered frequent.

4.7.4 Phase Detection

A final use of critical paths is to automatically detect repeating phases in a parallel application's execution. If the application's behavior is relatively static and the program executes a large number of iterations, which is common in most scientific computations, then the critical path should reflect the repeated phases of the program. Searching for phases in complete trace data would likely take longer than searching for phases in the much simpler critical path through the program's execution.

To make the problem of finding phases in the critical path easier, a critical path can be translated into a string over an alphabet of different types of tasks. The problem of finding repeating phases of execution along the critical path is the same as the problem of finding frequently repeated substrings. A preliminary implementation has been created using a dynamic programming technique to quickly build up the sets of frequently occurring substrings. This implementation arbitrarily requires that the frequently used substrings must have a minimum support level of 6, meaning that any candidate substrings must appear at least six times in the whole string. The higher the minimum support level, the faster the algorithm runs, but very long infrequent strings might not be observed.

The final output from the technique is the substring with maximal weighted coverage in the whole string. The weighted coverage is calculated to be $number\ of\ substring\ instances \times (substring\ length)^2$. This weighted coverage measurement favors frequently occurring repeated substrings while

Figure 4.12: A frequently repeated sub-path is shown in bold

especially favoring substrings with long lengths. Figure 4.12 shows a set of resulting repeated substrings, highlighted in the whole critical path. In the figure, each type of task is represented by a unique ASCII character. Multiple iterations, and their corresponding repeated portions are visible in the figure.

The execution time for the frequent sub-path technique is shown in Figure 4.11. The method empirically exhibits an execution time proportional to n for the trace produced for OpenAtom, with some beneficial cache effects for small strings. The actual worst-case performance is data dependent. With a minimum support level of 6, it takes about 0.1 seconds to extract the frequent sub-path from an OpenAtom critical-path profile of length 1721.

4.8 Other Types of Paths

In addition to critical paths, there is other useful information contained in a PAG. Two specific types of paths found in a PAG could be useful, namely *Near Critical Paths* and *Parallelism Exposing Paths*.

Near-Critical Paths: Paths with duration close to duration of the critical path are useful because they will become critical paths if the actual critical path is shortened. Hence, they can provide a bound on the execution time improvement in the program when a critical path is optimized.

Parallelism Exposing Paths: Nodes in the PAG that lead to large amounts of execution time across many tasks are important because they expose concurrency in the program. If these nodes, and their preceding critical paths are scheduled at higher priorities than other work, the potential concurrency in the program will be increased earlier in the program's execution. This would result in a potentially faster program. Although ideally all available concurrency ought to be exposed as early as possible, there may be costs associated with the degree of concurrency, and in programs

where this factor dominates, the paths exposing concurrency ought to be delayed. It is not yet clear how useful such paths are, or how they would be computed from a distributed PAG.

4.9 Summary

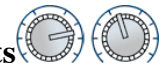
Just as chapter 3 proposed a new type of adaptivity within a parallel runtime system, this chapter presented a new type of adaptive behavior that can be performed within a parallel runtime system. Specifically, the new type of adaptivity is the ability to automatically adjust message priorities based upon observed critical paths for a running program. An application needs only to specify that this technique be used. One example application, OpenAtom, saw speedups of 10.2% when priorities were reconfigured based upon a critical path profile.

CHAPTER 5

Catalog of Control Points

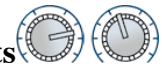
Chapters 3 and 4 described some new types of adaptivity within the runtime system that require little or no changes to applications. Further types of adaptations are possible when programs expose tunable parameters. This dissertation examines a special type of tunable parameter called a control point, for which an program provides a behavioral description of the parameter. This chapter provides a catalog of possible control points for HPC style applications. These potential control points are the basis for many decisions made designing the tuning framework described in this thesis.

A list of potential control points was constructed over a period of more than one year. All known Charm++ applications were considered, with attention being paid to the types of parameters already used in these applications. Any places where automatic tuning could conceivably be used were included in the catalog. These control points have been considered and grouped into categories. For each category, possible observable effects are enumerated. Finally, possible use cases in applications are listed. Tables 5.1 through 5.7 present this catalog. Of the control points listed in the catalog, some are analyzed in the application case studies of chapters 7, 8, 9, 10, and 11.

Catalog of Control Points 

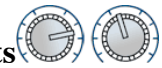
Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Data Decomposition Grain Size	Block size in each dimension Number of tree nodes per task Number of loop iterations per thread	Work unit durations Number of workers Number of messages Message sizes Degree of concurrency Potential for overlap Sequential performance	• 2D Gauss-Seidel • 2D Wave Equation • Matrix Multiplication • LU Matrix Factorization • NAMD (switch between 1-away and 2-away) • ChaNGa • FEM • Jacobi
Task Granularity	Divide & Conquer : Do serial / Spawn parallel threshold	Work unit durations Number of workers Number of messages Degree of concurrency	• Fibonacci • N-Queens • State Space Search
Multicore Node Usage	Number of cores to leave for OS Variance in priorities across cores Number of communication threads	OS Interference Sequential performance Degree of concurrency Messaging Overhead	• Anything on newer multicore clusters

Table 5.1: Control Point Catalog (1 of 7)

Catalog of Control Points 

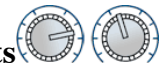
Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Low Priority Work Yield Frequency	Amount of work to perform before yielding	Work unit durations Number of messages Message sizes Degree of concurrency	• NAMD: Long non-preempt-able low priority entry methods can choose to yield to high priority tasks. The frequency at which they yeild can vary. In highly repetitive programs, like NAMD on BG/L, the exact optimal point for yielding an be determined through observing timings of message arrivals. • Structural Dynamics - ParFUM CUDA • OpenAtom • ChaNGa
Message Combining	Number of small messages packed into a single message Bytes worth of smaller messages packed into a single message Time to wait for additional small messages to be packed into a single message	Number of messages Message Size Degree of concurrency	*Programs with many small messages:* • PDES • ChaNGa
Message Compression	Type of compression Compression degree	Message overhead Message size	• Atmospheric Modeling • BT, SP, LU, sPPM, Sweep3D, AZTEC (cMPI shows improvements) • 2D Wave Equation • 2D Gauss-Seidel • Sorting

Table 5.2: Control Point Catalog (2 of 7)

Catalog of Control Points 

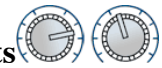
Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Phased Algorithm Decomposition	Number of phases	Work unit durations Number of workers Number of messages Message size Degree of concurrency Memory utilization	• Sorting • ChaNGa
Phases in Communication Operation	Number of phases in collective communication Number of phases in a multicast Extra barrier synchronization Loose non-phased throttling of collective communication	Compute overhead Message overhead Message size Number of messages Memory utilization Unnecessary Synchronization	• Collective communication algorithms • Sorting contains a large all-to-all data permutation. • Matrix multiply might contain a multicast. • Multiphase All-To-All uses less memory than all at once • 2D Wave Equation
Communication Granularity	How much data to put into a single message	Message size Number of messages Unnecessary Synchronization Memory utilization	• ChaNGa: *Subtree fetch size* = number of particles hashed and sent to a PE • Pipeline Filtering • Sweep3d • 2D Gauss-Seidel

Table 5.3: Control Point Catalog (3 of 7)

Catalog of Control Points 

Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Pipelining Decomposition of Chained Dependencies	Number of stages in pipeline	Work unit durations Number of workers Number of messages Message size Degree of concurrency	• Pipeline Filtering • Sweep3d • 2D Gauss-Seidel
Tree Algorithms	Subtree expand depth	Compute overhead Message overhead Message size Number of messages Memory utilization	• Cosmology N-body Simulation • Game Tree Search
Critical-path Priorities	Priority for objects on path Priority for entry methods on path	Degree of concurrency Memory utilization	• LU Matrix Factorization (critical path exposes concurrency, but increases memory usage) • OpenAtom • NAMD • Pipelines with other work • 2D Gauss-Seidel
Mapping Schemes	Which scheme: Load balanced, Round Robin, Block Dimensionality of mapping	Degree of concurrency Message overhead Memory utilization (transient due to messages)	• LU Matrix Factorization • 2D Wave Equation • 2D Gauss-Seidel

Table 5.4: Control Point Catalog (4 of 7)

Catalog of Control Points 

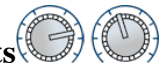
Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Load Balancing	Which load balancer(s) to use Load balancing frequency	Message overhead Compute overhead	*Any application with dynamic load imbalances*: • NAMD • ChaNGa • Structural Dynamics - ParFUM CUDA
Global Variables	Update frequency Update mechanism Update priority	Message overhead Compute overhead Number of messages	• Branch & Bound bounding variables
Software Caching	Cache size Amount of automatic prefetching Amount of pushing How many remote caches should be queried	Message overhead Compute overhead Number of messages Memory utilization	• Metric space nearest neighbor database search • ChaNGa • Multi-phase Shared Arrays
Array Section Multicasts	Whether to optimize the spanning tree To what degree is topology information used in mapping of objects	Compute overhead Number of messages Memory utilization	• LU Matrix Factorization • NAMD • OpenAtom
Accelerator Offload	Fraction of work to offload to accelerators	Compute overhead	• Structural Dynamics • ChaNGa • NAMD • Any program capable of using both GPU and CPU

Table 5.5: Control Point Catalog (5 of 7)

Catalog of Control Points

Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
FFT Algorithms	Transpose vs. non-transpose FFT Dimensionality of FFT decomposition	Compute overhead Message overhead Work unit durations Number of workers Number of messages Message size Degree of concurrency Memory utilization	• OpenAtom • PME phase of NAMD
Parallel Hash Maps	Hash map parameters	Compute overhead Message overhead Work unit durations Number of workers Number of messages	• Collision detection voxel grid size
Scalability of Algorithms	Which algorithm	Compute overhead Message overhead Work unit durations Number of messages	• Scalable Monte-Carlo or less scalable Deterministic (cf.)
Scheduler Configurations	Memory threshold for adaptive scheduler	Memory Consumption Degree of concurrency	• LU Matrix Factorization • ChaNGa

Table 5.6: Control Point Catalog (6 of 7)

Catalog of Control Points 

Control Point Category	Control Points	Parallel Performance Impact	Application Use Cases
Online Performance Analysis	Frequency at which performance data is gathered Frequency at which performance data is recorded	Compute overhead Message overhead Message size Number of messages Memory utilization	• For continuous performance monitoring, it is useful to gather performance data while the program runs, however the frequency at which the data is gathered will potentially affect the performance of the application. Some programs will be able to absorb the reductions of performance data, in which case the reductions ought to be performed more frequently.
Serial Algorithm Choices	Loop unrolling Code reordering Alternatives compiled with different optimizations Buffer sizes	Serial Performance	• Structured Grid Computations • Jacobi • LBMHD • Sparse Matrix Vector Multiply • Zip Compression • 2D Wave Equation

Table 5.7: Control Point Catalog (7 of 7)

CHAPTER 6

Tuning Applications with Control Points

This chapter describes how control points are exposed in applications, and how they can be dynamically tuned in response to measured characteristics of the running application. A new control point tuning framework has been created within the Charm++ runtime system to provide a concrete implementation of a system that automatically tunes control points. Within this chapter are descriptions of how this control point tuning framework is implemented and how it interfaces with applications to enable dynamic reconfigurations.

The tuning framework adjusts application control point values through a specified API (section 6.1) in response to performance measurements that indicate potential performance problems (section 6.2). The tuning framework supports two types of adaptations of the control point values. The first type of adaptation ignores any gathered performance measurements. Such adaptation algorithms perform traditional direct searches, attempting to find an optimal configuration without any behavioral knowledge of the available parameters (section 6.3). The second, and more complicated, class of adaptation algorithms uses performance measurements and behavioral knowledge to adjust control point values (section 6.4). The novel ideas found in this dissertation relate predominantly to this second class of measurement-based informed control point tuning, not the traditional direct-search algorithms. The high-level features of the tuning framework are visually depicted in figure 6.1.

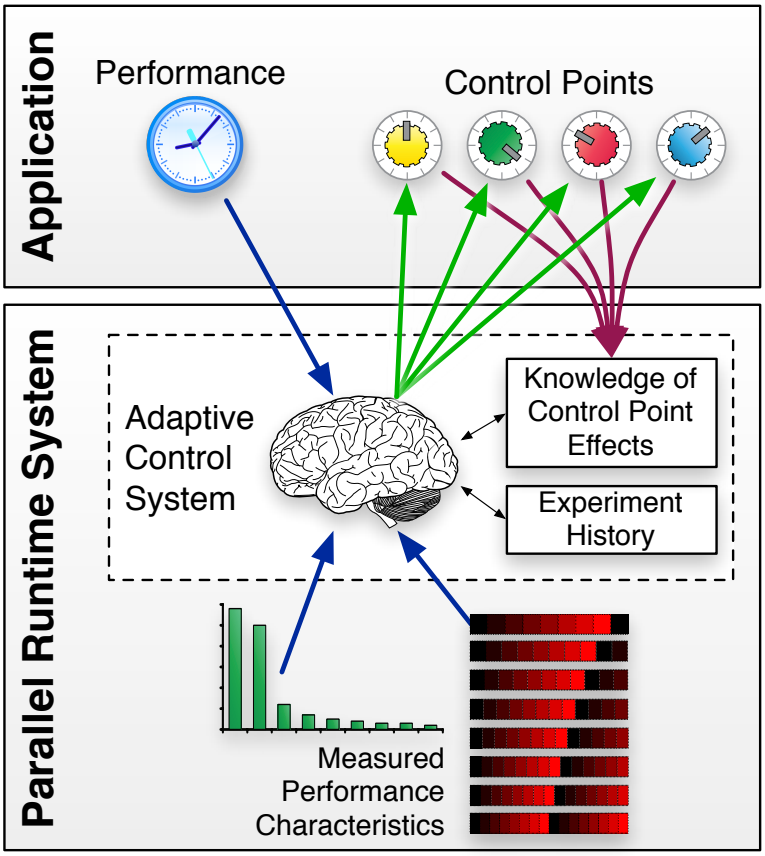


Figure 6.1: Overview of the Control Point Tuning Framework

6.1 Exposing Control Points Within an Application

This section describes how an application exposes control points to the tuning framework.

When optimizing the performance of the program, the actual application performance is generally the most important metric to maximize. For this work, scientific simulation applications are the focus, and these applications are generally composed of a sequence of steps. Such an application can specify that it has proceeded to its next iteration or step by calling:

```
controlPointTimingStamp();
```

Some programs may not have regular discrete steps, in which case they will not make the previous call. Of course, if no timing calls are provided, a direct-tuning scheme cannot be used. This dissertation proposes multiple scenarios where tuning can still be performed using other types of measurements to steer the control point values as a program runs.

A phase is a time range during which each control point value is held constant. A phase will likely contain many program steps, but in some programs it might not contain any. The tuning framework decides when to advance to a new phase and instructs the application to reconfigure itself through a callback. The callback is a standard Charm++ callback provided at startup by the application through a registration call such as:

```
registerCPChangeCallback(myCallback);
```

The application will then get the control point values for the phase by calling a simple function named `controlPoint`. This function takes as parameters the name of the control point and the range of acceptable integer values for the control point. An example call to controlPoint could be:

```
int controlPointValue = controlPoint("name", min, max);
```

In order for the control points to be tuned intelligently, the framework needs to be provided information about the effects of varying each control point. The effects specify high-level information about the structural changes in the program that occur as the control point knob is varied. The effects do not specify any low-level details about how the resulting program will perform.

The API includes functions that associate the high-level meaning to each control point. The information from all the calls made by the application are accumulated into a bank of knowledge stored by the tuning framework. For example, the program could make a call such as

```
EffectIncrease::AvailableParallelism("A");
```

to specify that the amount of available parallelism increases whenever the value for control point named "A" increases. Or, a call to

```
EffectDecrease::MemoryConsumption("B");
```

will specify that the memory consumption generally decreases as the control point name "B" increases.

Although some control point effects may directly and specifically relate to measurable characteristics, others, such as `AvailableParallelism` do not directly relate to performance or measurable characteristics such as processor idle time. Further information about the API can be found in [5].

To keep the design of the tuning framework as simple as possible, all the API functions listed in this section must be called on a single processor, which is currently processor zero. This restriction is made because the control point values provided within a phase are required to be consistent, and the control point lookup calls cannot perform communication (entry method invocations are non-preemptable). Additionally, because messages are delivered asynchronously, and not necessarily in order, it is impossible to ensure that control point values are consistent within a phase.

6.1.1 Necessary Runtime System Support for the Control Point API

It is relatively straightforward to create the necessary functions that record information about each control point, such as the range of acceptable values, or the behavioral effects, because this is merely standard sequential code called on a single processor. The more difficult portion of the interface is the part that notifies the application of a change to the control points. Callbacks are natural mechanisms within the Charm++ system, but they do not exist in other parallel programming paradigms such as MPI. In an MPI program, it would be necessary to poll or periodically check for changes requested by

the tuning framework, or to explicitly advance phases when desired.

6.2 Gathering Performance Measurements

The first step in automatic tuning of control points based on observed characteristics of a running program is to gather measurements upon which decisions will be made. This section describes the necessary runtime system mechanisms for gathering certain types of measurements, and how the control point tuning framework in Charm++ implements them. These performance measurements are broader in scope than just the execution time for a step in the program. Furthermore the measurements are orthogonal to the behavioral information provided by the application about each control point. Hence the specific types of measurements could vary, without requiring any changes to an application, because the measurements are only visible to the tuning algorithms.

6.2.1 Runtime System Support for Gathering Measurements

To gather measurements of a parallel program, there are three main facilities that must be provided:

1. The programming language, runtime system, or compiler must expose measurable characteristics of the program.

2. The runtime system must be able to record these measurements on each processor.

3. The runtime system must be capable of periodically gathering and combining measurements from all processors as the program runs.

The Charm++ runtime system satisfies these three prerequisites. Its programming model, based upon method invocations on migratable objects, inherently provides demarcated points throughout an execution where the runtime system's scheduler can record information about the running program. Communication is typically performed through method invocations[1],

[1]Other less frequently used communication mechanisms include a one-sided mechanism called CkDirect [22] and the discouraged practice of communicating through shared memory within a node.

so the runtime system can record information about all such communication events if desired. The Charm++ runtime provides an easy way for new modules, such as the tuning framework, to be composed with other modules and a user's program. The module interface allows the tuning framework to execute code on, store data on, and send messages to any processor. The Charm++ communication mechanisms readily support the broadcasts and reductions necessary for gathering the performance measurements from all processors.

Other commonly used programming models such as MPI or PGAS languages such as CoArray Fortran or UPC do not traditionally provide these three facilities. The programming models themselves do not require programs to be written in a manner that exposes the different types of computational sub-tasks that occur within each processor. Each program specifies its behavior at the level of one thread per processor. A compiler or additional programmer annotations could provide mechanisms for measuring where time is spent in the program, but this support is not part of these standard models. These programming models do provide support for creating instances of a library on all the processors, as is typically done in large scientific programs, although the flow of control in the user program must include explicit calls into the libraries for the libraries to perform any necessary actions[2]. If the runtime system, for example the MPI implementation itself, were modified, it would likely be possible to provide support for recording performance measurements at the points where any MPI routine is called. The most unnatural prerequisite for implementing a control point tuning framework in MPI or other similar programming models is the periodic gathering and combining of measurements from all processors. Some MPI implementations such as MPICH2 internally use an active message approach for communication, and adding another message handler to facilitate the gathering and combining of measurements would be trivial. Alternatively, an external communication subsystem like SUPERMON [23, 24] could be used.

Other parallel programming models such as X10 [25] necessarily use a more complicated runtime system than would be found in MPI or UPC [26, 27]. X10 programs can spawn parallel activities at runtime and hence each

[2]It may be possible to use multithreading to allow some progress in a library without help from the user program, but messaging typically is limited to a single thread.

processor, or more precisely each *place*, must have a scheduling mechanism responsible for scheduling the available activities. In such a system, the program exposes discrete units of work as activities to the runtime system. Thus because a scalable X10 runtime system would be quite similar to the Charm++ runtime system, it should be easier to create a control point tuning framework in X10 than in MPI or UPC. The performance measurements could be made by the X10 runtime schedulers, while the gathering of performance data could be performed through the execution of some X10 code that spawns activities on all places to extract the necessary data.

6.2.2 Some Useful Types of Measurements

One part of the Charm++ software ecosystem is a performance analysis tool called *Projections*. Any types of information that can be obtained through this visualization and analysis tool could potentially be used at runtime for automatically reconfiguring an application. Because Charm++ already supports dynamic load balancing, it contains mechanisms that are capable of measuring certain performance characteristics of a running program at runtime. To gather measurements that are useful for the tuning of control points, a new custom tracing module was created. The custom tracing module records the amount of time spent in each type of entry method, time spent idle, or time spent in overhead (the remaining unaccounted for time) on each processor. The overhead time represents time spent in the runtime system for handling communication and scheduling. Additionally, the tracing module can record memory usage statistics, and the average number of bytes for each entry method invocation.

The measurements produced by the tracing module are used by the tuning framework when it tries to make intelligent decisions. Thus it is important to gather measurements that are likely to inform the decision making processes. The current implementation is capable of gathering four main types of measurements:

- Processor utilization profiles
- Processor overhead profiles
- Memory footprint profiles

- Critical path profiles [9]

6.2.3 Measurements are Orthogonal to Control Point Effects

This work attempts to build general-purpose tuning mechanisms that are useful across multiple applications and runtime systems with a single interface. Due to the general-purpose nature of this goal, it is useful to separate the application interface from the underlying runtime system measurements that might change. Thus the following assumptions are made in these regards:

- A program should not contain its own application-specific tuning algorithms unless a general-purpose one cannot suffice.
- A program should not directly use runtime measurements, as these might change from one runtime system to another, between runtime system versions, or even dynamically as a program runs.
- The tuning algorithms should bridge the gap between the measurements taken by the runtime system and the behavioral changes that ought to be enacted.

The formulation of control points in this dissertation is significantly different than any other parallel autotuning system because an application provides information about the behavioral changes that occur when a control point is adjusted, but not the specific mechanisms by which the control point ought to be adjusted. Furthermore, this information is specified clearly at a high level, so that the runtime system can record relevant types of measurements, and potentially even change the types of measurements performed without requiring any modifications to an application. This division of labor between the runtime system and the application can be seen in figure 6.2. These resulting measurements are therefore useful to the tuning algorithms within the runtime system, but are not visible to the user program.

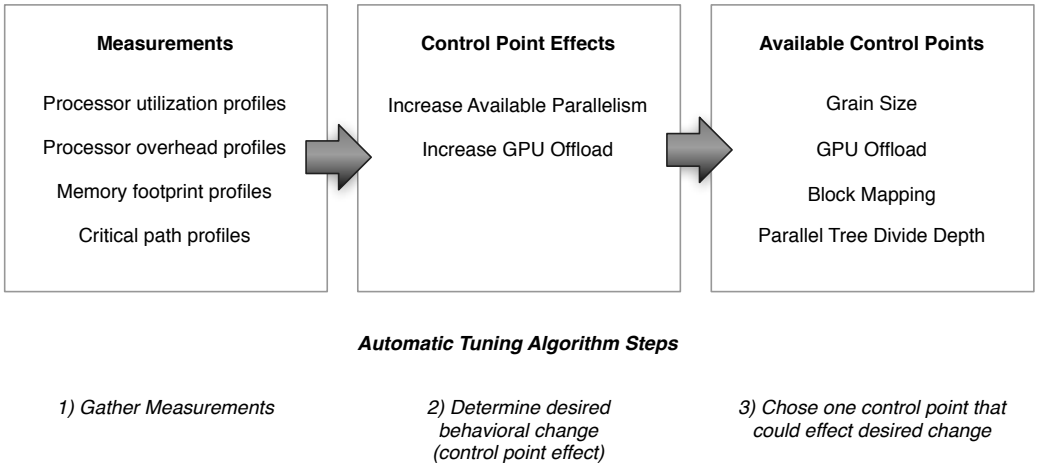


Figure 6.2: Measurements taken by the tuning framework are related to application provided control points by various tuning schemes through information about control point behavioral effects.

6.2.4 Implementation Details

Measured Charm++ Entities

To understand exactly what types of measurements can be made for a Charm++ program, it is first necessary to describe the entities within a Charm++ program. Charm++ implements a message-driven parallel programming paradigm. Each Charm++ program consists of collections of worker objects called *chares*, possibly grouped into *chare arrays*. Typically many chares are mapped onto each processor by the runtime system. The chares communicate with each other predominantly by invoking *entry methods* asynchronously and remotely on each other. Each standard method invocation results in the enqueuing of a message for the scheduler on the processor on which the chare currently lives. The scheduler on each processor executes the available entry method invocations one at a time non-preemptively from the prioritized scheduler queue.

Tracing Module

The charm++ runtime system supports the enabling of user defined tracing modules. Each tracing module contains a class which inherits from the `Trace` class, providing methods for any desired hooks. These methods for the

Tracing Interface Hooks	**Intercepted by Control-Point Tracing Module**
void traceBegin()	Yes
void traceEnd()	Yes
void traceClose ()	Yes
void beginExecute (envelope *)	Yes
void beginExecute (CmiObjId *tid)	Yes
void beginExecute (int event, ...)	Yes
void endExecute (void)	Yes
void beginIdle (double curWallTime)	Yes
void endIdle (double curWallTime)	Yes
void malloc(...)	Yes
void traceBeginOnCommThread()	No
void traceEndOnCommThread()	No
int traceRegisterUserEvent (...)	No
void userEvent (int eventID)	No
void userBracketEvent (...)	No
void userSuppliedData (int e)	No
void userSuppliedNote (char *note)	No
void userSuppliedBracketedNote (...)	No
void memoryUsage (...)	No
void creation (...)	No
void creationMulticast (...)	No
void creationDone (int num=1)	No
void messageRecv (char *env, int pe)	No
void beginSDAGBlock (...)	No
void endSDAGBlock (void)	No
void beginPack (void)	No
void endPack (void)	No
void beginUnpack (void)	No
void endUnpack (void)	No
void enqueue (envelope *)	No
void dequeue (envelope *)	No
void beginComputation (void)	No
void endComputation (void)	No
void endPhase ()	No
void traceClearEps ()	No
void traceEnableCCS ()	No
void traceWriteSts ()	No
void traceFlushLog ()	No

Table 6.1: Some of the traceable events that can be captured by a Charm++ tracing module, and which of these are used by the control point tuning framework's tracing module.

tracing module will be invoked by the scheduler whenever certain events occur. Table 6.1 lists many of the possible hooks, of which a few are used in the control point tuning framework[3]. The most important hooks are those that mark the beginning or ending of an entry method execution for a chare, namely `beginExecute` and `endExecute`. When there is no available work in the scheduler queue, then the scheduler will enter an idle state, indicating this transition with a call to `beginIdle`. Once a task is found, likely arriving on the network, or possible appearing locally as a periodic scheduled event, the scheduler will transition out of the idle state with a call to `endIdle`.

The tracing module uses wall timer calls to lookup the time at which each of these events occurs. By taking simple differences of the times, the amount of time spent in real work, between `beginExecute` and `endExecute`, can be tallied, as can the amount of time spent idle, or the remaining time which is considered to be overhead. Also a count of the total number of the number of entry method invocations is maintained. [4]

When desired, the tracing module can reset all its counters. This is useful when gathering performance data for different phases of an application.

The tracing module provides a set of methods that produce useful information from the counters and accumulated times that are stored within. One method provides a ratio for the fraction of time spent idle since the last reset: `double idleRatio()`. A similar method provides a ratio for the fraction of time spent in overhead since the last reset: `double overheadRatio()`. One method, `memoryUsageMB()` returns the high-water memory usage in MB since the last reset. Finally, a grain size approximation can be provided by `grainSize()` while the average number of bytes communicated per entry method invocation can be given by `bytesPerEntry()`.

The tracing module instance on each processor just accumulates values as its various hooks are called while the program runs. The gathering of the performance data from all processors is handled outside of the trace module.

[3]The full implementation can be found in the src/ck-perf/trace-controlPoints.C file in the Charm++ distribution.

[4]One important semantic issue with the tracing module is that in some cases it is possible for the executions of multiple entry methods to become nested. Hence the tracing module also keeps track of the current nesting depth, ignoring any entry method invocations that are internal to others. The cases where this scenario can arise are when using certain special types of entry methods: `[inline]`, `[immediate]`, and `[local]`.

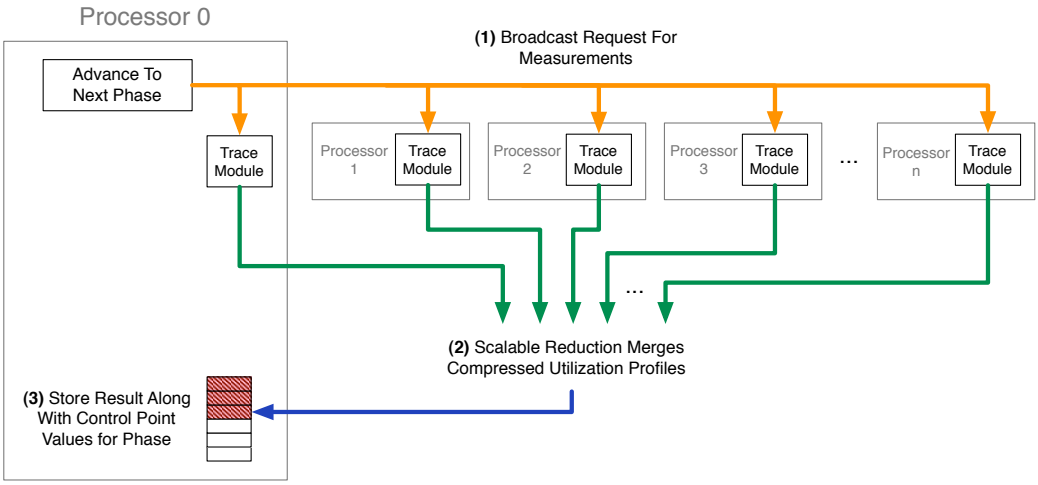


Figure 6.3: Measurement Gathering Mechanism Overview

Gathering Performance Data

The control point tuning framework is capable of aggregating the measurements produced by the tracing module on each processor. There are two methods for gathering this data. In programs composed of many phases, the program will specify that it desires to transition to the next phase. Alternatively in programs not composed of regular steps or phases, the phases are advanced periodically based upon a timer. In either scenario, upon the transition to the next phase, the data is gathered from all processors to be stored on processor zero as shown in figure 6.3. The phases are always advanced on processor zero, so at that point the broadcast is made to all processors, and the results are combined through a reduction back to processor zero. These broadcasts and reductions are scalable, but asynchronous. Thus they are interleaved with the execution of the program.

More specifically, the broadcasts requesting data are performed by invoking an entry method named `requestAll(cb)` on the `controlPointManager` group which contains an instance on all processors. This method on each processor contributes 12 double precision floating point values into a reduction whose destination is the specified callback `cb`. The values include the following: minimum idle time, maximum idle time, total idle time, minimum overhead time, maximum overhead time, total overhead time, memory usage, four values representing message sizes (in bytes), and the average computation grain size (i.e. average entry method invocation

duration). A custom reduction handler function was created that merges these fields correctly. The result of the reduction gets sent to the specified callback which in this case is a method named `gatherAll(CkReductionMsg *msg)` on the `controlPointManager` instance on processor zero.

There are numerous tradeoffs to performing the measurement gathering process asynchronously. One benefit is that the process is interleaved with the program's execution, which eliminates the need for a possibly costly global barrier. Secondly it reduces the need for application modifications. For example, if measurements are to be gathered within a single program iteration, the application might otherwise need to add multiple function calls throughout its code to yield the flow of control. There are a few drawbacks to asynchronously gathering data. The first is that if the frequency of data collecting is too high with respect to the time spent or number of messages sent between reconfigurations of the control point values, the data might not accurately represent the expected phase. The second difficulty encountered with asynchronous data collection is that an application's behavior might be perturbed in an unexpected or detrimental manner. Additionally, the entire measurement gathering mechanism runs asynchronously overlapped with the execution of the program. The results will eventually be combined and stored in a data structure on processor zero, but these measurements are not immediately available. This means that a tuning scheme can likely only access performance measurements that lag one or two phases behind, but the tuning algorithm implementations developed so far, and presented elsewhere in this dissertation, take this restriction into account.

6.3 Direct-Search Algorithms for Choosing Control Point Values

The common approach for auto-tuning is to directly search through the parameter space to find a configuration yielding an optimal value for an objective function, which is usually just the execution time for a program. Direct searches are only useful for programs whose behavior doesn't dynamically vary because multiple configurations are tested as the program runs and their resulting measured execution times are directly compared. Simple direct search schemes usually assume that the difference in execution

times between multiple configurations must be caused solely by the changing of parameter values, not by some dynamic characteristic of the program itself.

In a direct-search approach to optimization, the only information about the partial derivatives of the objective function with respect to the parameters is what can be inferred from multiple sampled configurations. In the measurement guided approach proposed in this dissertation, some information about the partial derivatives of the objective function may be known because the program provides behavioral information about the control points. This information can then be used to know which direction to adjust certain parameters when certain performance problems are observed. Although the control point tuning framework contains steering mechanisms that use the information about the parameters, the framework is also capable of performing simpler direct searches that do not use any information about the effects of varying the parameters. To date, four direct-search methods have been implemented in the Control Point Tuning Framework, and are available to any Charm++ program that exposes its control points:

1. Random Search *(described in section 6.3.1)*
2. Exhaustive Search *(described in section 6.3.2)*
3. Simulated Annealing *(described in section 6.3.3)*
4. Nelder-Mead Simplex Method *(described in section 6.3.4)*

The tuning framework can save the data from any run to be used in future runs. This allows a costly direct search, such as an exhaustive search, to be performed once in a training run, with the results being reused in all future runs. If a program has no dynamic performance characteristics, a single costly direct search might be useful. Additionally, direct searches are especially useful in programs where no knowledge is available about the behavioral effects of the tunable parameters, or in programs with complex non-linear, but static behaviors. The remainder of this section describes these four direct-search algorithms, as they are implemented in the control point tuning framework.

6.3.1 Random Search

One incredibly simple scheme for finding good configurations is to just randomly choose values for each control point. After some number of configurations have been tested, the best configuration can be selected for use in all future phases. The Charm++ control point tuning framework provides such functionality. Random searches are useful for programs with static but non-differentiable and non-continuous objective functions over the parameter space. In some programs the best configuration might occur at a random configuration. Simply trying many configurations is all that can be done to optimize the performance of such programs.

The `+CPSchemeRandom` command line argument enables this randomized scheme.

6.3.2 Exhaustive Search

To find the parameter configuration that performs best for a program with static performance characteristics, all configurations must be examined. To do this, a brute-force search exhaustively scans through all possible configurations. The control point tuning framework provides this capability to any Charm++ program that exposes control points. The implementation tries all configurations and then after all have been examined, chooses the best one. Exhaustive searches are useful when the parameter space is small, and the absolute best performing configuration is required.

The `+CPExhaustiveSearch` command line argument enables the exhaustive search scheme.

6.3.3 Simulated Annealing

Simulated Annealing refers to a class of optimization algorithms whereby configurations are iteratively examined in a certain manner. Each subsequent configuration is constrained to be chosen randomly within a distance d of the previous configuration. As more and more configurations are tested, the value for d decreases, until converging at a single configuration. Using a large value for d early in the tuning allows configurations to be examined that are far away from a previous configuration. This helps the algorithm

escape from any local minima early on. The implementation of Simulated Annealing added to the control point tuning framework, is a slight variation on the traditional simulated annealing scheme in that each new configuration must be within a distance d of the best known configuration instead of a distance d of the previous configuration. Simulated Annealing is useful when the static objective function over the parameter space contains multiple local minima, but finding a global minima is desired, but an exhaustive search is too expensive.

The `+CPSimulAnneal` command line argument enables this simulated annealing scheme.

6.3.4 Nelder-Mead Simplex Method

The Nelder-Mead Simplex Method for function minimization was described in 1965 [28]. The method iteratively moves a simplex comprised of $n+1$ points through the n dimensional parameter space, eventually contracting all the points toward a minima. Three basic operations are applied to the simplex to move it through the parameter space. The most important operation, *reflection*, examines a point found by reflecting the worst performing point in the simplex across the centroid of the other points in the simplex. Once the new point is evaluated, it might replace the old worst point. A second operation, called *expansion*, further expands a reflected point if the reflected point was a good choice. The final operation, *contraction*, may move all points in the simplex towards a previously computed centroid. To construct the initial simplex, $n + 1$ configurations must be evaluated. Then, the reflection and expansion operations each require evaluating one new configuration, while the contraction operation requires $n + 2$ configuration evaluations. For large numbers of control points, many initial configurations will need to be tested before the simplex is moved towards the local minima. This simplex algorithm is useful in a many-dimensional parameter space but only when the objective function is differentiable.

The implementation of the Nelder-Mead Simplex Method within the control point framework largely follows the algorithm proposed in [28], with two important differences. First, the implementation constrains the parameter space to the discrete integer values supported by the available

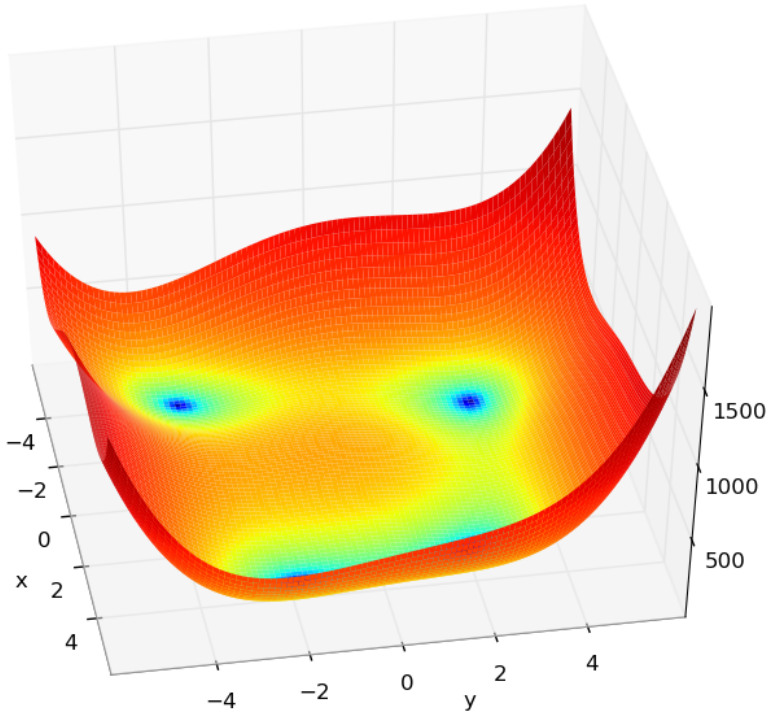


Figure 6.4: The Himmelblau function used for testing the Nelder-Mead Simplex Method in two dimensions.

control points, whereas the original algorithm works over unbounded real valued n-dimensional spaces ($\mathbb{R}^n$). Second, the implementation does not contain any quitting criteria. The algorithm in [28] does not proscribe the initial simplex, so in this implementation, the points in the initial simplex are randomly chosen from the whole parameter space.

To illustrate how the Nelder-Mead simplex method works, and to test our implementation, a simple program was created. The program performs fake computations that take an amount of time proportional to the Himmelblau function over a range of 2 control point values. The control points each vary within the range $[0, 100]$, comprising a configuration space of size 10000. The Himmelblau function, frequently used for analyzing optimization schemes is $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$. The control point values X and Y are each linearly mapped from their range $[0, 100]$ to the x and y values in a range of $[-6, 6]$. Within the range $[-6, 6] \times [-6, 6]$, the Himmelblau function contains four local minima, as can be seen in figure 6.4.

Figure 6.5 shows that after about 30 phases, the algorithm has converged

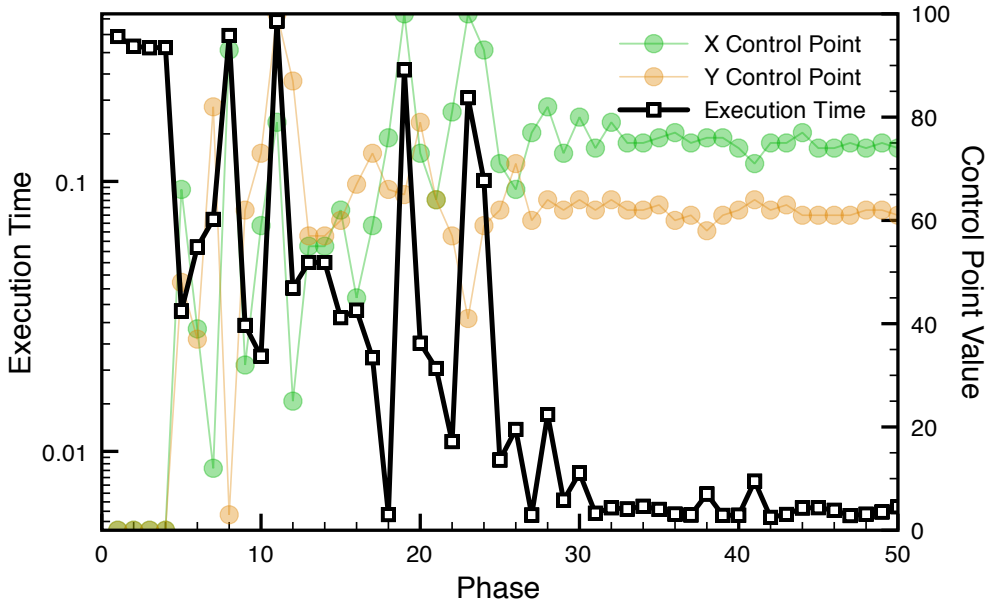


Figure 6.5: The performance of an example application tuned using the Nelder-Mead Simplex algorithm. The program performs synthetic amounts of work proportional to the 2-dimensional Himmelblau function.

upon a neighborhood of configurations with low execution time.

To illustrate how the Nelder-Mead Simplex algorithm works in higher dimensions, we consider synthetic test programs with more than 2 control points. The first has three control points, and it performs work proportional to $f(x, y, Z) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 + (Z - 50)^2$. This program is identical to the earlier program, except it has an additional control point Z that affects the runtime quadratically. The optimal performing case should occur at a configuration where (x, y) is one of the minima of the Himmelblau function, and where $z = 50$. The results from a run of the program tuned using the Nelder-Mead Simplex algorithm with a random initial simplex configuration is shown in figure 6.6. As would be expected, the additional Z control point is roughly 50.

To illustrate how the Nelder-Mead Simplex algorithm works for dimensions higher than 3, a test program was created that can have an arbitrary number of control points. It performs a synthetic amount of work proportional to the function $f(x_1, x_2, x_3, ...) = (x_1 - 25)^2 + (x_2 - 75)^2 + (x_3 - 25)^2 + \ldots$, where the control point values are $x_1, x_2, x_3 \ldots$. Hence the amount of work is minimized and performance is maximized when half of the control point

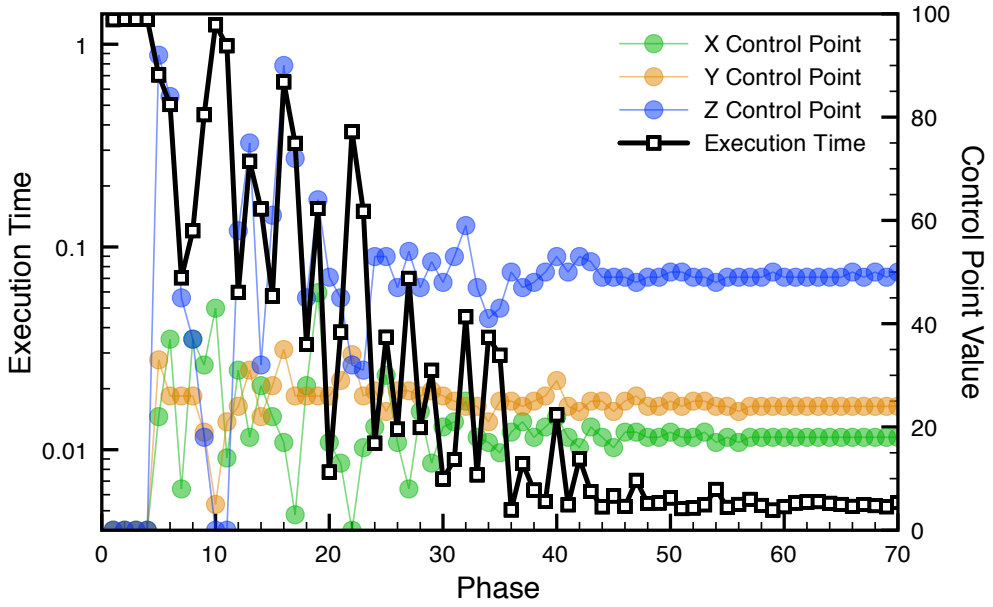


Figure 6.6: The performance of an example application with 3 control points tuned with the Nelder-Mead Simplex algorithm.

values are 25 while the other half are 75. The range of acceptable values for each control point is $[0, 100]$. Figure 6.7 shows the values for all 10 control points as a program is run with the Nelder-Mead Simplex algorithm. The control points generally separate and group together near their optimal values, although there is some variation in the values. It took about 300 configuration evaluations before the performance stopped improving much.

The `+CPSimplex` command line argument enables this Nelder-Mead Simplex tuning algorithm.

6.4 Guided Steering of Control Point Values

Although the values for control points can be adjusted in the traditional, direct-search autotuning approach by the new tuning framework, the novel possibilities investigated by this thesis involve the use of observations of a running program to determine the direction each tunable knob ought to be turned.

The tuning framework uses a mechanism for describing specific behavioral effects that are expected to occur as a control point knob is adjusted. The

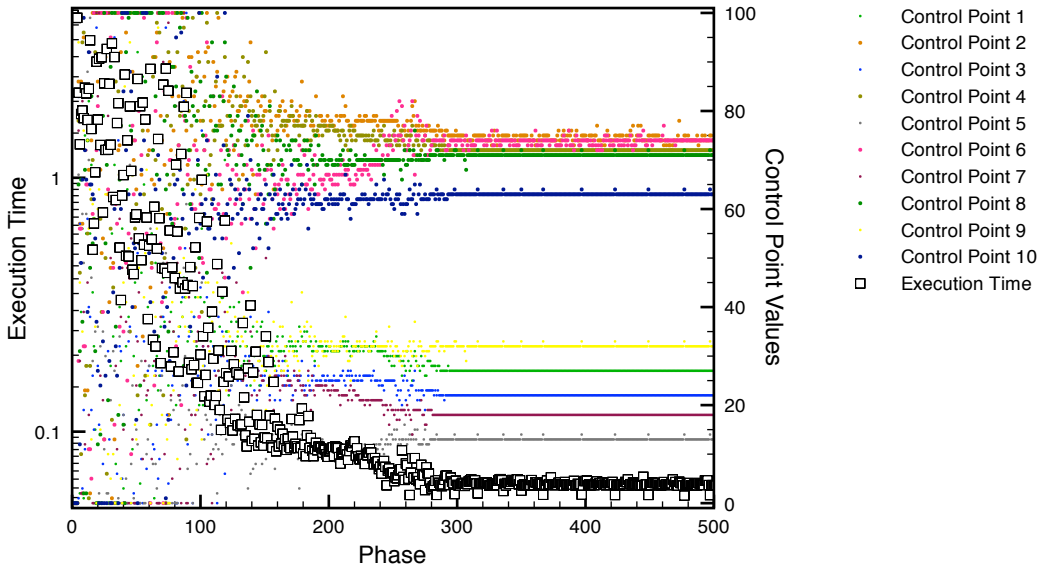


Figure 6.7: The performance of an example application with 10 control points tuned with the Nelder-Mead Simplex algorithm.

application can therefore express the behavioral effects without knowing anything about the specific types of measurements or tuning schemes used in the tuning framework. These behavioral effects include the following among others that are described elsewhere in this thesis:

- Amount of available parallelism[5]
- Memory consumption
- Amount of work to offload to accelerator devices

When the application specifies that a control point affects one of these characteristics, it also specifies the direction of the effect. That is, the application specifies that a control point either increases or decreases the effect. Some of these behavioral effects could be deduced from runtime observations, but there would be additional costs and difficulties in automatically deducing the effects. The intelligence in the tuning framework can therefore use the measured characteristics of the program and select control points that can possibly improve performance. Then one or more of these selected control points are adjusted in the appropriate direction. By

[5]The amount of available parallelism encompasses more than just the grain size for the program. It might depend upon the mapping of activities to processors or the dependencies between activities.

having a knowledge base of the effects of the available control points, the dimension of the tuning space can possibly be reduced while the number of possible configurations in each selected dimension is also reduced. Although the greatest benefits are expected to arise when many control points are used within a single program, this dissertation examines the relationships between measurements and effects for different control points one at a time.

Chapters 7, 8, 9, 10, and 11 describe various tuning algorithms that use measurements to guide the steering of the tunable parameters.

6.5 Combining Multiple Tuning Schemes

In applications with multiple control points, it is likely that multiple tuning algorithms will discover or produce different planned control point configurations. Thus it is necessary to choose between these multiple candidate configurations. The control point framework currently allows all enabled tuning schemes to produce candidate planned configurations. Then one of the resulting schemes is chosen at random from the set of all produced candidates for use in the next phase.

This dissertation focuses mostly on the individual characteristics of multiple independent tuning algorithms for different types of control points. In the future, continuing research will expand this currently simple random choice between the candidates.

6.5.1 Alternative Methods for Combining Multiple Tuning Schemes

A few straightforward techniques could be used to combine the generated plans from multiple tuning schemes:

1. If each tuning algorithm predicts the benefit of its prediction, then a weighted decision could be made, or the predicted best plan could be used.

2. For long-running programs, certain types of control points could be adjusted in a prescribed order.

3. The difference vectors between the previous configuration and each new new generated plan could be combined using various vector operations, such as a sum or a normalized product.

6.6 Summary

This chapter described an API for how control points could be exposed in applications and how this interface supports dynamic tuning in response to measured characteristics of the running application. The chapter also described a concrete implementation of a control point tuning framework that has been been created within the Charm++ runtime system. Various implementation details were discussed to elucidate the necessary runtime system capabilities were a control point infrastructure to be implemented in a different parallel language or system.

CHAPTER 7

Control Point for Divide & Conquer Grain Size

7.1 Application Overview

This chapter describes the automatic adaptation of a divide & conquer parallel program through a control point that determines the grain size of the sequential tasks. In divide & conquer parallel programs, a task is decomposed, often recursively, into subtasks that can be solved independently.

A classic example of divide & conquer parallelism in the literature is the recursive computation of the Fibonacci numbers [29, 30, 31]. This section describes the use of a control point in determining the grain size of the sequential tasks in a parallel recursive Fibonacci program. In divide & conquer parallel programs, a task is decomposed, often recursively, into subtasks that can be solved independently. The task of computing F_n in parallel can be performed by computing in parallel two independent sub-tasks, namely computing F_{n-1} and F_{n-2} and summing their results. Computing F_{n-1} and F_{n-2} can each in turn be computed by dividing into two sub-tasks, or could be computed using the sequential algorithm as shown in figure 7.1. This algorithm explicitly produces parallelism as it proceeds, rather than relying upon an alternative work-stealing approach whereby parallelism is extracted by idle processors.

Some text and data in this chapter were produced by Jonathan Lifflander

if $n \leq T$ **then**
  compute F_n sequentially
  Return F_n
else
  spawn 2 chares that separately compute F_{n-1} and F_{n-2}
  Return $F_{n-1} + F_{n-2}$
end if

Figure 7.1: Parallel algorithm for computing the Fibonacci number F_n

Although this algorithm is a poor way to compute the n^{th} Fibonacci number, the program is a useful representative in the class of divide & conquer applications. The choice of a threshold value T, at which point the sequential algorithm is used instead of the parallel algorithm, must be chosen carefully in order to maximize performance of the algorithm. A control point is used to adjust this value T between successive Fibonacci computations. The program exposes knowledge about its behavior as its control point is varied. Specifically, the program specifies that increasing the control point value decreases the available parallelism. It is expected that if T were too small, too many chares would be spawned. Too many chares will potentially hinder performance because there are extra messaging and task creation costs. If T were too large, then too few chares would be spawned to adequately balance the load across all processors for the duration of the computation.

In the past, researchers have developed numerous methods to automatically find the appropriate amounts of parallelism in divide & conquer dynamic task-based parallel programs that run on shared memory systems. For example, one paper describes an "adaptive cut-off" method that profiles the first 100 OpenMP tasks spawned at different levels in the recursive tree to determine the actual grain size that is expected for each level of the tree [32]. Once the profiling is done, the depth threshold is then chosen for the remainder of the program to produce tasks with grain size around one millisecond. The one millisecond value "was obtained through microbenchmarking of task creation" [32]. An alternative heuristic is proposed in this chapter that doesn't require direct measurements of grain size and it is intended to improve the performance in a distributed memory multi-node system.

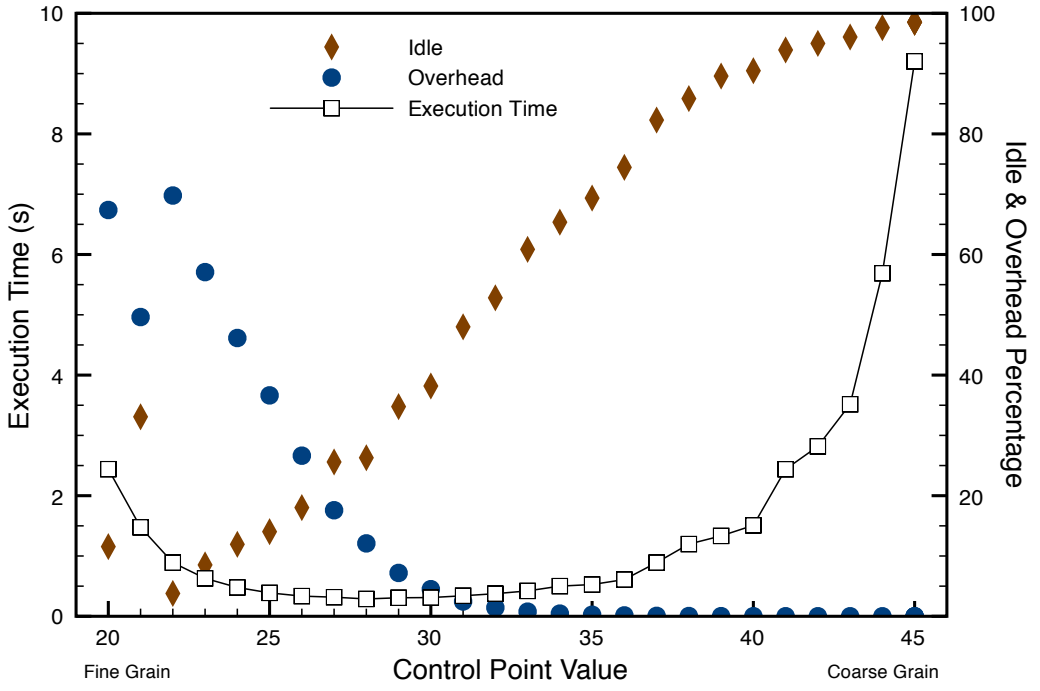


Figure 7.2: Computing F_{45} on 110 processor cores of a Cray XT5 system Kraken.

7.2 Adding a Grain Size Control Point

The Charm++ Fibonacci program contains a threshold T that determines how many worker chares are spawned. Specifically, 2^{n-T} chares are spawned as leaves in a binary tree. By default, in our implementation, each chare is created on a random processor. Alternatively, Charm++ provides various *seed load balancers* that could have been used instead to dynamically balance the chares across the processors [5].

When specifying the effects of the control point, we expose the knowledge that we have about the control point. We specify that increasing the control point decreases concurrency.

7.3 Tuning Between Successive Fibonacci Computations

Let T be a threshold such that F_i is computed using a sequential algorithm when $i \leq T$ or is computed by further dividing it into two parallel sub-tasks

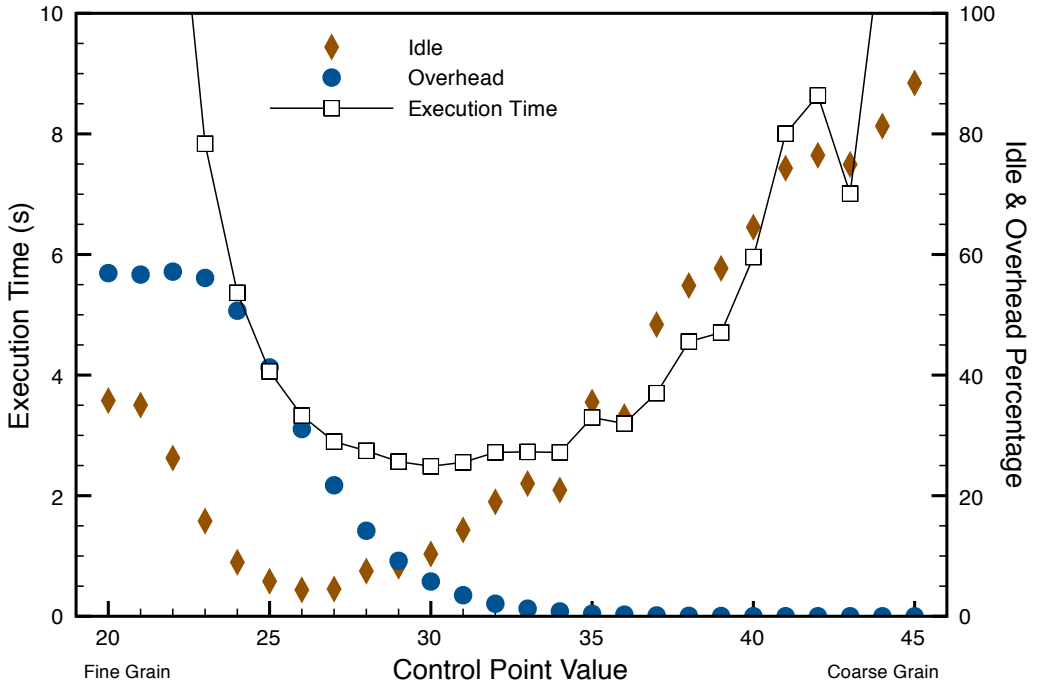


Figure 7.3: Computing F_{45} on 14 processors (2 nodes) of the NCSA Xeon/Infiniband cluster Abe.

when $i > T$. T must be chosen carefully in order to maximize performance of the algorithm, and hence is exposed as a control point by the program. The control point can be varied between multiple subsequent Fibonacci computations. The program exposes knowledge about the behavior of its control point.

Figure 7.2 shows the performance of the program as all possible control point values within a certain range are evaluated when computing F_{45} on 110 processor cores (10 nodes) of the Cray XT5 system Kraken at NICS. It can be seen that the optimal performance is achieved as expected in a valley while reduced performance occurs for configurations with the threshold T too high or too low. Figure 7.3 shows the similar behavior occurring when computing F_{45} on 14 processor cores (2 nodes) of the NCSA Abe cluster. Although the overall plots are similar, the optimal configurations for the same problem are different for these two different parallel machines. In the first case, the optimal control point value is 28 while it is 30 for the other case.

Additionally, it can be seen in these figures that the measured average idle time across all processors increases as the control point value T increases. This is expected because the idle time occurs when there is not enough

available parallelism to provide an adequate balance of work both across processors and in time. The second important measured observation is that as the threshold T decreases, the measured overhead generally increases. The increased overhead is expected because there are increased costs of messaging and scheduling as exponentially more worker chares are created to solve the same problem.

To tune the program, it is important to steer the control point value toward the optimal program performance. Regions of poor performance within the parameter space occur concurrently with high measurements of idle time or overhead time.

One such simple yet powerful tuning scheme for this program has been implemented. It turns the knob in the direction that reduces the available parallelism whenever $Time_{idle} < Time_{overhead}$. The available parallelism in this case ought to be reduced because the largest observed problem is the large value of $Time_{overhead}$. Conversely, the knob increases the available parallelism whenever $Time_{idle} > Time_{overhead}$, which indicates that the largest observed problem is the large value of $Time_{idle}$. The best performing configurations experimentally are observed to be near the point where $Time_{idle} = Time_{overhead}$. Thus, any performance steering scheme that works well with this program would need to converge to a configuration close to this heuristic.

The necessary direction to turn the knob is encapsulated in the control point (i.e. it is specified by the program). The tuning framework in the runtime system can measure $Time_{idle}$ and $Time_{overhead}$ without any modifications or help from the program.

Using this heuristic, the threshold T can be adjusted in the direction that steers the program towards better performance. The exact trajectory taken by the program depends upon its initial configuration, and how fast the threshold is adjusted. In the case shown in figure 7.2, the program could start with any initial threshold from 20 to 45, and the heuristic would lead to a tuning that converges at 26 or 27.

7.4 Tuning Within One Fibonacci Computation

Section 7.3 found that repeated Fibonacci computations could be tuned using a heuristic based upon the relationship of two measured quantities, $Time_{idle}$ and $Time_{overhead}$. Because the heuristic is a function only of two measured quantities, neither of which is the application's reported performance, the heuristic could possibly be used to tune the program within a single Fibonacci computation. This section describes how such tuning is achievable and how the dynamically tuned program performs better than corresponding static configurations most of the time.

To effectively vary the threshold as the program runs, it is important to ensure that varying the threshold actually will affect the program. In this case, the program must expand its computational tree, formed as each task is subdivided, in the correct order. When a tree is expanded in a depth first order, a dynamically varying threshold can result in the left and right sides of the tree being expanded to different depths. If the nodes in the tree are expanded in a breadth first manner, then all nodes in the tree will be expanded at startup to the initial depth threshold. Then the depth can only be increased because the nodes in the tree cannot be un-expanded. For these reasons, the Fibonacci program spawns its tasks using a LIFO scheduling mechanism, hence ensuring that a depth first traversal is used. LIFO is also natural for divide and conquer applications because it reduces memory usage.

As the control point is periodically adjusted, e.g. once every one second, its new value is broadcast to all processors using a new entry method within a *group*. In the current implementation, this new group has been added to the application, although in the future we hope to add native support for globally shared variables with loose synchronization to the Charm++ language [1]. If the necessary type of shared variable were supported by the Charm++ language, the need to add a group and a corresponding entry method would be eliminated.

The program is run in two configurations. The first configuration is a baseline static case where the threshold starts at some initial value, and the value does not change as the computation proceeds. In this baseline program, no instrumentation, gathering of measurements, or tuning analysis

[1] In the past non-scalable types of globally shared variables were supported in Charm++, but these are no longer found in the modern Charm++ language.

is performed. The second configuration of the program dynamically adjusts the tree expansion depth threshold through a control point. The control point is set to some initial value, and the heuristic discovered in section 7.3 is used to adjust the control point value once every 2 seconds. The amount of change in control point value each time is -1, 0, or 1.

Figure 7.4 shows the measured idle time and overhead time for a single computation of F_{56} on 55 processors (5 nodes) of the Cray XT5 Jaguar system. In this case the program is instrumented to gather idle time and overhead measurements to be displayed in the figure. The threshold is held constant at 25 throughout the whole computation that takes 309 seconds. Figure 7.5 shows the same computation being performed while the control point value starts at 25 again, but is adjusted based upon the heuristic. The control point value increases from 25 to 32 before shrinking back to 29. The execution time is reduced to 59 seconds, an improvement of 81%. If compared to the static case without the unnecessary and in this case expensive instrumentation, the dynamic case is still 16% better. It can be seen that the idle times and overhead times are both greatly reduced by adjusting the control point. For some other initial configurations, the difference is even higher.

The benefits due to dynamically varying the control point value are not restricted just to this one example. Figures 7.6 and 7.7 shows that the dynamic control point tuning almost always achieves better performance than the corresponding static configuration for a wide range of initial thresholds, both on 55 processors (5 nodes) or 220 processors (20 nodes) of the Cray XT5 Jaguar system. On the left side of the figure, where computation starts with a very fine grain configuration, the control points are able to move to coarser grain sizes to improve efficiency. Toward the right sides of these figures the control points start at configurations that result in coarse grain computations, not exposing enough parallelism to the available processors. The control points are able to move to somewhat finer grain configurations, improving performance by about 25% in some cases.

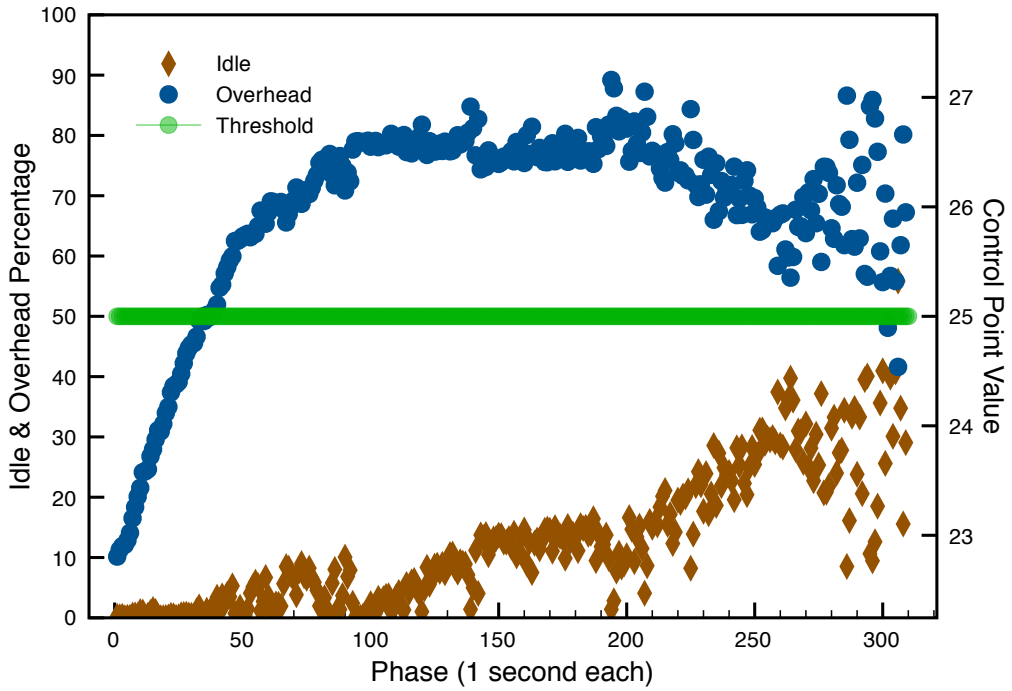


Figure 7.4: Computing F_{56} using a fixed threshold. Initially 10% of the time is overhead which increases as more fine grained work is produced. The whole execution takes 309 seconds.

7.5 Programmer Burden

Figure 7.8 and figure 7.9 list the new code added to the Fibonacci program to add a control point that modifies the threshold that determines whether a sequential algorithm is used or whether parallel subtasks are created. Eight lines of code were added to the Charm++ Interface file for the program, while 26 lines of C++ were added to the program. These 34 total lines of code create the mechanisms responsible for receiving the callback from the tuning framework and broadcasting the new value to all processors. Most of this code would be eliminated if the Charm++ language were to support globally updated variables. Although the code has substantial length compared with the whole program, the programmer effort to produce such code is still small because much of it is straightforward boiler-plate code that would be easily written by a Charm++ programmer.

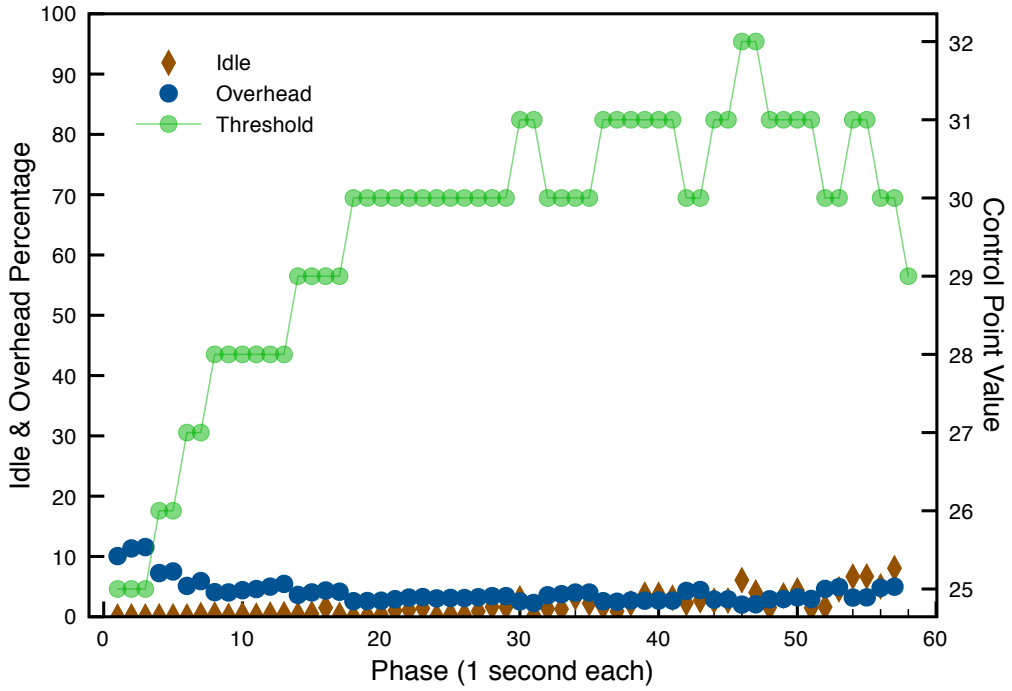


Figure 7.5: Computing F_{56} using a dynamically varying threshold. The threshold starts at the same value as in figure 7.4, but the heuristic causes it to be increased because the overhead time is initially larger than the idle time.

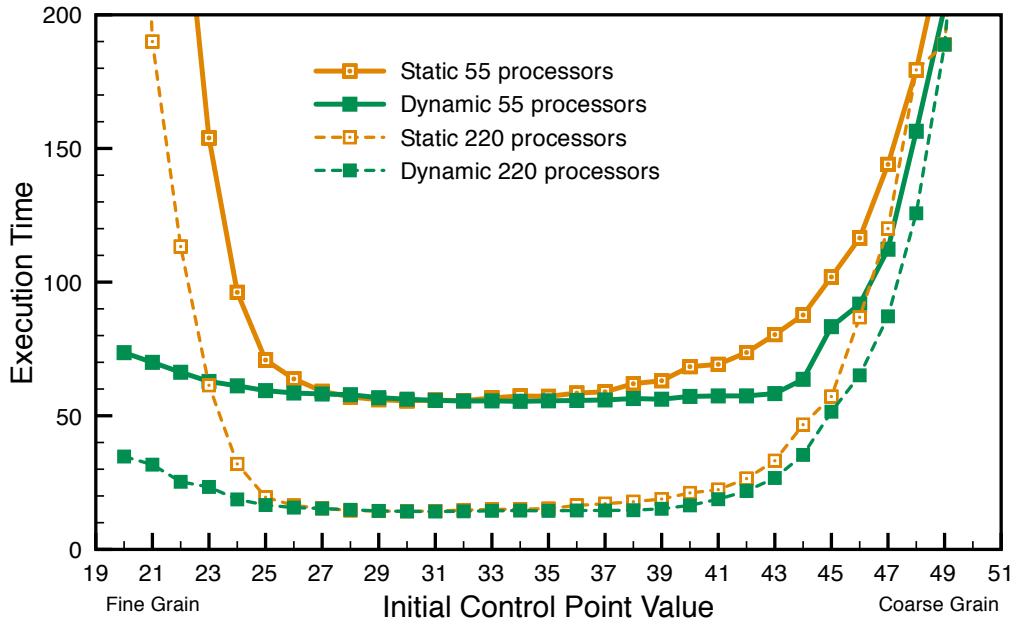


Figure 7.6: Performance of a Fibonacci program (computing F_{56}) that dynamically varying the control point values and a static baseline program with no instrumentation on 55 or 220 processors.

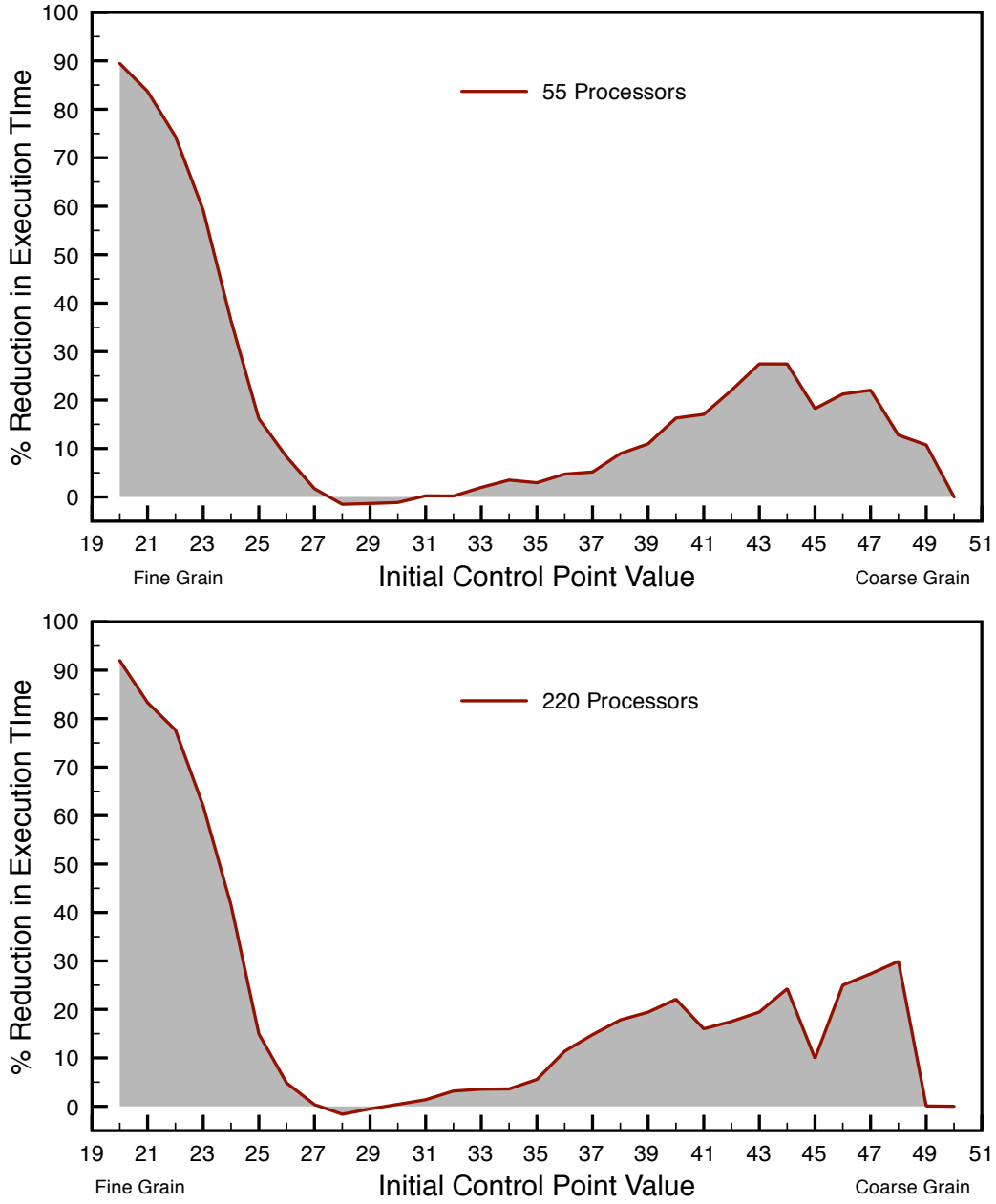


Figure 7.7: The benefit due to dynamically varying the control point values over a baseline program with a static threshold T. Runs were performed over a range of initial control point (or threshold T) values while computing F_{56} on 55 and 220 processors. These represent the same data seen in figure 7.6.

Read-only variables:

```
readonly int threshold;
readonly CProxy_BThreshold threshGroup;
```

New message type:

```
message ThreshMsg;
```

New entry method for notification of control point changes:

```
entry void controlChange(controlPointMsg* msg);
```

New group for broadcasting threshold updates:

```
group BThreshold {
  entry BThreshold();
  entry [expedited] void changeThreshold(ThreshMsg *msg);
};
```

Figure 7.8: New Charm++ Interface code added to Fibonacci program to add a control point.

7.6 Future Work

The use of a naive Fibonacci algorithm to illustrate the use of control points for divide & conquer algorithms is not practical, but rather is pedagogical. Thus in the future, grain size control points ought to be added to varying types of programs that exhibit divide & conquer computations. Thes programs could include state space searches, sorting, nested dissection, Delaunay mesh refinement, numerical integration, and even LU factorizations.

Additionally, in the future it would be beneficial to compare the approach presented in this chapter to the work-stealing approach which has proven to be useful for some applications on distributed memory parallel machines [33]. The approach in this chapter produces parallelism as the computation proceeds while the work stealing approach allows processors to search for work to perform when they are idle.

In our steering approach to tuning control point values, control points are instantaneously adjusted by the minimum amount possible each phase. For long running programs, the cost for the overly slow and conservative speed at which adjustements are made will be amortized away once a local optimal value is found. Other techniques such as proportional controller feedback could be used to allow control points to be quickly adjusted when very large

C++ Declaration:

```
#include <controlPoints.h>
int threshold;
CProxy_BThreshold threshGroup; /*readonly*/
```

At startup:

```
ControlPoint::EffectDecrease::Concurrency("threshold");
threshold = controlPoint("threshold", THRESH_MIN, THRESH_MAX)
    ;
threshGroup = CProxy_BThreshold::ckNew();
CkCallback cb(CkIndex_Main::controlChange(NULL), mainProxy);
registerCPChangeCallback(cb, true);
```

New entry method used as a callback that receives notification of new control point values:

```
void controlChange(controlPointMsg* msg) {
  controlPointTimingStamp();
  threshold2 = controlPoint("threshold", THRESH_MIN,
     THRESH_MAX);
  ThreshMsg *msg = new ThreshMsg(threshold);
  threshGroup.changeThreshold(msg);
}
```

Definition of broadcast threshold entry method and group:

```
class BThreshold : public CBase_BThreshold {
 public:
   BThreshold() {}
   void changeThreshold(ThreshMsg *msg) {
     threshold = msg->threshold;
   }
};
```

Definition of threshold message:

```
class ThreshMsg : public CMessage_ThreshMsg {
  public:
    int threshold;
    ThreshMsg(int t) : threshold(t) {}
};
```

Figure 7.9: New C++ code added to Fibonacci program to add a control point.

idle time or overhead time measurements are observed.

7.7 Summary

This chapter described how a control point could be used to adjust the computational grain size of a divide-and-conquer style parallel program, using a naive Fibonacci program as a case study. A heuristic was discovered that allowed the automatic tuning of the control point value without any application provide performance metrics, even within a single Fibonacci calculation. The resulting performance of the program improves in almost all cases, sometimes by as much as 90% when poor initial values for the control point are used. This case study showed that automatic tuning of a grain-size parameter in divide-and-conquer parallel programs is both possible and useful.

CHAPTER 8

Control Point for GPU Offload Ratio

In recent times, heterogeneous systems with two or more different types of processors have become increasingly popular. Different types of processors have different performance characteristics. A GPU for example might be ten or one hundred times faster at executing a computational kernel than would a standard CPU core. A control point could be used to adjust the balance of work between the two types of processors.

8.1 Application Overview

The application described in this chapter simulates the physical behaviors of functionally graded materials using an explicit finite element method. In the program, a physical object is represented by a 3-D tetrahedral mesh. Because the material properties vary throughout the material, each tetrahedron will contain numerical data describing its behavior, namely how the tetrahedron responds to forces exerted upon the tetrahedron. The simulation of non-homogeneous materials, is an area of active research [34, 35, 36].

The application itself uses the ParFUM framework [37, 38], which is built upon Charm++. The ParFUM framework provides support for partitioning a mesh and distributing the partitions across the parallel machine. Each mesh partition is associated with a migratable user-level thread. The ParFUM framework also provides the communication mechanisms that allow values to be easily synchronized across the partition boundaries during each application timestep. The ParFUM framework also provides MPI style

communication between the user-level threads, so the program can easily incorporate any additional synchronization or communication required.

To make use of GPU accelerators such as the NVIDIA Tesla, the computational kernels were ported to the CUDA language [39, 40]. CUDA allows kernels (written in C with some restrictions) to be executed on a GPU. GPUs are useful because they provide high floating-point performance, and they provide very fast bandwidth to the device's memory.

The program can run on clusters of compute nodes, each with one or more attached GPU devices. The numerical scheme used in the program involves repeated explicit update steps that update values over the tetrahedra in the mesh. The application simulates non-homogeneous materials, so each tetrahedral element contains an unusually large amount of data describing its material properties and responses to stress, so memory bandwidth becomes the main bottleneck in the application.

The application is further described in [41]. The remainder of this chapter describes the issues associated with adding a control point to the application that automatically adjusts the fraction of the work offloaded to a GPU.

8.2 Adding an Accelerator Offload Control Point

In the existing code for the program, the mapping of virtual processors, i.e. mesh partitions, onto physical processors was specified on the command line when running the program. Whether each VP executes on a CPU or on a GPU was specified in addition to the mapping onto processors. When the program starts up, any VP that is to run on the GPU executes a function that creates its data structures on the GPU and copies all necessary mesh data into these data structures. Thus the code to move a partition from the CPU to the GPU already existed prior to adding a control point. However, to add a control point that adjusts the number of partitions on each GPU, and correspondingly the processor on which each VP executes, two main mechanisms were added to the program. The first mechanism implements the necessary serialization of a VP in order to support its migration to a different processor. The second mechanism allows a VP whose execution occurs on the GPU to be switched to execute on a CPU. This second mechanism is simply the inverse of the existing code that copies data structures onto the GPU's

device memory. This second mechanism was mostly implemented within a library which was not part of the program itself.

The final modification to the program was to add a control point and construct a new mapping of VPs to processors based on the control point value. The total lines of code added to the program, excluding comments, is shown in table 8.1.

8.3 Tuning Scheme

A control point could be used to adjust the balance of work between the two types of processors. If too much work is offloaded to the GPU, the CPU cores will become idle, while if too much work is kept on the CPU cores, then the GPU will be underutilized and the CPU cores will remain fully utilized. Hence, it should be possible to measure the time the CPU cores are idle, and use this information to steer the control point value to the optimal work balance between CPU and GPU.

This steering approach requires no model of performance relating the speed of the GPU to the CPU, nor does it require any instrumentation of the GPU itself. It simply uses CPU utilization measurements to steer the performance to the optimal balance between work on the GPU and CPU.

Figures 8.1 and 8.2 show how the program performs over a range of values for the control point. The control point specifies how many mesh partitions are assigned to and executed by each GPU. The experimental platform for both examples uses two compute nodes of the NCSA Lincoln Cluster, each containing two Intel quad core CPUs and two NVIDIA Tesla GPUs. In the example shown in figure 8.1 only one of the GPU devices is used, whereas both GPU devices are used for the example shown in figure 8.2. In both cases one of the eight CPU cores on each node is left unoccupied to reduce operating system interference.

The trends seen in the figures are generally as expected: as more work is offloaded to the GPU accelerators, the program speeds up for a while until at some point the performance suffers when too much work is offloaded to the GPU accelerators. There are two expected contributions to the degraded performance when too much work is offloaded to the accelerators. The first contribution to the performance degradation is the fact that the CPU cores

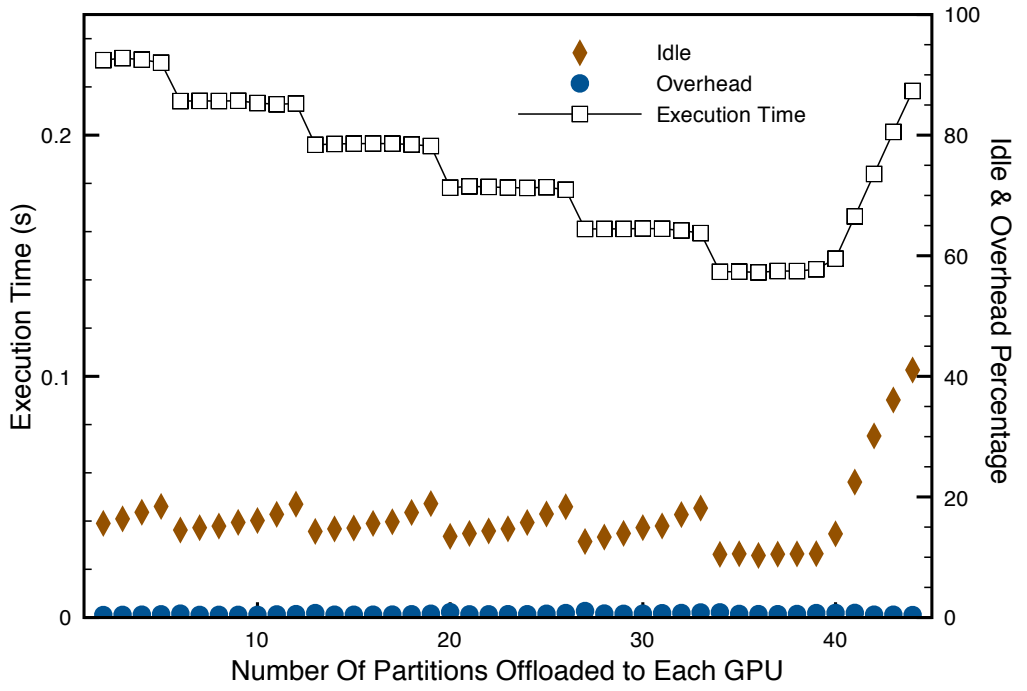


Figure 8.1: Varying the amount of work offloaded to each of the one GPU accelerators per node.

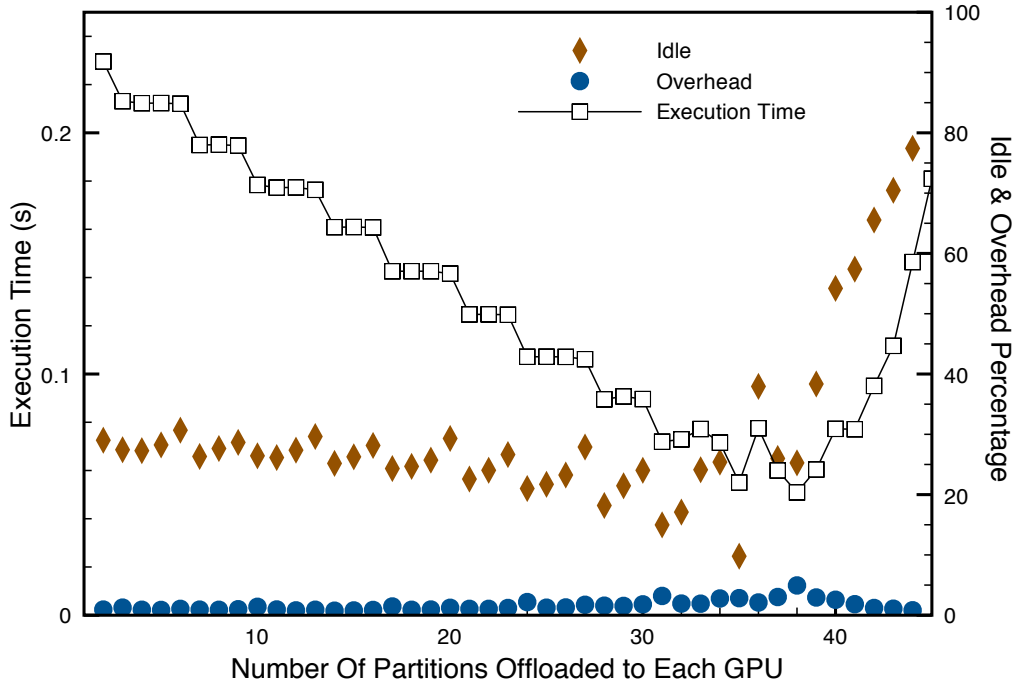


Figure 8.2: Varying the amount of work offloaded to each of the two GPU accelerators per node.

Modification	Lines of code (excluding comments)
Add control point	6
Compute and broadcast new mapping of VPs to PEs	25
Migrate each VP to new processor	5
Switch a VP from executing on a GPU to executing on CPU	7
Total	43

Table 8.1: Lines of code required to add control point to the structural dynamics finite element program.

run out of work and become idle. The second is that communication costs for the CPU cores handling the GPU increase as more and more mesh partitions are shifted there.

An automatic tuning scheme is implemented in the Charm++ control point tuning framework for this type of control point. It starts with a low amount of work offloaded to the GPU, and increases the amount until the idle time has significantly increased. If the idle time starts to increase significantly from its previous measurement, then the optimal performing configuration is nearby. Due to the seemingly noisy behavior in figure 8.2, it may be advantageous to also search a small number of nearby configurations after steering the performance toward the near optimal point where there are multiple local minima. This would ensure that the global minima is reached.

8.4 Programmer Burden

Table 8.1 shows that 43 lines of application code were added to expose a control point in the program described earlier in this chapter. These modifications to the program performed the reconfiguration of the program when a new control point value was obtained. After a new control point value is obtained the virtual processors, i.e. MPI ranks, are redistributed to the available physical processors. This mapping of VPs to processors is based on the control point value. Once the virtual processors have been redistributed, some of them must be reconfigured to use the GPU instead of the CPU for their computations. Prior to adding the control point, the virtual processors

were not capable of switching their computations from the GPU to the CPU: they were only capable of switching their computations from the CPU onto the GPU. The switching in this second direction required new code, but mostly the code was a mirror of the existing code already present in the program.

8.5 Summary

This chapter has shown that a control point that adjusts the amount of work offloaded to an accelerator device such as a GPU can be tuned using measurements of the CPU utilization. There is no need to actually measure the occupancy or utilization of the GPU device. Adding this type of control point to a program is simple if the program already supports the moving of work to and from the accelerator device.

CHAPTER 9

Control Point for Load Balancing Period

Load balancing is an important task in many types of parallel programs. The work must be partitioned across the available processors in a balanced manner in order to obtain good parallel efficiencies on large systems. Historically, parameters related to load balancing in HPC applications are manually configured by the creator of an application based on some experimental runs, as is done in various Charm++ applications [42, 43], or are prescribed by a user when launching the application. Even in other parallel programming systems it is common to specify the load balancing frequency as a runtime argument to the program [44]. Many HPC style load balancers such as those in Zoltan are very expensive, as the entire problem domain is repartitioned every load balancing step. Therefore the load balancers are used infrequently, commonly just at startup. Other load balancing systems such as PLUM perform dynamic load balancing globally across processors, investigating ways of minimizing the cost of load balancing, but without addressing the issue of how frequently to perform the load balancing [45].

This chapter discusses how to automatically determine the frequency at which dynamic load balancing operations ought to occur. An example program is modified to expose a control point that controls the frequency or period at which load balancing takes place. Two types of methods for automatically adjusting the load balancing period are proposed. One method, described in section 9.4, computes an approximation of the benefit of a load balancing operation and adjusts the period based on

whether the load balancing operation was beneficial. The second method, described in section 9.5, calculates the load balancing period using a closed-form analytical formula which is optimal under certain assumptions of an application's behavior.

This chapter concludes with an analysis of the resulting performance of a program when four different methods for choosing the load balancing periods are used dynamically.

9.1 Application Overview

The example program simulates a 3-D volume discretized into tetrahedra. As the simulation progresses, each tetrahedral element is considered to have one of two material properties: *elastic* or *plastic*. Different physics routines are used for the element depending on which material property is active, and hence the amount of computation performed for each element varies. The variation in computation time over the mesh changes as the simulation progresses. Thus it is important to dynamically load balance the program, as has been shown in previous research efforts that evaluated dynamic load balancing techniques [42, 43].

This example program, called Fractography3D, was written primarily by Scot Breitenfeld, Professor Philippe H. Geubelle, and Orion Lawlor. It consists of about 6000 lines of Fortran code that uses the FEM framework (later renamed ParFUM) to partition and store data on the parallel tetrahedral mesh [46, 37]. The FEM framework is built upon Adaptive MPI (AMPI), and hence on top of Charm++, which provides two useful interfaces to the application: the FEM framework interface and an MPI interface.

9.2 Dynamic Load Balancing

The Charm++ runtime system provides instrumentation-based dynamic load balancing. This load balancing system can be applied to AMPI applications by migrating the virtual MPI processors (VPs) among the physical processors. The time spent in computation by all virtual MPI processors is automatically measured for use in a load balancing algorithm.

The principle of persistence, which is relevant for many scientific and engineering applications, suggests that the recent measured performance characteristics are useful for this task of load balancing because the future behavior of the program should be similar to the measured recent behavior.

It has been shown in the past that the performance of Fractography3D improves when dynamic load balancing is used [42, 43]. Prior to this work, the users of the Charm++ load balancing framework were required to specify the frequency at which the load balancing operations occur, either with a command line argument or within the application code. Poor choices of load balancing frequency could result in poor performance. This chapter describes an automatic method for dynamically adjusting the frequency at which load balancing operations are performed.

Figure 9.1 demonstrates that poor decisions for the load balancing period result in poor application performance. The leftmost data point represents a run where the GreedyLB load balancer is invoked every 5 application timesteps while the rightmost data point represents a run where the same load balancer is invoked every 7000 steps. All these runs were performed on 200 processor cores of an IBM Power cluster named Blue Print at the NCSA[1].

In figure 9.1 it can be seen that the performance of the program varies depending upon the load balancing period. The optimal configuration appears to be around 1000. For periods less than 1000, the overheads of load balancing become larger, while for periods greater than 1000, the decreased load balance reduces performance. A scientist or engineer running this program, however, would likely not want to run the program many times with various load balancing configurations just to find the best period. Hence this chapter proposes an automatic solution for dynamically varying the load balancing period for the program. Ultimately the automatic scheme provides uniform good performance when any value is chosen initially for the load balancing period. Furthermore, automatic tuning of this parameter can provide benefits in the case where the application's load balance characteristics change over time.

Ultimately, a load balancing period should be chosen for each load balancing operation to minimize the total application time. For long-running

[1]The application is compiled with no optimization flags because it crashes due to an unknown problem when the `-O3` optimization flag is used.

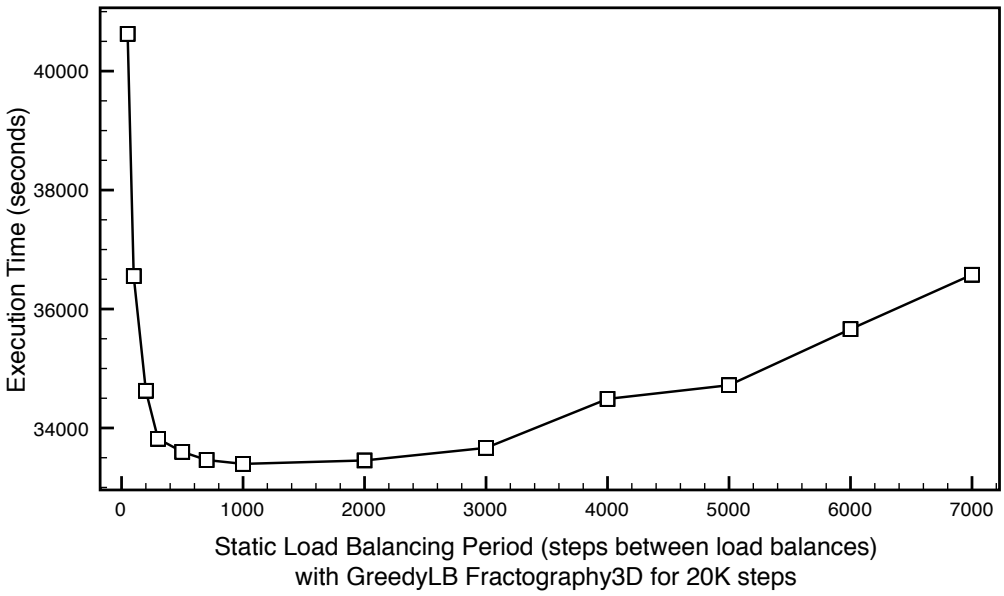


Figure 9.1: Performance of Fractography3D for separate runs each using a different load balancing period.

applications, this is equivalent to minimizing the total time spent by a large number of steps.

9.3 Adding a Load Balancing Period Control Point

One control point has been added to the Fractography3D program. This control point determines the number of application timesteps between successive calls to the load balancing library, which already were included in the application albeit at fixed timesteps.

A few modifications to the program were required to add the control point and respect the semantics required by the API. The modifications to the program are listed in figure 9.2. They include a call that keeps the first VP from migrating away from processor 0. The tuning framework is instructed not to advance the phases on its own. A call to retrieve the control point value and to advance to the next phase were added to the timestep loop in the program just after each load balancing operation has completed. Because the control point framework provides the new values only on processor 0, the new control point value is broadcast to all VPs. Finally, the application provides a notification to the tuning framework every step. This allows the

Declarations:

```
INCLUDE 'controlPointsf.h'
INTEGER :: newLdbPeriod
```

Prior to timestep loop:

```
IF(myid .EQ. 1) THEN
    CALL CPEffectsIncrease_LoadBalancingPeriod()
    CALL setFrameworkAdvancePhaseF(noAdvancePhase)
    CALL MPI_Setmigratable(MPI_COMM_WORLD, 0)
ENDIF
```

Within the timestep loop:

```
IF(myid .EQ. 1) THEN
    CALL gotoNextPhase()
    newldbperiod = controlPoint(5,20000000)
ENDIF
CALL MPI_BCAST(newldbperiod, 1, MPI_INTEGER, 0,
   MPI_COMM_WORLD, ierr)
nextLdbStep = itstep + newLdbPeriod
```

```
CALL MPI_Barrier(MPI_COMM_WORLD, ierr)
IF(myid == 1) THEN
    CALL controlPointTimingStamp()
ENDIF
CALL MPI_Barrier(MPI_COMM_WORLD, ierr)
```

Figure 9.2: New code added to Fractography3D to add a control point that adjusts the load balancing period. No name is specified for the control point because the fortran interface does not yet support multiple control points in a single program.

framework to determine the execution time for each step. Because there are more than one VP on each processor, due to the multiple possible orderings of their computations within a single step, two barrier calls were added to reduce the variation in the times observed for each step.

9.4 Adjusting the Period Based on Utility

A principled method for analyzing the utility of a load balancing operation is to consider the execution times of steps before, during, and after the load balancing operation to determine whether the load balancing operation is beneficial.

The overhead required for the load balancing operation can be extracted

and compared to the benefit in execution time for the steps following the load balancing operation. In a run of the program, only the actual execution times of steps post load balancing can be measured, not the times for the same steps if the load balancing operation had not been performed. Thus in this analysis, an estimation must be made of the step execution times that would have occurred if load balancing had not been performed.

To predict or estimate these alternative execution times, a model is created to extrapolate the times from before the load balancing operation to the steps following the load balancing operation. Then these predicted execution times are compared to the actual execution time to find the benefits of the load balancing operation. Finally the benefit is compared with the cost of the load balancing operation itself.

The control point tuning framework can make this determination of whether recent load balancing operations were beneficial or detrimental. To automatically steer the load balancing period parameter, if recent load balancing operations were beneficial, the load balancing period is halved. If the recent load balancing operations were deemed to be detrimental to performance, then the load balancing period is doubled. Although the doubling and halving is potentially too coarse grained, it can quickly span a wide range of values. Some Charm++ programs perform load balancing every step (ChaNGa), while others only need to perform load balancing after tens or hundreds of thousands of steps. Even though the doubling is possibly too coarse grained, this chapter shows that in a real application, the achieved application performance is good in spite of this potential problem.

The method for predicting the utility of each load balancing operation requires the prediction of the execution times of the following steps had the load balancing operation not been performed. Three such predictor models are described in sections 9.4.1, 9.4.2, and 9.4.3. Then sections 9.4.4 and 9.4.5 describe specifically how to determine if the load balancing operation is beneficial and correspondingly how to adjust the period.

9.4.1 Constant Predictor Model

The simplest way of predicting the time that would have been spent in the steps following a load balancing operation is to assume that the average

time for a step prior to the load balancing operation would remain constant for the steps after load balancing. If we have n steps prior to the load balancing operation and m steps following the load balancing operation, then the average expected execution time for the steps post load balancing would be $\hat{t} = \frac{\sum_{i=1}^{n} t_i}{n}$, where t_i is the measured time for step i. hence the total execution time expected for the m steps would be $E = m\frac{\sum_{i=1}^{n} t_i}{n}$.

The temporal cost of load balancing is incurred by the step during which load balancing is performed, and depending on the amount of asynchrony in the program, possibly by one or more following steps as well. Thus we can estimate the overhead of the load balancing operation at step b as $L = t_b + t_{b+1} - 2\hat{t}$. This overhead L represents the extra time spent in steps b and $b+1$ when compared to the average time of the preceding n steps. In tightly synchronized programs, only the time for step b will be affected by the load balancing, but to more accurately represent the general case, the implementation of this predictor method in the control point tuning framework examines both steps b and $b+1$ as described above.

9.4.2 Linear Predictor Model

When the load is dynamically changing, then a higher order model could more accurately describe the expected execution times for steps after the load balancing operation. Thus a linear model is proposed in this section. Figure 9.3 displays graphically how this linear prediction model works.

Instead of simply using the average of the n steps prior to a load balancing operation, as is done by the constant predictor model, a line is fit through those n points. Obviously, if $n > 2$ there are many such lines that reasonably approximate the n points and their trend. A least-squares fit is a commonly used fitting method, but it gives larger importance to outlier points proportional to the square of their distances from the resulting line. It is more beneficial in this case to use a line such that the area under the line corresponds to the total time spent by the n steps because the total time for a long series of steps is the ultimate metric of interest. Furthermore, for polynomial fits of degree d, the least squares method costs $O(nd^3)$ instead of the $O(nd)$ methods used in this and the following sections[2]. Thus a least-

[2]if implemented using a $O(n^3)$ solver for a system of n equations.

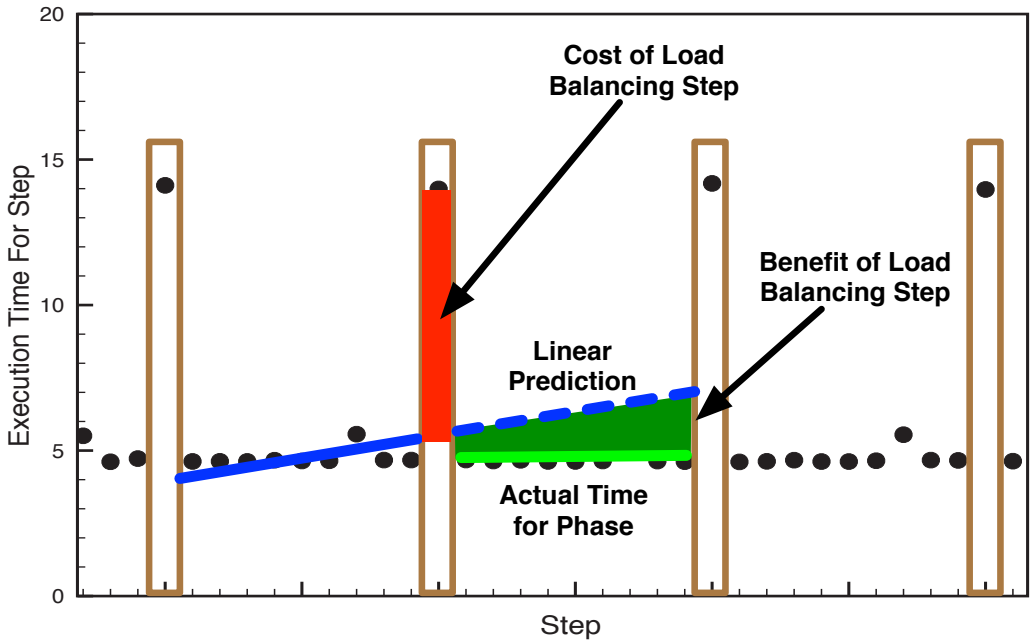


Figure 9.3: The linear prediction model can be used to estimate the times that would have occurred after a load balancing step if it were not performed.

squares fit is not used. The method used is both simpler than the least-squares method, and it better represents the total time spent in the n steps.

The linear prediction model constructs a line that goes through the points $\left(\frac{n}{4}, a_1\right)$ and $\left(\frac{3n}{4}, a_2\right)$, where $a_1 = \frac{\sum_{i=2}^{\lfloor \frac{n}{2}+1 \rfloor} t_i}{\frac{n}{2}-1}$ and $a_2 = \frac{\sum_{i=\lfloor \frac{n}{2}+2 \rfloor}^{n} t_i}{\frac{n}{2}-1}$ are the average execution times for the first half of the steps and the second half of the steps prior to the load balancing operation (excluding 2 steps that are affected by a previous load balancing step).

It is a simple matter of algebra to evaluate the sum of the execution times of the m steps following the load balancing operation. Because the load balancing period may have been adjusted at the load balancing step, it is not necessary that $n = m$. The total expected execution time for the m steps following the load balancing operation is therefore:

$$E = m \cdot \left(\frac{a_1 + a_2}{2} + \frac{(n+m)(a_2 - a_1)}{n} \right).$$

The execution time spent performing the load balancing operation for step b can be approximated by finding how long steps b and $b+1$ take, and subtracting out the portion of that time which is likely to be the actual application work. This amount of work is taken to be the average of the actual

measured steps from $b+2$ until the next load balancing operation, which we call $a' = \frac{\sum_{i=b+2}^{m} t_i}{m-b-1}$. Thus the cost of load balancing can be approximated by

$$L = t_b + t_{b+1} - 2a'.$$

9.4.3 Quadratic Predictor Model

Although it is logical to consider further higher order models, higher order polynomial interpolations do not behave well when predicting execution times far away from the points used to construct the polynomial. Thus it is not expected that higher order curves will be useful when $m \gg n$.

Furthermore, if a parabola is fit through the data points, it will likely be concave downward because the steps just after load balancing will likely be better than the later steps which settle into a poorly balanced configuration with near constant step times. This downward concavity can produce a gross underestimate of the execution times expected post load balancing because the parabola's values become negative, which is a problem especially when $m \gg n$.

Even though it was not expected for the higher order models to be useful, the quadratic model was still implemented in the control point tuning framework.

Just as the linear model was constructed by interpolating two points representative of subsets of the $n-2$ points, a quadratic model could be constructed as a parabola that passes through three points, $\left(\frac{n}{6}, a_1\right)$, $\left(\frac{3n}{6}, a_2\right)$, and $\left(\frac{5n}{6}, a_3\right)$, where a_1, a_2, and a_3 are the average execution times for the first third, middle third, and final third of the points in the range $2 \ldots n$ respectively:

$$\begin{aligned} a_1 &= \frac{\sum_{i=2}^{\lfloor \frac{n-2}{3}+1 \rfloor} t_i}{\frac{n}{3}} \\ a_2 &= \frac{\sum_{i=\lfloor \frac{n-2}{3}+2 \rfloor}^{\lfloor 2\frac{n-2}{3}+1 \rfloor} t_i}{\frac{n}{3}-1} \\ a_3 &= \frac{\sum_{i=\lfloor \frac{n-2}{3}+2 \rfloor}^{n} t_i}{\frac{n}{3}-1}. \end{aligned}$$

To produce the expected execution time for m steps following a load balancing operation assuming the load balancing had not occurred, a finite integral of the area under the parabola, going through the three points can be determined as follows. The area of the curve that is of interest is for the m points following the load balancing operation, namely from $x_1 = \frac{1}{2}$ to $x_2 = \frac{m}{n} + \frac{1}{2}$.

$$\begin{aligned} E &= a\frac{x^3}{3} + b\frac{x^2}{2} + c \cdot x \Big|_{x=x_1}^{x_2} \\ a &= \frac{a_1 - 2a_2 + a_3}{2} \\ b &= \frac{a_1 - 4a_2 + 3a_3}{2} \\ c &= a_3 \end{aligned}$$

The same calculation of L for the linear prediction model can be used without any modifications:

$$L = t_b + t_{b+1} - 2a'.$$

9.4.4 Determining if a Load Balancing Operation is Beneficial

It is easy to predict whether a load balancing operation was beneficial once the expected execution time E and the actual execution time A have been calculated. First obtain A, as it is the actual measured execution time of m steps following the load balancing operation prior to the subsequent load balancing operation. The expected benefit due to load balancing is thus $B = E - A$. Once L and B have been calculated, it can be determined whether or not a load balancing operation was beneficial. If $L > B$, the load balancing cost is higher than the expected time saved due to the improved load balance. In this case the load balancing operation was likely detrimental to the overall program's performance because either the load balancing operation did not help improve performance enough or the load balancing operation was performed too soon after the previous one. Conversely, if $L < B$, then the benefit due to load balancing outweighs the cost of the load balancing operation and the performance of the overall program was likely

improved by the load balancing operation.

In the case where there is dynamic behavior (such as a continual increase in the amount of work across the whole system) in the program, this determination of the utility of each load balancing operation is still accurate because only local information is used. At each load balancing operation, only a local window of application steps around the operation are examined to determine the operation's utility. If a program's behavior is slowly varying, then the errors in this utility calculation will be small. If however, the program's behavior is wildly erratic, then neither this scheme nor any other will be of much use.

9.4.5 Automatically Adjusting Load Balancing Period Based On Utility

Anytime a recent load balancing operation is deemed to be beneficial, the load balancing period ought to be increased, as more frequent load balancing may further be beneficial. If the load balancing operation was deemed to be detrimental, then load balancing operations ought to be performed less frequently. In the control point framework, the load balancing period control points are either multiplied by $\frac{1}{2}$ or 2 in either of these two scenarios.

9.5 Analytical Model for Optimal Load Balancing Period

This section derives an analytical model for the optimal load balancing period for programs exhibiting a certain type of load imbalance pattern. Although just one analytical model is contained herein, in the future other analytical models could be developed to accurately reflect other more complicated models of dynamic load imbalance.

Assume that the execution time for a step that occurs in the perfectly load balanced state is the constant t_{min}. Assume further that the execution times for steps degrade at a rate of $m\frac{\text{sec}}{\text{step}^2}$. Assume the cost of each load balancing operation is a constant value c sec. Assume that each step immediately after each load balancing step has an execution time of t_{min}. It can be proven that the optimal load balancing scheme performs load balancing operations

at fixed intervals of steps. Specifically the optimal choice for load balancing period is $\sqrt{\frac{2c}{m}}$. The proof of this claim is found in Appendix A.

9.5.1 Practical Concerns for the Analytical Model

There are some practical concerns with just choosing the optimal value $\sqrt{\frac{2c}{m}}$ for the load balancing period. Figure 9.4 shows the per-step execution times for a run of Fractography3D. The first two load balancing steps are performed at steps 1500 and 3000 respectively. It is clear in the figure that the second load balancing step does not return the subsequent step times to the minimal step time of about 0.3s. Instead the load balancing operation only results in a modest decrease in execution time. This means that the load balancing operations in the Fractography3D application do not result in a behavior that satisfies the assumptions upon which the $\sqrt{\frac{2c}{m}}$ analytical model is based. Thus it is not expected that this simple analytical model will be very useful for all applications. In this dissertation, no other analytical models have been developed for more complicated types of applications, although it is likely possible that other models could be useful.

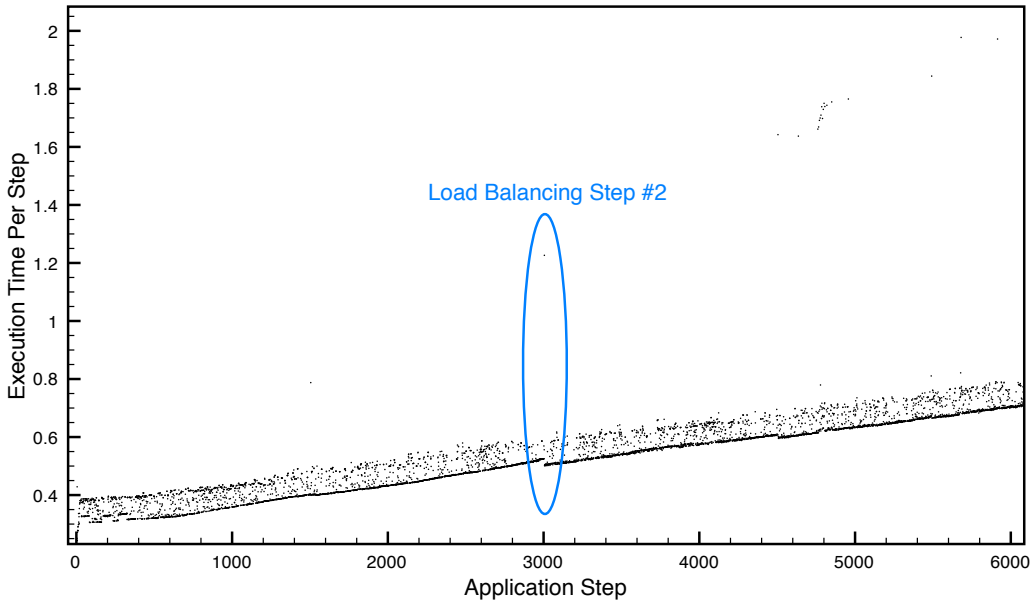


Figure 9.4: The load balancing operations in the Fractography3D application do not result in a behavior that satisfies the assumptions upon which the $\sqrt{\frac{2c}{m}}$ analytical model is based.

One specific problem with a naive direct application of the $\sqrt{\frac{2c}{m}}$ analytical model is that sometimes the measured values of the degradation rate is negative (i.e. $m < 0$), which results in a non-real value for $\sqrt{\frac{2c}{m}}$. This occurs frequently in Fractography3D when short load balancing periods are evaluated. Thus the mechanism implemented in the tuning framework will only decrease the load balancing period if $m > 0$, and will double the load balancing period if $m < 0$. This ensures that the load balancing period does not get stuck at the minimal value as commonly occurs otherwise.

9.6 Results

The Fractography3D program was run on 200 processor cores of an IBM Power cluster named Blue Print at the NCSA. The program's 3-D simulation domain was decomposed into 1000 AMPI VPs. Figure 9.5 shows the resulting performance of the Fractography3D program over a range of values chosen for the load balancing period. The load balancing period control point is either static, or it is adjusted automatically using each of the three utility approaches described in section 9.4 or the analytical model found in section 9.5. The results show that if a poor initial choice of load balancing period, such as 5 or 10 is made, then the static cases perform poorly, up to 71% worse than no load balancing at all. But if the control point values are automatically steered, even starting with these poor initial configurations of 5 and 10, the resulting program performs 48% better than the program without load balancing.

Selected resulting trajectories for the control point values as the program runs are displayed in figures 9.6, 9.7, 9.8, and 9.9. These trajectories reveal that the constant predictor and quadratic predictor heuristics generally favor longer load balancing periods (less frequent load balancing) than the linear predictor heuristic. The analytical model results in the most frequent load balancing, converging to about 200 steps. The quadratic predictor scheme is always faster than the static case when starting with any of the initial load balancing periods up to 2000 steps. When using an initial value of 4000 steps, the three initial warm-up phases represent 60% of the total execution time, and hence the utility of tuning is minimal. Longer runs for hundreds of thousands of steps, such as those used by the original engineers writing

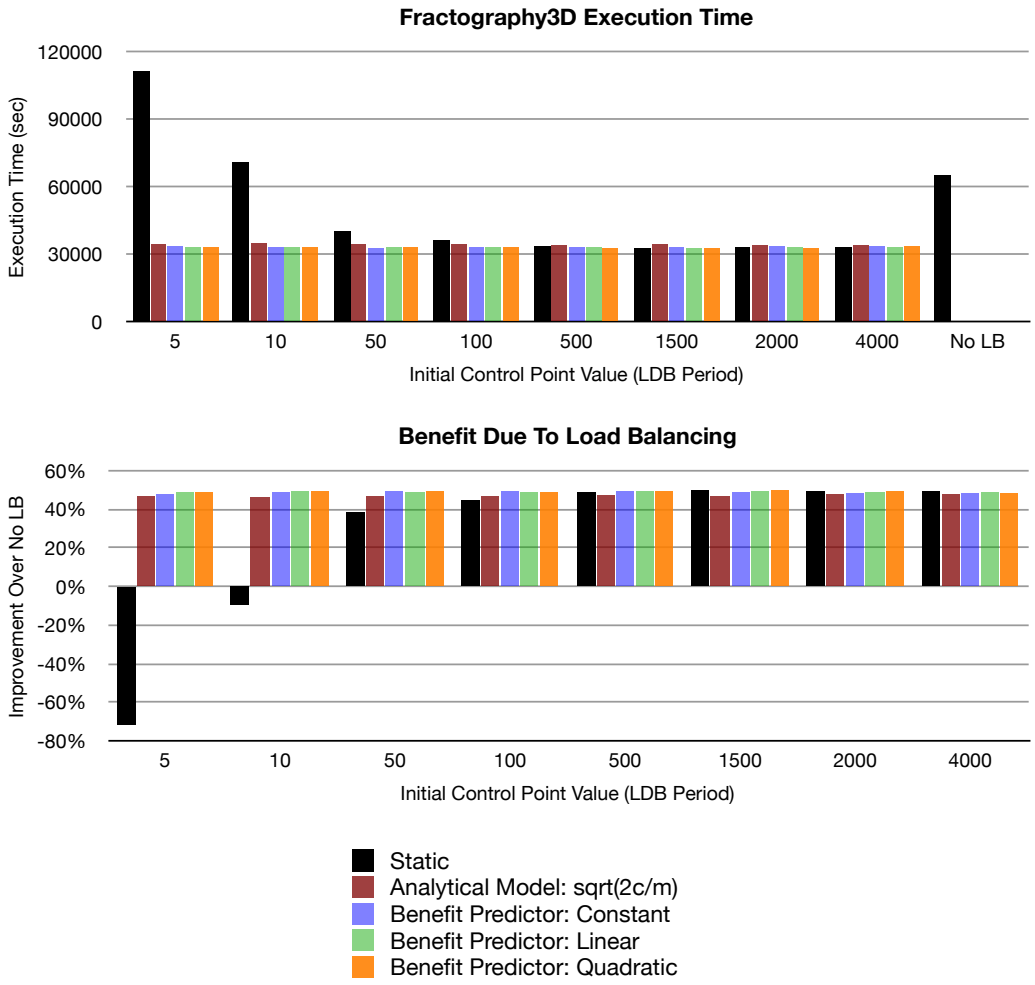


Figure 9.5: Performance of Fractography3D program for various choices of initial control point value (load balancing period).

the program, would likely show the utility of tuning even with an initial load balancing period of 4000 steps.

It can also be inferred from the decisions made by all the tuning schemes that it is beneficial to perform frequent load balancing early in the program and infrequent load balancing later on. This behavior is consistent with previously reported analyses of the application [43]. Additionally, it was shown in figure 9.5 that the best performing tuning scheme was generally the utility based approach that uses the quadratic predictor. All the other automatic tuning schemes performed similarly or slightly worse than the quadratic one.

All 4 of the tuning schemes can be enabled in the control point tuning framework using the following command line arguments: `+CPLDBPeriod`, `+CPLDBPeriodLinear`, and `+CPLDBPeriodQuadratic` for the utility models and `+CPLDBPeriodOptimal` for the analytical model described in this chapter.

9.7 Programmer Burden

The programmer burden of adding a load balancing period control point is small. Only 18 lines of code are added to the program. These small modifications to the program are displayed in figure 9.2. The most burdensome modifications to the program are those for which the application developer would need to understand the semantics of the control point framework. Specifically, the first MPI rank explicitly disables its ability to migrate during load balancing steps, because the first rank makes the calls to `controlPoint()` . All control point calls are currently required to be made on processor zero. Also, the framework is instructed not to automatically advance from one phase to the next, so then the program can call `gotoNextPhase()` when it is convenient. The tuning framework expects the load balancing to be performed at the beginning of a phase, so the calls must then be placed in the correct parts of the application timestep loop, respecting certain ordering requirements. Other than these semantic requirements, the adding of such calls to a program ought not take more than an hour or two, or at most one day. Similar amounts of time might be spent to manually tune the load balancing period for the program on a new platform. Thus the programmer burden is low for this type of control point.

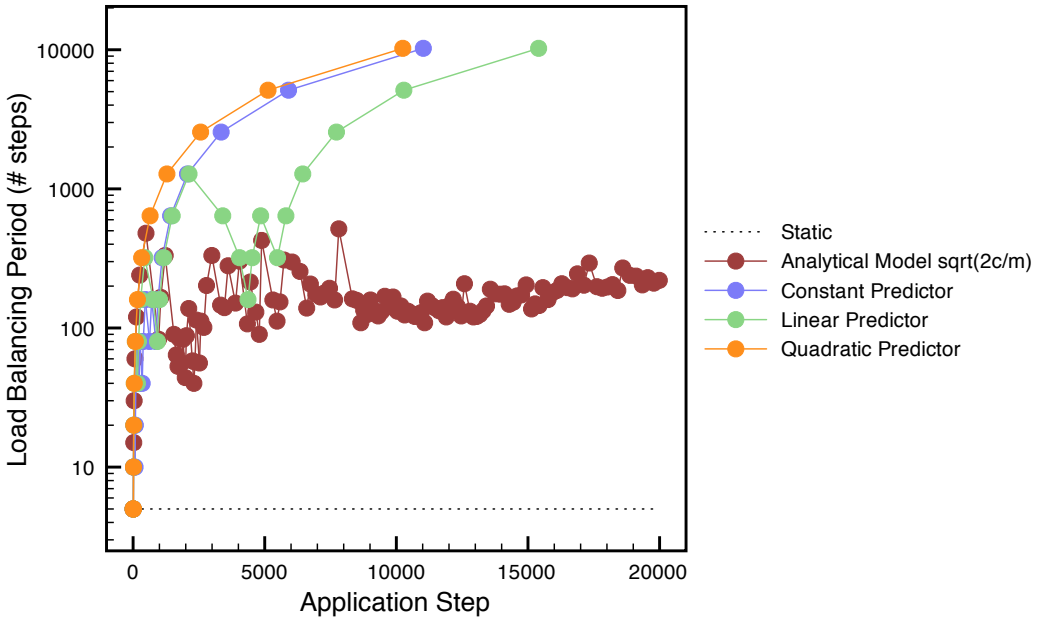


Figure 9.6: Trajectory of a control point's values for two steering heuristics from initial value of 5 steps between successive load balancing operations.

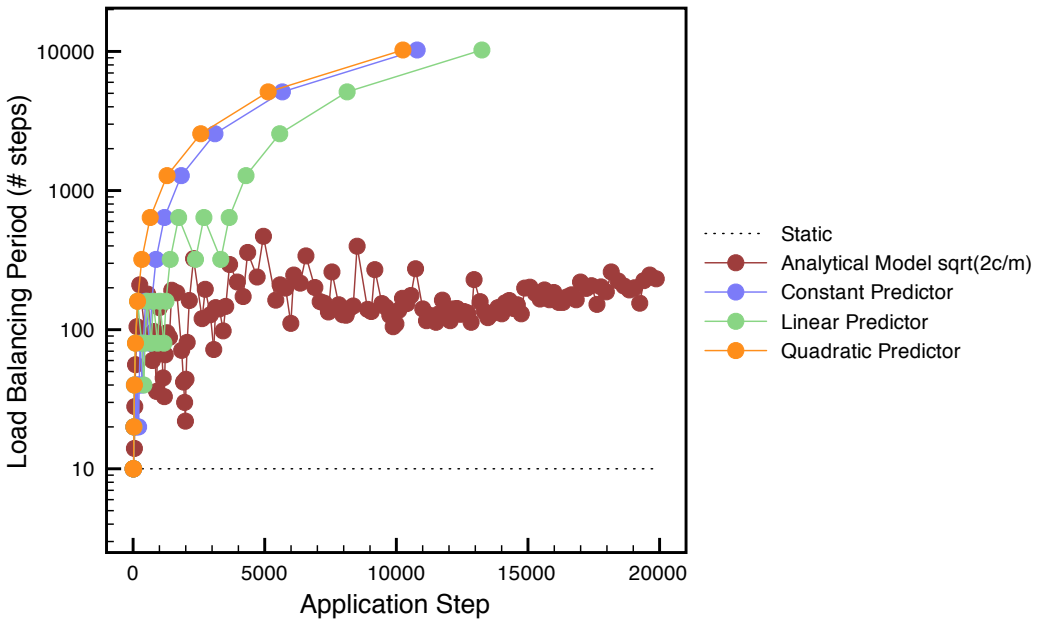


Figure 9.7: Trajectory of a control point's values for two steering heuristics from initial value of 10 steps between successive load balancing operations.

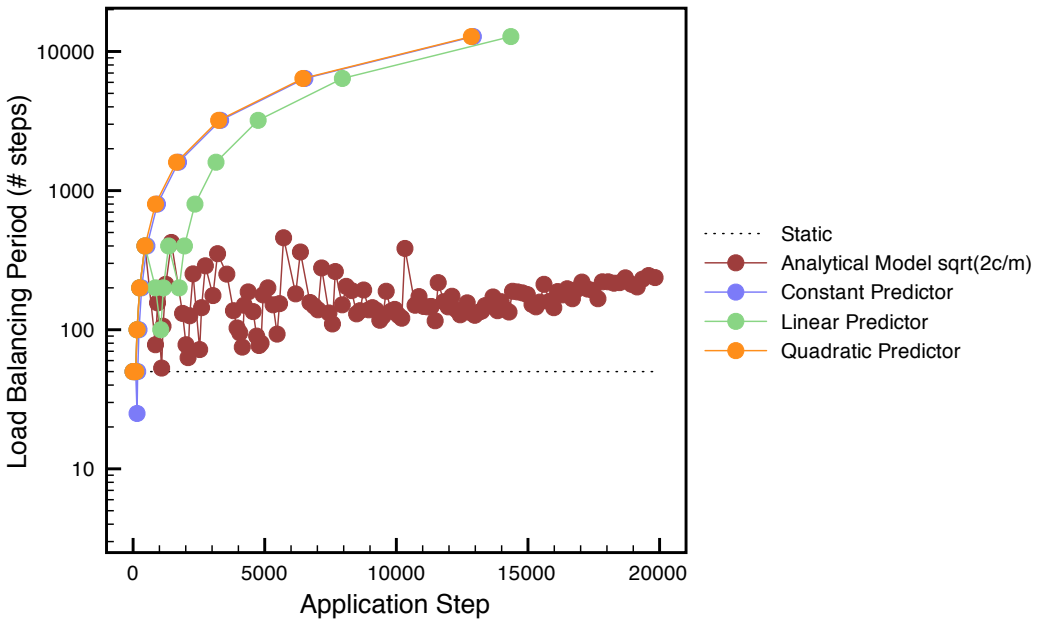


Figure 9.8: Trajectory of a control point's values for two steering heuristics from initial value of 50 steps between successive load balancing operations.

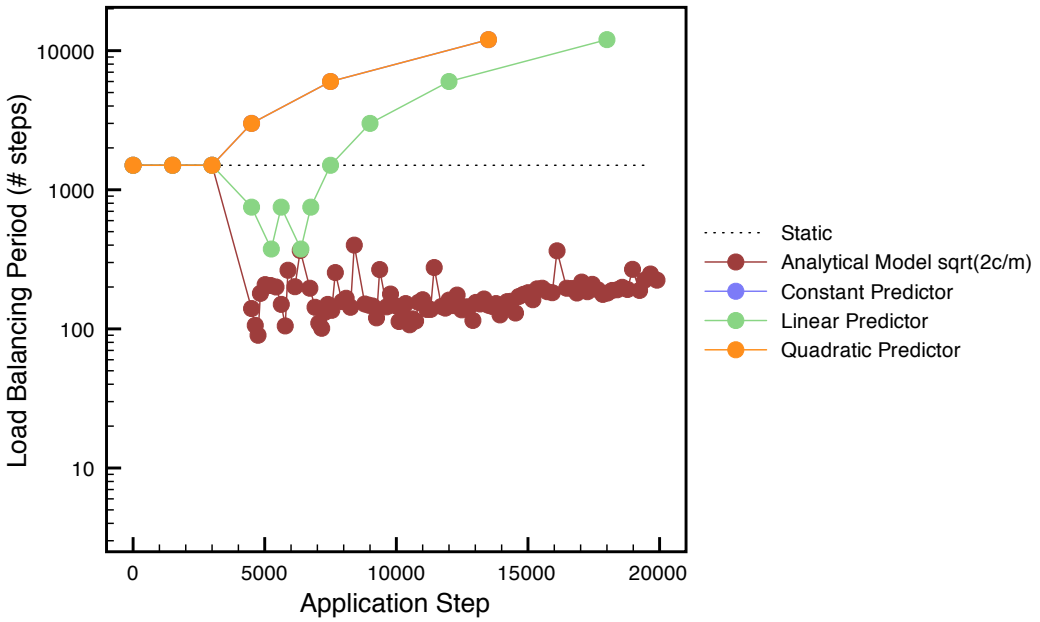


Figure 9.9: Trajectory of a control point's values for two steering heuristics from initial value of 1500 steps between successive load balancing operations. The first three phases are visibly flat because the control point framework treats the first three phases for all applications as warm-up phases.

The burden of adding the control point, however, is only incurred once, whereas the manual tuning of load balancing periods for each new input dataset or for each new parallel platform is incurred repeatedly. Thus an automatic scheme, possibly the one provided in this chapter, certainly ought to be used.

9.8 Summary

This chapter has shown that a control point that adjusts the frequency of load balancing can be automatically adjusted, either using an analytical model, or by using a utility evaluation approach. Even when a poor initial load balancing period is chosen, the four proposed algorithms quickly correct the poor choices resulting in good performance for the Fractography3D program that exhibits dynamic load imbalances.

CHAPTER 10

Control Points in LU Factorization

10.1 Application Overview

In parallel implementations of dense matrix factorizations, there are many parameters that drastically impact the performance of the application. Some important parameters include the matrix decomposition block size, the scheme for mapping the blocks onto processors, and parameters that balance between memory requirements and forward progress. Portions of this chapter have been published by Dooley et. al. [1].

10.2 Adding Control Points

A parallel implementation of dense LU factorization [1] written in Charm++ has been modified to include three control points. One control point determines the block size. The second chooses between alternative schemes that map blocks onto the processors. The third adjusts a threshold that restricts the progress of the algorithm in order to restrict the required amount of available memory.

10.3 Adapting Block Sizes

The first control point determines the size of the square blocks into which the input matrix is partitioned. Varying the size of the blocks affects performance. Larger block sizes provide higher performance for the sequential matrix-matrix multiply kernel that composes a large fraction of the total execution time for the LU factorization. Larger block sizes however reduce the available parallelism both at the beginning and at the end of the factorization. Figure 10.1 shows the effect of the reduced available parallelism where processors are increasingly idle for larger block sizes. The tuning scheme that adjusts grain sizes from section 7.4 was applied to this LU program's block size control point. The figure corresponds to a run of the program with a $N = 10240$ sized matrix on 64 cores, one core per node, of the Surveyor BG/P system at Argonne National Laboratory.

Switching block sizes between successive matrix factorizations might require an additional permutation of the data across the processors, however the matrix is of size $\Theta(n^2)$ while the computation time is $\Theta(n^3)$, so for large matrices, the cost of switching from one block size to another is negligible. Furthermore, the program might already be permuting the data when assembling the matrix.

10.4 Selecting Block to Processor Mappings

When the LU program was written, two different mapping schemes were investigated. The classic block-cyclic mapping scheme is a scheme that has low network communication costs. There are two mapping schemes implemented in the LU program. The mapping schemes define the processor that creates and perform operations on each chare array element and its corresponding matrix block. The first is a traditional block-cyclic mapping that exhibits low network communication costs. A new more balanced mapping called *balanced snake mapping* has higher communication costs but a better balance of work across all processors especially at the beginning and end of each matrix factorization [47], and achieves better performance for certain problem sizes and numbers of processors. In both cases, the mapping schemes are static, so the blocks do not migrate between processors within a

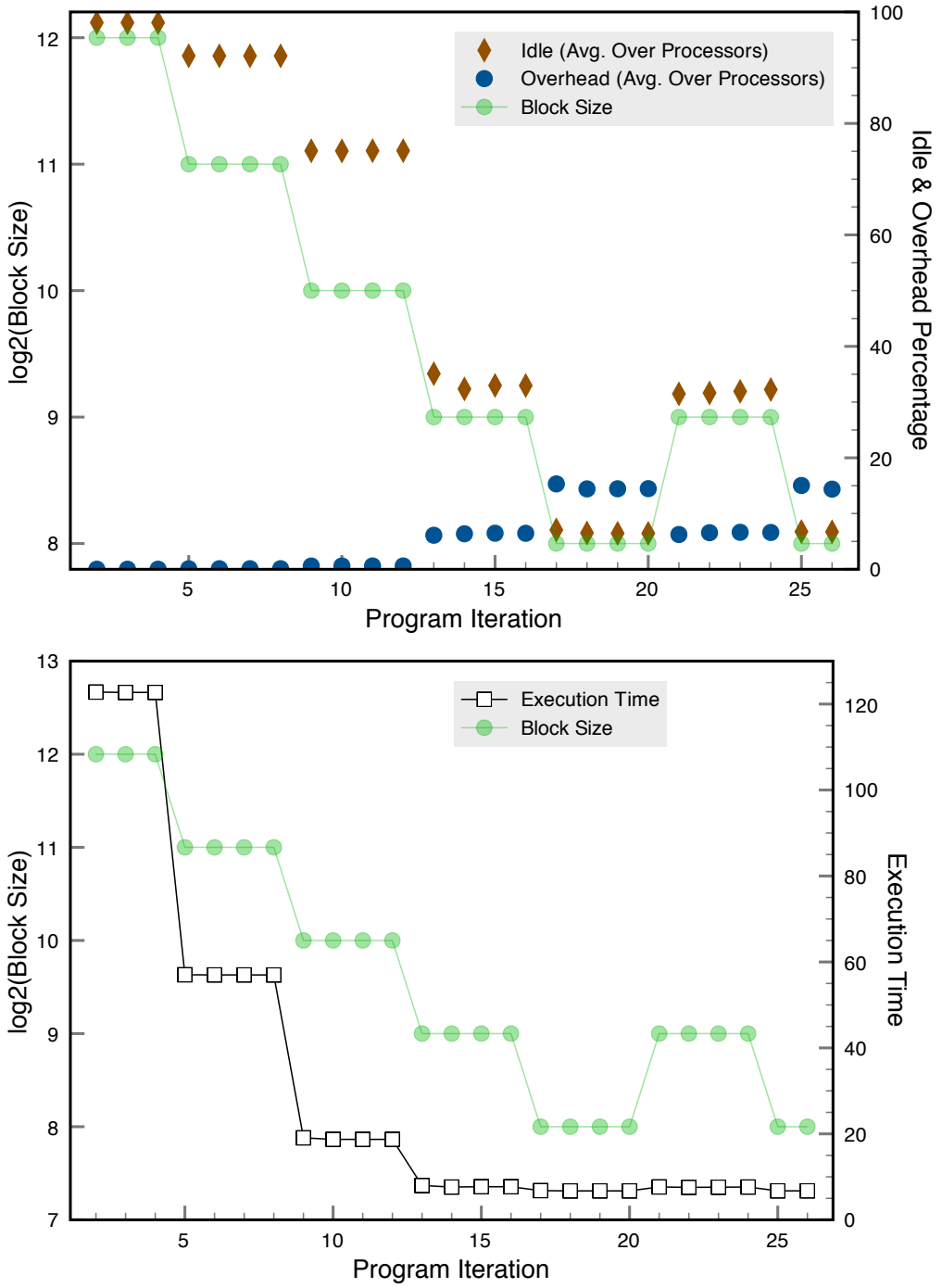


Figure 10.1: 20 successive matrix factorizations are performed as the control point determining the decomposition block size is decreased every four iterations when idle time is high. Ultimately a steady state alternates between block sizes of 2^8 or 2^9 which both achieve nearly identical execution times.

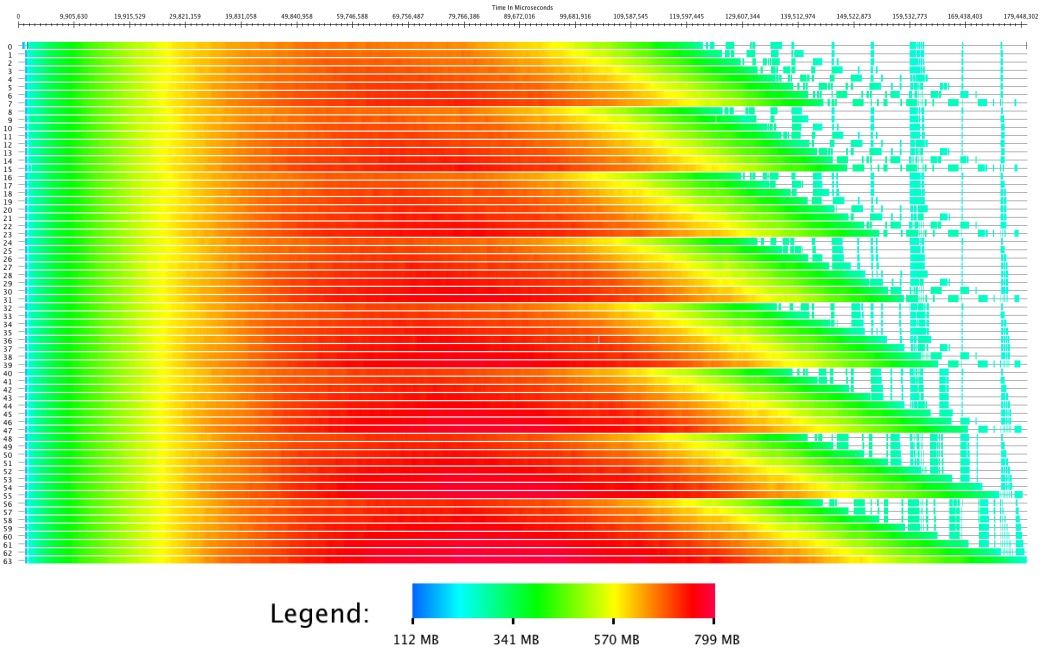


Figure 10.2: A timeline view, colored by memory usage, of an LU program run on 64 processors using a traditional Block-Cyclic Mapping for a $N = 32768$ sized matrix with 512×512 sized blocks. The traditional block-cyclic mapping suffers from limited concurrency at the end (the right portion of this plot).

single LU factorization. All work associated with a block will be performed on the processor owning the block.

Block-Cyclic Mapping

The block-cyclic mapping scheme is the traditional method used by many parallel LU implementations [15]. The advantages of a block-cyclic mapping are its simplicity and its relatively low communication volume. Each row or column of blocks spans only $\sqrt{p}$ of the p processors. Thus all of the multicasts have at most $\sqrt{p}$ destination processors. However, the disadvantage is that the work is unevenly balanced near the end of the computation. Figure 10.2 visualizes the entire computation for a run of the LU program on 64 processors. In an attempt to fix the imbalance near the end of the computation, a second mapping scheme was developed.

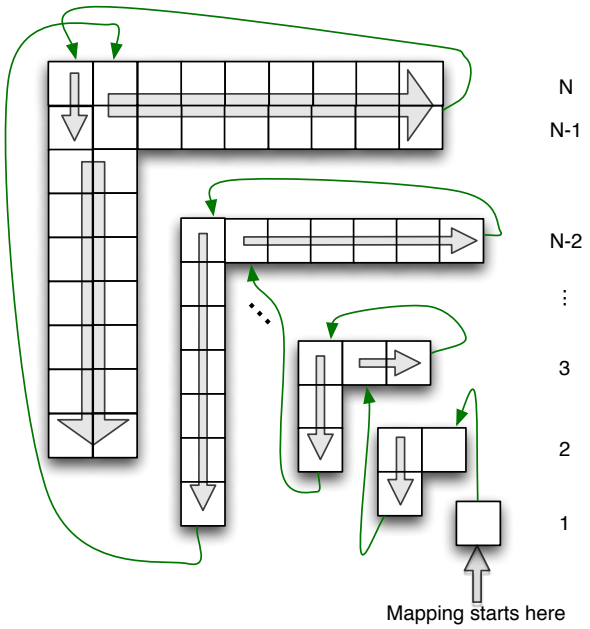


Figure 10.3: The traversal order for the *balanced snake mapping*.

Balanced Snake Mapping

In order to balance the amount of work that is performed on each processor, a new mapping scheme was developed called a *balanced snake mapping*. Figure 10.3 helps illustrate the order in which blocks are mapped in this scheme[1]. The blocks are traversed in the order shown by the arrows. This traversal order visits the blocks in roughly decreasing order of the amount of work expected to be performed by each block. As each block is visited, it is assigned to the processor which has been assigned the smallest amount of work so far. Thus the first p heaviest blocks will be assigned in a round robin manner to the processors, and the remaining blocks will be assigned in a manner that attempts to balance the load across the processors. The assignment function also forces subsequent blocks in traversal order to be on different processors.

It is expected that the number of processors spanning each row of blocks is larger than $\sqrt{p}$. In the case of 64 processors, with a matrix partitioned into 64×64 blocks, there are on average 43 unique processors spanning each column of the matrix and 49 unique processors spanning each row of the matrix. So in this case, the average number of unique processors on each row

[1]Figure 10.3 was created by Jonathan Lifflander.

and column is much higher than $\sqrt{p} = \sqrt{64} = 8$. Thus the multicast of a block along a row or column will involve more processors than the traditional block-cyclic scheme, and the multicasts will therefore incur a higher overhead. For large numbers of processors, the block-cyclic mapping performs better than this newly proposed balanced snake mapping.

Comparison of the Two Mapping Schemes

Although the balanced snake mapping does a much better job of evenly distributing the workload, the increased overhead for communication results in small delays between many of the matrix-matrix multiplications when compared to the block-cyclic mapping. Figure 10.2 shows that the block-cyclic mapping exhibits a load imbalance near the end of the computation, while the balanced snake mapping for the same problem exhibits a much better load balanced, as seen in figure 10.4. When the $N = 32768$ problem with 512×512 block sizes is run on 64 processor cores the balanced snake mapping performs better, achieving 138 GFlop/s, whereas the block-cyclic mapping yields 131 GFlop/s. A theoretical analysis of the computation and communication properties of the block-cyclic mapping and some other matrix decomposition schemes are provided elsewhere [48].

Automatically Determining The Optimal Mapping Scheme

Although it is clear that the block cyclic scheme has benefits for large numbers of processors, and the balanced snake mapping exhibits a better load balance for small matrix sizes, the decision of which scheme to use for a specific problem size and machine depends upon the performance characteristics of the machine as well as the problem size. Thus it is advantageous for the choice to be made automatically. This section describes one such method for choosing between the mapping schemes at runtime. It is possible to automate the choice between the two mapping schemes. The automatic decision can utilize the fact that the block-cyclic mapping scheme produces larger amounts of idle time for some of the processors toward the end of the factorization.

An automatic decision can be made between these two schemes by utilizing the fact that the block-cyclic mapping scheme produces larger amounts of

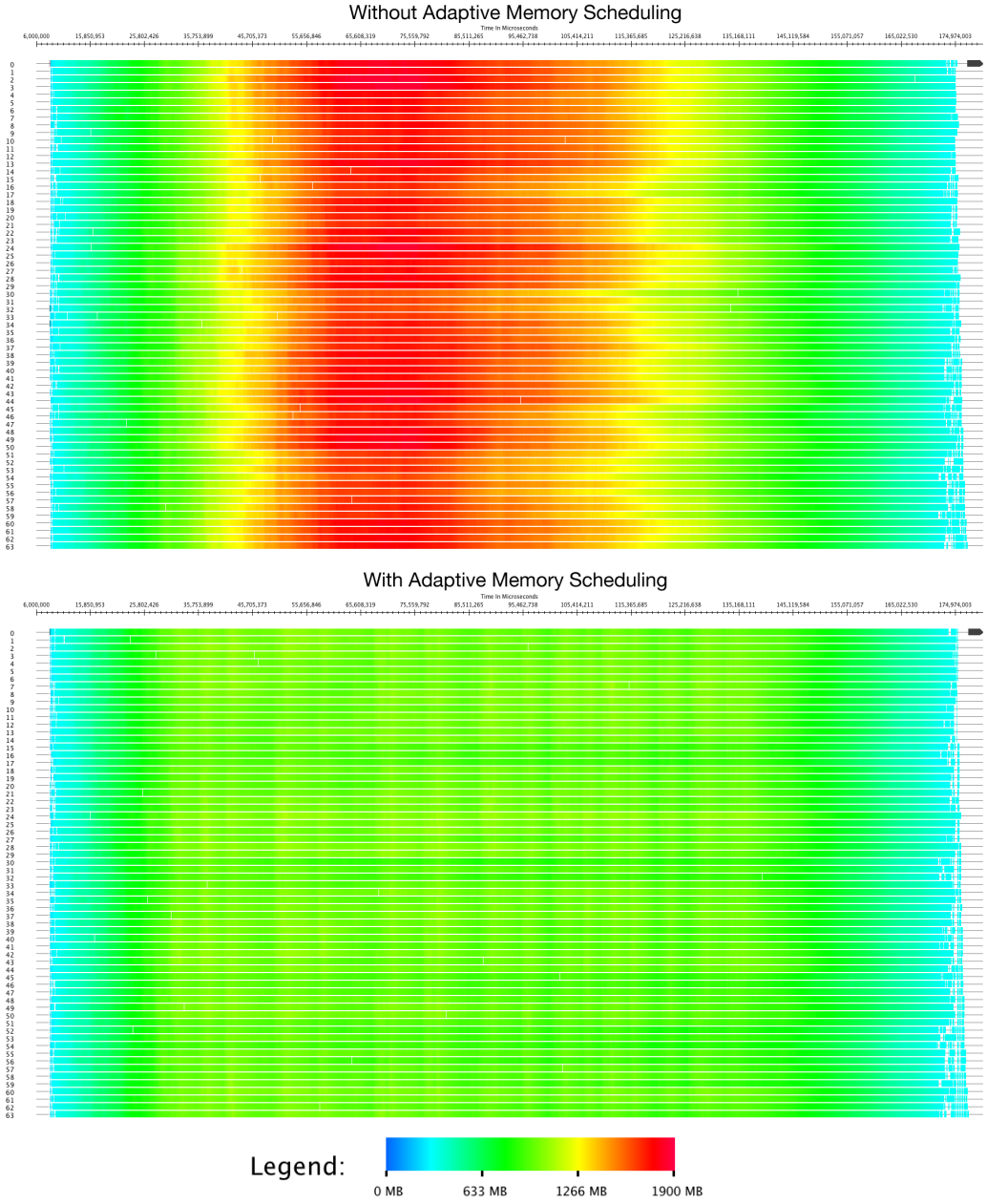


Figure 10.4: Plot of memory usage on each processor over time, both without and with adaptive scheduling using a 1000MB threshold.

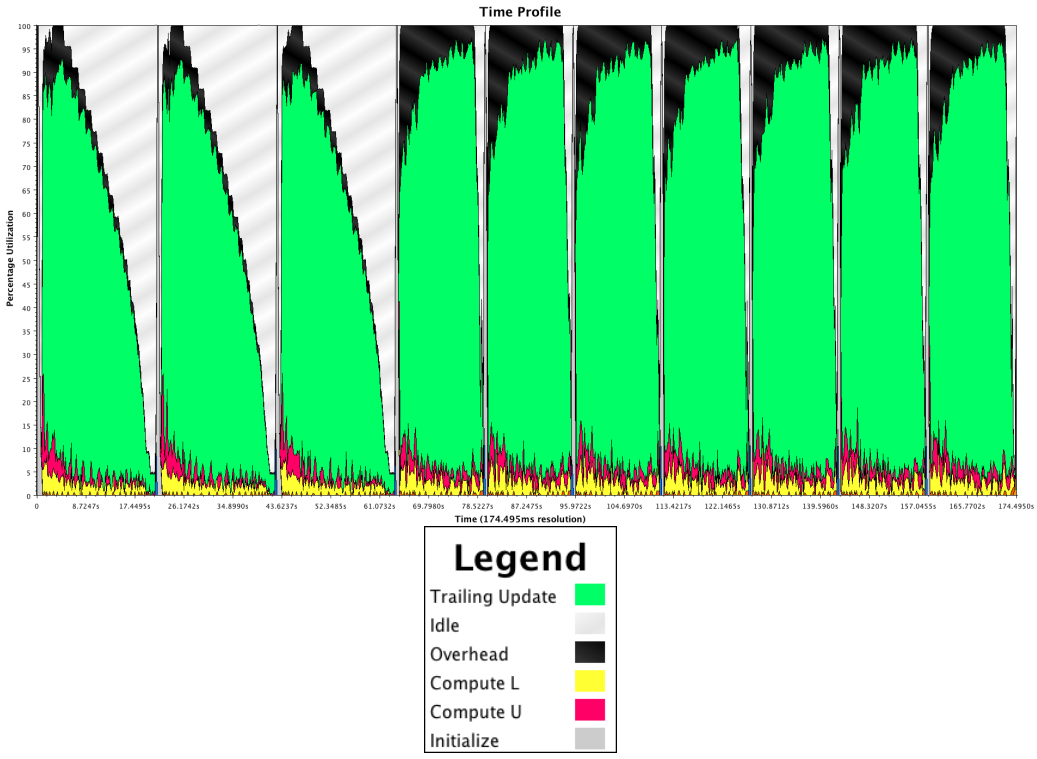


Figure 10.5: Visualization of a program performing 10 LU factorizations. After the third LU factorization, the measurement based automatic steering framework instructs the program to use the snake-mapping instead of the block-cyclic mapping. This adaptation reduces the amount of idle time found in the subsequent seven factorizations.

idle time for some of the processors toward the beginning and end of each factorization. To automatically determine which scheme to use, a control point has been added to the LU program. In this case, the program specifies that one mapping scheme will increase the available parallelism. The steering framework can therefore turn the control point knob when a large amount of idle time is detected. Figure 10.5 shows a performance visualization of an execution of the program performing 10 consecutive LU factorizations. The initial LU factorizations result in a large amount of idle time because the matrix blocks are not well distributed across the processors. Hence after a period of observation for the first 3 factorizations, the tuning framework notices the large amount of measured idle time and adjusts the control point value that switches from the block-cyclic mapping scheme to the balanced-snake scheme. The subsequent LU factorizations complete more quickly.

10.5 Adapting Algorithmic Parameters that Affect Memory Consumption

The third control point added to the LU program affects the memory usage of the algorithm. In all high-performance LU factorization programs, an effort must be made to maximize the parallelism while restricting the memory requirements of the resulting program. Some possible methods include processors communicating with each other to coordinate their data transfers, or a user specifying a fixed amount of permitted lookahead. In this LU implementation, a memory aware scheduling technique is used, and a control point has been added to adjust a threshold T used in the scheduler. As the value T increases, the scheduler encourages more parallelism to be exposed in the program, and when the threshold is decreased, less parallelism is exposed, so the program's memory usage is generally reduced. The threshold does not produce a hard memory bound.

We implemented an automatic tuning scheme that starts with a safer low value for the control point and increases it while observing memory consumption across all processors. The threshold is automatically increased when it is observed that there is available unused memory on all processors. When little free memory is available, the control point value is no longer increased. The program's memory usage does roughly increase with the control point values, as seen in figure 10.6.

Figure 10.7 shows the relationship between the control point values and the measured idle time and overhead time for the program. In this figure, there is a large amount of idle time, 25%, occurring when $T \leq 2$, but smaller amounts of idle time 10% when $T > 2$.

10.6 Programmer Burden

The existing LU program, prior to adding control points, was capable of performing an LU computation using different parameters such as multicast strategy, block size, and mapping scheme. Thus to adjust these values using control points from one factorization to the next was almost trivial. Figure 10.8 lists the code added to the program that exposes the three control points. In total only 15 lines are added.

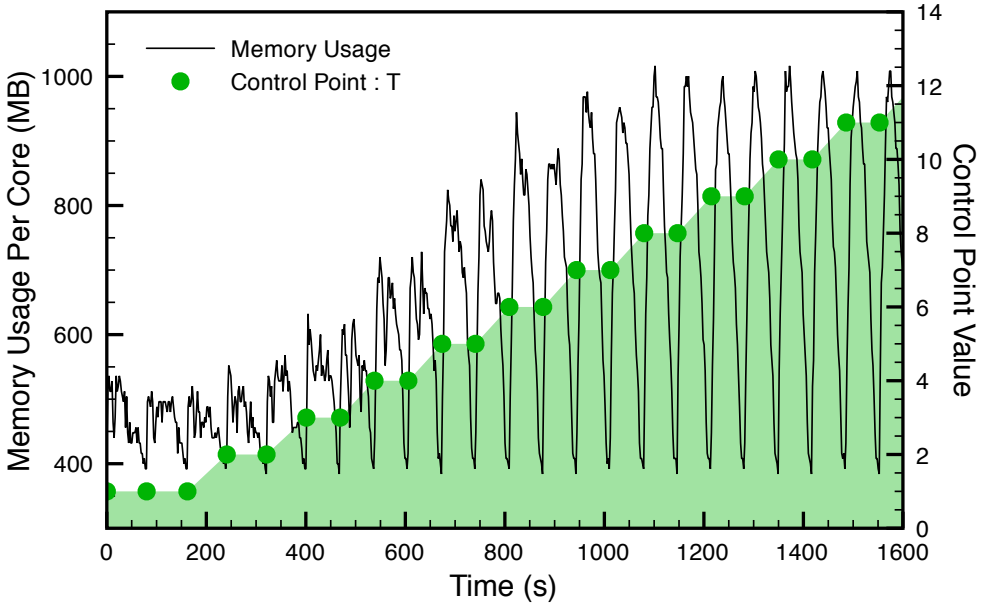


Figure 10.6: Actual memory usage increases for an LU program as it performs 23 successive factorizations while a control point value (scheduler threshold T) is increased.

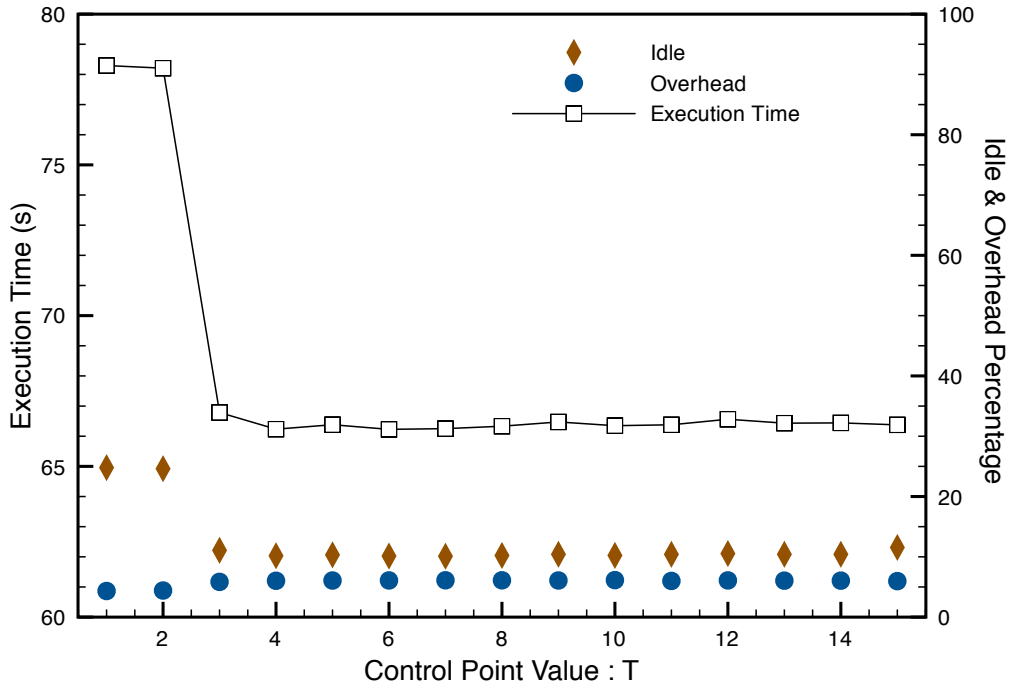


Figure 10.7: Measurements show that the idle time is higher(25%) for low values of scheduler threshold T.

Declarations:

```
#include <controlPoints.h>
```

Once at startup:

```
ControlPoint::EffectIncrease::GrainSize("block_size");
ControlPoint::EffectIncrease::Concurrency("mapping");
ControlPoint::EffectIncrease::NumMessages("mapping");
ControlPoint::EffectIncrease::MessageOverhead("mapping");
ControlPoint::EffectIncrease::MemoryConsumption("
   memory_threshold");
```

Between each iteration:

```
registerControlPointTiming(duration);
```

```
// Only advance phases every other iteration
if(iteration % 2 == 1 || iteration==1){
   gotoNextPhase();
   whichMulticastStrategy = controlPoint("multicast_strategy"
      , 1, 3);
   BLKSIZE = 1 << controlPoint("block_size", 5, 12);
   mapping = controlPoint("mapping", 0, 1);
   memThreshold = 100 + controlPoint("memory_threshold", 0,
      20) * 50;
}
```

Figure 10.8: Code added to LU program to expose three control points.

The specification of the effects of the control points is straightforward, but a mapping of the integer control point values to the values used by the application is required. The LU program only supports block sizes that evenly divide the matrix size, thus for matrices of size 2^n, only block sizes that are smaller powers of two are permissible. Thus whenever a control point value $5 \leq x \leq 12$ is retrieved, the value 2^x is stored into the application variable `BLKSIZE` . Just as the integer range needs to be mapped to a set of values useful to the program for the block size parameter, a set of values $\{0, 1\}$ are mapped into two different mapping schemes, but this mapping already existed in the LU program prior to the control point modifications. For the memory threshold, a small range of integer values $0 \leq x \leq 20$ are mapped onto a range of memory threshold values in 50MB increments. This range was chosen arbitrarily, but the scaling factor of 50 is necessary because the control points are only varied in increments of one, and increments of 1MB to the memory threshold would be too slow on modern machines with thousands of megabytes of memory.

10.7 Summary

This chapter has shown that control points can adjust multiple aspects of a parallel LU factorization program. The block size of the program can be adjusted in response to the measured idle times. A binary control points can choose between two alternative mapping algorithms. A final control point allows a program to expand its memory footprint whenever excess available memory is observed. These control points demonstrate that these three types of parameters can be adjusted in response to measurements of memory usage and idle time.

CHAPTER 11

Control Point for Communication Throttling

Sorting is an operation that appears in numerous applications. For example, in the decomposition of an n-body gravity program, a sorting operation is required to partition the space-filling curve that orders the n bodies. One main problem with sorting large numbers of items in parallel is the large permutation of items that must occur. In parallel, correct algorithmic choices must be made to keep this large permutation from overwhelming the interconnection network, while also maximizing the use of the network to minimize the total time spent in the sort operation.

11.1 Application Overview

A highly scalable parallel sorting algorithm has been implemented as a Charm++ library [49]. The sorting algorithm performs a histogramming operation to determine the final destination processor of all data values prior to a single permutation of the data from the original owner processors to the resulting destination processors. The parallel histogramming operation computes appropriate *splitter keys* that mark the processor boundaries onto which the original data items will be mapped. A special feature of this algorithm is that it performs partial local sorts as the histograms are being constructed. The full local sorting operation on each processor can therefore be overlapped in part with the parallel histogramming operation

that determines the appropriate splitter keys. This overlap improves the performance of the algorithm significantly. After the histogramming operation completes, the splitter keys are then used to permute the locally sorted data from its original processor to its final destination processor. Finally, each processor sorts its local data that it received from the permutation.

This sorting algorithm has been modified to expose multiple control points. These control points expose pre-existing algorithmic parameters that were originally adjusted manually or by a simple driver routine to find high performance configurations. Converting these parameters to control points was quite simple.

11.2 Adding Control Points

Although four tunable parameters have been exposed as control points, attempts to adjust two of these resulted in application crashes due to bugs in the library[1]. The two control points causing crashes are the hist_thresh and splice_thresh parameters. Two other control points, however, can successfully be varied causing the performance of the program to vary.

The first useful control point is called *bucketCP*. This parameter determines the degree of throttling for messages in the all-to-all data shuffle. Specifically, processor p will send its outgoing data to processors $(p+1)$ through $\left(p + \frac{chares}{2^{bucketCP}}\right)$ without waiting on any incoming data. Then as each incoming piece of data arrives, a corresponding piece is sent to a subsequent processor. If the value of the control point is very high, then few outstanding messages will be in flight, and the shuffle might not fully utilize the available interconnection network. If the value of the control point is low, then a large number of messages will be in flight, possibly causing contention in the network. The effect of the control point is specified by the program: lowering the control point value increases the available concurrency. The tuning scheme proposed in section 7.4 already is capable of handling control points with this specified effect.

[1]Converting application parameters to control points is only effective when those application parameters actually can be varied without impacting the correctness of the program. If varying these "parameters" causes the program to crash, then perhaps the parameters were not actually parameters but rather were some type of constant.

The second control point is an application parameter called *probe_max* which determines the number of splitter key guesses, also called *probes*, used for substeps in the histogramming operation. More splitter keys will allow the histogram to converge more quickly, but the amount of work required to build each local histogram increases.

11.3 Tuning Between Successive Sorting Operations

After the two control points were added to the parallel sorting program, many random configurations were evaluated. Varying the bucketCP control point produced the greatest changes in the performance of the sorting operations, while the probe_max control point had little effect on performance. Figures 11.1 and 11.2 show the resulting performance of the sorting operations as this control point is varied when running on 604 processor cores for two input data sizes of 2^{23} and 2^{34} 64-bit keys respectively. Figures 11.3 and 11.4 show results for larger problem sizes of 2^{34} and 2^{36} keys but on 1870 processor cores. The figures display the minimum measured idle time across all processors and the maximum overhead measured across all processors. All four of these runs were performed on Jaguar, a Cray XT5 system at ORNL.

The bucketCP control point was allowed to vary within a range of 3 to 9 inclusive. Values less than 3 exhibited specific performance anomalies known to occur on the Cray XT5 machines, so such small control point values were not further analyzed. This specific performance anomaly causes messages to be delayed for multiple seconds for no obvious reason. Figure 11.5 shows a timeline view of the end of the data shuffling phase of a sorting operation on 1870 processor cores (170 nodes). The message was sent seconds before it arrives on an otherwise idle processor core. Prior to the message finally being delivered, all processors for the job are idle. Not only in the parallel sorting program does this detrimental performance anomaly occur. The delayed messages have been observed in other Charm++ applications on Cray XT5 systems, and no solution is yet known to fix the problem.

For large values of the bucketCP control point, few messages are sent at the same time, resulting in a underutilization of the interconnection network. Many one-way latencies are incurred as the messages are injected slowly into

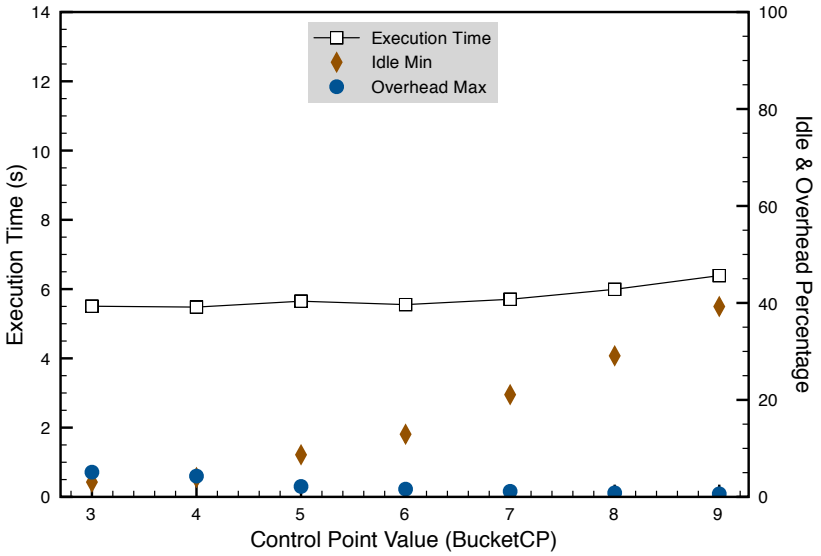


Figure 11.1: Performance of sorting algorithm ($N = 2^{32}$) on 605 processor cores (55 nodes) of Jaguar, over a range of values controlling the amount of data sent early in the algorithm (bucketCP).

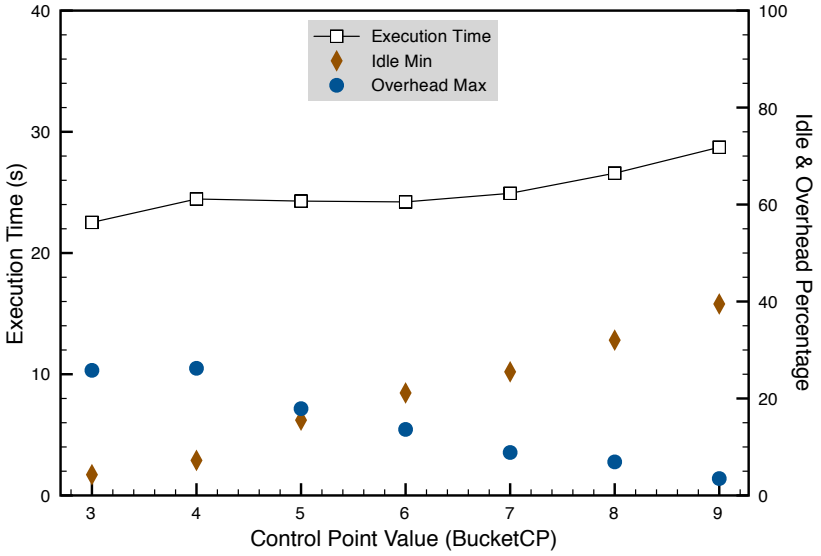


Figure 11.2: Performance of sorting algorithm ($N = 2^{34}$) on 605 processor cores (55 nodes) of Jaguar, over a range of values controlling the amount of data sent early in the algorithm.

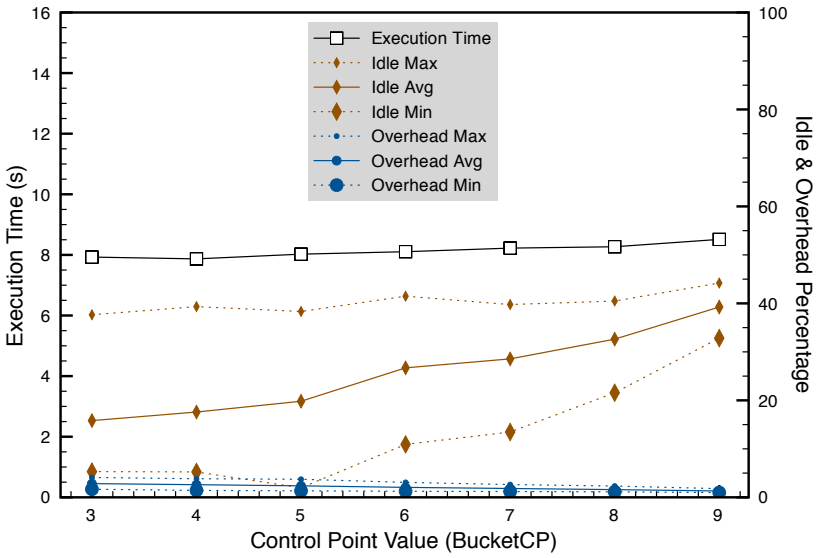


Figure 11.3: Performance of sorting algorithm ($N = 2^{34}$) on 1870 processor cores (170 nodes) of Jaguar, over a range of values controlling the amount of data sent early in the algorithm.

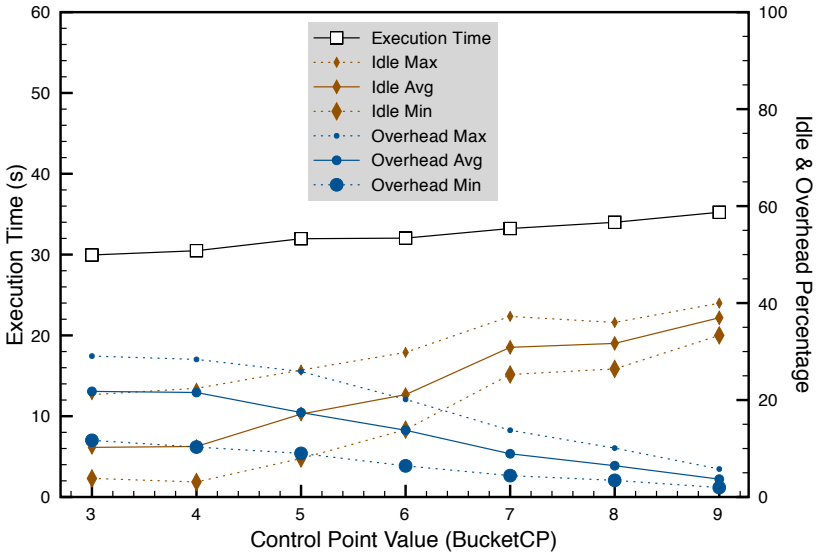


Figure 11.4: Performance of sorting algorithm ($N = 2^{36}$) on 1870 processor cores (170 nodes) of Jaguar, over a range of values controlling the amount of data sent early in the algorithm.

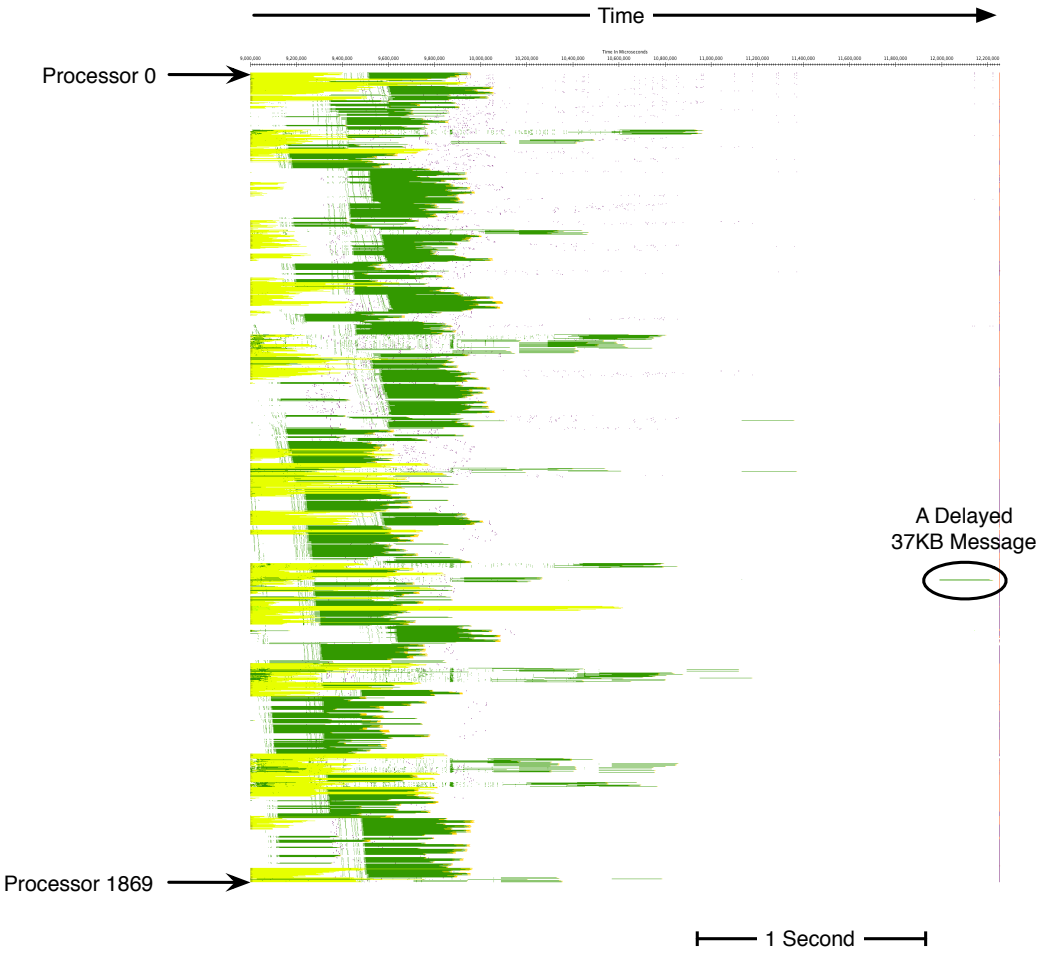


Figure 11.5: Horrible message delays occur for $bucketCP = 1$ and $N = 2^{36}$ on 1870 processor cores (170 nodes) of Jaguar, a Cray XT5 system at ORNL. This timeline shows that a message was processed after a prolonged period of no work on any processor. The cause for the multi-second network delays for relatively short messages on XT5 systems is unknown.

the network. As processors are waiting for messages, the processor cores are about 40% idle when this control point value is 9. For smaller values of the control point, more messages are sent at once, and the processors spend less time idle. For large problem sizes, as seen in figure 11.2 and 11.4, the larger volumes of communication cause processors to exhibit larger overheads for the smaller control point values.

In the four figures (11.1, 11.2 , 11.3, and 11.4), the best performing configurations are for low bucketCP control point values. For these best performing configurations, the minimum idle time across all processor cores is a few percent. For the poor performing configurations with larger control point values, the measured idle times are higher, up to about 40%. Thus it is clear that a tuning scheme that decreases the control point values whenever large idle times are measured ought to be able to successfully tune this control point effectively. In the future if the XT5 performance anomaly is fixed, the lower range could be expanded to gain a better understanding of how this control point behaves.

11.4 Programmer Burden

The parallel sorting algorithm was tested by a program that already was capable of varying parameters between multiple sorting operations. Thus it was nearly trivial to modify this program to expose the pre-existing parameters as control points. Just nine lines of code needed to be added.

11.5 Summary

This chapter analyzed the use of a two control points in a parallel sorting library. One of the two control points caused only minor changes in performance, but the second one influenced performance significantly. In response to high idle time measurements, this second control point ought to be reduced to increase the amount of available parallelism. Unfortunately, the range of low values for the parameter could not be fully examined due to peculiar observed performance bugs in the Cray XT5 system.

CHAPTER 12

Costs of Performance Tuning

There are three types of performance costs that need to be understood to accurately evaluate the utility of tuning a parallel application with control points. The first is the cost of measuring characteristics of the running program. The second is the gathering of the measurements from all processors. The third is the cost of generating a plan of which configuration ought to be tested next. This chapter examines each of these three costs.

12.1 Cost of Tracing

The first important cost that must be incurred by a control point tuning framework is the extra time spent recording measurements about the tasks executing within each processor. When recording performance measurements, each Charm++ entry method is timed, and the resulting duration is accumulated into a few counters. Additionally, other data is accumulated (summed) including the number of bytes in the message that caused the invocation. The specific points where tracing is performed are described in section 6.2.4. Modifying the counters and accumulating the time and message sizes adds a fixed constant amount of work to each entry method invocation. Because the entry methods may be short or long, the relative impact of the constant additional cost varies.

The time spent recording the measurements for m entry method invocations is mt_c where t_c is the cost added to each entry method invocation. The relative cost, that is the overhead incurred by the program,

is approximately $\frac{t_c}{t_e}$ where t_e is the average duration of an entry method invocation (i.e. average grain size). For large entry methods, the overhead becomes negligible: $\lim_{t_e \to \infty} \frac{t_c}{t_e} = 0$. For fine-grained programs with short entry methods along the critical paths, the costs of these measurements will become important.

Traditionally, Charm++ applications have been designed to exhibit neither fine, nor coarse grain sizes, but rather to exhibit medium grain sizes [50, 51]. A typical configuration of NAMD might, for example, have entry method invocations with durations from about $0.5ms$ to $1ms$ [52].

To analyze the costs incurred by the tracing module instances on each processor, a synthetic benchmark program has been created. The benchmark program exhibits, in a controlled manner, a wide range of different computational grain sizes. The range of grain sizes investigated with this benchmark includes those of typical Charm++ applications. The benchmark program can then be run with or without the tracing module enabled to measure the overhead cost incurred by the tracing module.

The benchmark sends a small message around a ring of processors multiple times. The resulting entry methods on each processor perform a specified amount of synthetic work. The amount of work is varied, resulting in varying grain sizes. This program was then run in four configurations on 110 processor cores of 10 nodes of Jaguar, a Cray XT5 parallel machine. The first configuration simply runs the benchmark with no tracing modules enabled. In the second configuration, a tracing module was compiled in to the program, but all of its function bodies are empty. The functions in this no-op tracing module are invoked for each entry method invocation, but no timing calls or measurements are made. This second configuration tests just the overheads of the Charm++ tracing interface.

The third and fourth configurations actually make measurements of the entry methods in a program. Both configurations measure grain sizes, overheads, idle times, and message sizes. One of these configurations also measures the memory footprint on the processor at each entry method invocation. The resulting timings of the second, third, and fourth configurations were compared with the baseline first configuration to infer the costs of the different types of measurements. Figure 12.1 displays these resulting costs across a range of application grain sizes.

In the figure it can be seen that the costs of performing these measurements

are very low, except for programs composed of incredibly short entry methods. The costs are less than 1% for all configurations with entry methods of durations at least $43\mu s$. The cost when entry methods are $10\mu s$ is about 4% when all measurements are made. Furthermore, there is a noticeable cost incurred when using a tracing module that doesn't even perform any work. Thus a significant portion of the costs are inherent to the current Charm++ tracing interface, which includes an iteration through lists of enabled tracing models then multiple virtual method calls for every trace point, which occurs at least twice per entry method invocation. Further optimizations or refactoring of the tracing interface in Charm++ could reduce this cost.

When examining the results, the average cost per entry method invocation can be calculated. In the configuration performing all measurements, the cost for each entry method invocation is $0.60\mu s$. When the memory footprint is not measured, the cost drops slightly to $0.53\mu s$. The expected costs can be inferred from these two values. The expected cost model, as shown in figure 12.2, matches the observed measurements of figure 12.1.

For applications with fine or ultra-fine grain sizes, it may be necessary to further reduce the time used in the tracing module, if control points will be of use. The types of applications exhibiting such fine granularities include the many modern applications that will be strong-scaled to the next generation of supercomputers. For example, scaling a given molecular system in NAMD to large numbers of processor cores will necessarily result in a finer grain decomposition and hence shorter entry method invocations.

The benchmark program has not yet been used to measure the costs of performing other more complicated types of measurements. However, the costs of performing one such complicated measurement, that of recording critical paths, is discussed in chapter 4.

12.2 Cost of Gathering Measurements From All Processors

After entry methods have been instrumented by a tracing module, the measured results need to be combined from all processors so that a decision can be made on the next set of control point values to use. The control point framework broadcasts a request to all processors, and a reduction is

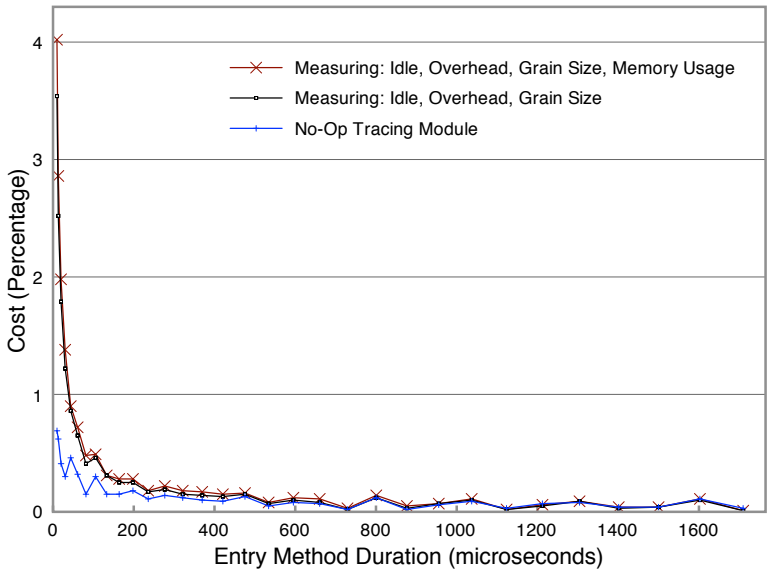


Figure 12.1: Costs of gathering measurements for a synthetic benchmark performing a 1-D ring communication pattern with varying amounts of synthetic work each hop.

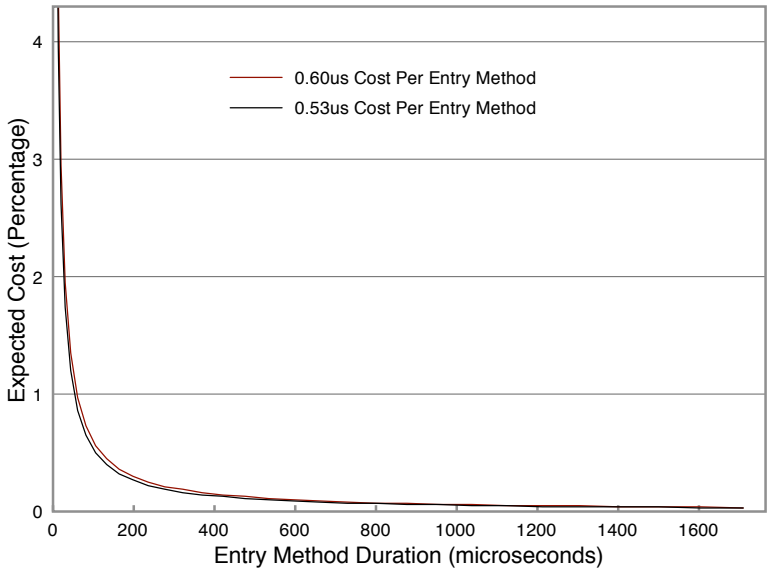


Figure 12.2: Model for the expected costs incurred when introducing a fixed overhead for each entry method invocation. This model matches well the observed costs shown in figure 12.1.

performed to merge the data across all processors as described in section 6.2.4 and displayed in figure 6.3. A program was created for measuring the time taken to perform this broadcast, reset the local counters, and perform the reduction of data to one processor. The program performs this operation 10000 times, and was run on a range of processors of Jaguar. Table 12.1 reports the time required to request and gather the data. Even on a large number of processors, it takes less than $2ms$ to perform one operation. In a real-world application, this gathering of data would be interleaved with the execution of the program itself, possibly perturbing the application's behavior. These low costs imply that it ought to be possible, even on the largest scale machines, to gather measurements every few seconds, with negligible costs.

Number of Processor Cores	Time to Request and Combine Measurements from All Processors
110	0.64 ms
10010	1.30 ms
50006	1.65 ms
100100	1.98 ms

Table 12.1: Time to request and combine trace data from all processors for large numbers of processors, including the time to perform a broadcast then extract measurements and perform a reduction.

12.3 Costs of Determining Next Control Point Values

The final cost investigated in this chapter is the cost of determining the next set of control point values based on the gathered data from a small number of recent phases. In this dissertation, a performance steering approach is taken, whereby only a small number of recent phases must be examined prior to making the decision on what values to use in the next phase. Thus the cost of scanning through a bounded number of recent phases is $O(1)$.

To analyze the typical expected times for plan generation, the nelder-mead simplex algorithm is run for a benchmark application. This algorithm was chosen because it examines data for at most n recorded phases, when there are n control points. Thus, we can vary the number of control points to measure the cost of a plan generation algorithm that examines different

Search Space Dimension	Time To Generate Plan
10	0.77 ms
3	0.31 ms
2	0.24 ms

Table 12.2: Cost of generating a plan (new control point values configuration) for a phase using the Nelder-Mead Simplex Algorithm.

amounts of the historic measurements. Although various tuning schemes will have different computational characteristics, the simplex algorithm ought to provide a useful prediction of at least the order of magnitude of the costs.

Figure 12.2 shows that it takes on average $0.77ms$ to determine the new set of control point values for each phase using the Nelder-Mead Simplex algorithm with 10 control points, where up to 10 previous phases are examined. When two or three control points are used, the time spent determining the next configuration is less than $0.5ms$ [1].

12.4 Summary

This chapter showed that the overhead of instrumenting an application, the expected costs of gathering performance measurements, and the costs of planning new configurations is low. Microbenchmarks showed that the costs of recording and gathering measurements does indeed depend upon the grain size of the application, and for typical grain sizes, these costs are just small fractions of 1%. The costs of planning new configurations for each phase are likely to be under one microsecond.

[1] These times correspond to runs of programs on a single Apple OSX 10.6.2 Mac Pro with two 2.8 GHz Quad-Core Intel Xeon processors.

CHAPTER 13

Related Work

13.1 Single Node SMP Autotuning

A common modern approach to obtaining good performance on single-node multicore systems is to use an autotuning framework. Such frameworks generate multiple implementations of a program, library, or function and execute the variants on the target platform to find the one that performs best [53, 54]. Such frameworks include PERI [55], POET [56], SPIRAL [57], FFTW [58], ATLAS [59], and PHiPAC [60]. The autotuning frameworks which are implemented in compilers or runtime systems often achieve performance comparable to hand-tuned programs [53]. All these projects try to prune the search space to reduce the set of configurations that need to be evaluated. It is also well known that the performance space is often non-linear, and thus it is difficult to effectively prune the exponentially large search space.

The PERI project has included the creation of application-specific code generators for three programs. They believe that "application-specific auto-tuners are the most practical near-term approach for obtaining high performance on multicore systems" [61]. Their generated versions of application-specific algorithms are tested to determine the one with optimal performance [62, 63, 61, 64]. These existing approaches all show that autotuning is indeed useful for finding good implementations for single shared-memory nodes, that the optimization spaces are large, that the optimal configurations vary between different types of processors, and that

the actual program performance space is too complex to be fully understood and controlled by an application programmer. From a software engineering approach, it has been argued that "autotuning is indispensable, as manually tuning thread assignment, number of pipeline stages, size of data partitions and other parameters is difficult and error prone. ... Tunable architectural patterns with parallelism at several levels need to be discovered" [54]. This thesis work has proposed methods for tuning parallel programs at a high level of parallelism, namely across many distributed memory nodes.

Various techniques have been developed for controlling the grain size or scheduling mechanisms for dynamic task parallelism models. For example, various efforts have found that it is important to create the appropriate number of threads for a divide & conquer style parallel program [32, 65, 66, 67]. The techniques involve instrumenting trial runs or the beginnings of a run in order to build models of the execution times for tasks at each level in a recursion tree. Then the models allow the further decompositions of the program to be of a desired grain size [32, 67]. The issue of automatically decomposing a problem is particularly important in the field of parallelizing compilers [68, 69]. Various feedback guided dynamic techniques have been developed to adjust the grain size for the blocks in the decomposition of parallel loops [70, 71, 72, 73]. Although these techniques are of great importance to parallel programming for single-node shared memory systems such as upcoming multicore desktops when using languages or models such as Cilk or OpenMP, these techniques do not address many of the difficulties that are encountered running on distributed-memory systems that do not use a single task scheduler.

The TADL project deals with software engineering aspects of single-node multicore autotuners. Programs are written using their frameworks and compiler, while an associated autotuner tunes the parameters found in the framework. The TADL compiler can automatically instrument programs written in TADL script. The tunable parameters, which are annotated with applicable ranges, are adjusted by three plugin optimizers: random, swarm, and hill climbing. All tuning is performed entirely offline from run to run [74, 75, 76, 77, 78]. One main benefit of this system is that the TADL scripting language shortens the amount of code written by the programmer, hopefully improving productivity. The main downside is that only certain patterns or frameworks are provided, and the programmer must compose

a program from the existing patterns. The TADL project recently started investigating how idle time measurements might be used to adjust *replication ratios*, but this work is not yet published.

An alternative approach for automatically improving the performance of a code is to use a feedback guided compiler to iteratively recompile a program to test compiler optimizations in the hopes of eventually improving its performance [79].

13.2 Tuning Large-Scale Distributed-Memory Applications

Recently, performance tuning and optimization has been an important research field for new computing systems with large processor core counts. No unified solution has been found for effectively programming the new multicore, heterogeneous, and massively parallel machines. The complexities of the systems, both in the software development ecosystem and the complex architectures, can hinder the performance of even simple applications.

It has previously been shown that communication libraries benefit from effectively choosing between multiple algorithms or protocols for a single communication pattern [80, 81, 82]. Algorithms and protocols have various performance tradeoffs involving latencies, latency tolerance, total communication bandwidth, applicability to specific network topologies, and transient memory overheads. In the *inspector-executor paradigm*, accesses to distributed data structures are observed at runtime by an inspector then an optimized communication schedule is created for subsequent uses in an HPF program [83, 84].

13.2.1 Autopilot

The tuning system with the most similarities to the research in this dissertation is called Autopilot [85]. Autopilot is a system that gathers performance data for grid applications through *sensors*, either directly accessing program variables or calling functions that have been added to a program. Information provided by these sensors can be analyzed by a set of fuzzy logic rules to trigger *actuators* that adapt the behavior of a program.

It is possible in the autopilot system to adapt a program's use of resources such as a filesystem by changing its access patterns or prefetching methods. The autopilot system does not attempt to allow programmers to express the behavioral effects that would occur when an actuator is executed. Thus the rules that trigger actuators need to be written to handle the specific set of sensors provided by the program. So far, only a prototype implementation of Autopilot exists and the only published use cases of Autopilot are in the tuning of prefetching and caching parameters in a parallel file system for the PPFS II I/O benchmark [86] and for rescheduling grid applications when performance contracts are violated [87].

13.2.2 Active Harmony

Relatively little has been published about automatic, dynamic tuning for large-scale distributed memory systems. One of the few projects to attempt to automatically tune applications on the scientific computing world's distributed-memory systems is Active Harmony [88].

Active Harmony supports various automatic tuning idioms:

- Sequential programs and their libraries can expose tunable parameters [89, 90].
- Applications (possibly parallel) can expose tunable parameters that are tuned automatically in an offline manner across program runs [88], or in a manner that reuses performance data and configurations from previous runs to speedup convergence in the "black-box tuning process" [91].
- MPI applications and libraries can expose tunable parameters that are tuned automatically in an online manner from one iteration to a subsequent similar iteration [89].
- Parallel applications or their libraries expose tunable parameters that are tuned "in parallel" with different configurations tested across processors within a single application iteration [92, 93].
- Migration of threads or procedures for load balancing purposes using measured communication costs and processor loads in the harmony

SDSM scheduler on up to 8 processors [94].

Published application tuning scenarios with Active Harmony are listed in table 13.1. The tuning of these applications follows a standard auto-tuning approach, namely, evaluating different configurations over one or more program iterations, and eventually converging on a configuration that maximizes an objective function. The assumptions are that the programs will have little variability from iteration to iteration, except for short transient spikes, so eventually the objective function will be optimized. Almost always the objective function is simply the execution time for an application iteration, but Active Harmony is capable of optimizing other objective functions, of which the literature describes two: compressed file sizes (the effectiveness of compression algorithms), and memory usage. These objective functions are optimized directly using actual measurements of the objective function. The tuning never takes into account any other type of behavioral measurements of the application (except for load balancing), as is done in this dissertation.

13.2.3 MATE

The third similar parallel automatic tuning framework is called *MATE* [96, 97, 98, 99, 100, 101, 102]. A main goal of the MATE system is to facilitate the automatic tuning of parallel applications without modifying the application source code at all. Thus their "automatic", also called "black-box", approach uses DynInst to rewrite the memory of a running program in order to modify function calls or variables within a program [102]. This approach allows functions to be replaced, eliminated, inserted, or instrumented in the running program. Additionally, modifications can be made to function parameters when function calls are intercepted. This approach explicitly addresses the types of adaptations that do not benefit from application level knowledge.

The second type of automatic tuning approach supported by the MATE system is called the "cooperative approach." It requires that "developers must prepare the application for the possible changes by modifying its source code." To do this, the developer is required to express "what should be measured in the application, what performance model should be used, and what can be changed in the application" [102]. This requires a

Application	Measurements	Objective Function	Number of Processors	Publication
Table Abstraction Library	Time	Time	1	[89]
Compress Library	Time Compression Ratio	Function of: Time, Compression Ratio	1	[89]
Active Data Repository Image Reconstruction	Time	Time	32	[89]
Table Abstraction Library	Memory	Memory	1	[90]
GS2[a]	Time	Time	64	[95]
PETSc	Time	Time	32	[88]
POP	Time	Time	32 or 480	[88]
GS2	Time	Time	128	[88]
HPL	Time	Time	16	[92]
PSTSWM	Time	Time	32	[92]
POP	Time	Time	32	[92]

Table 13.1: Some published use cases of Active Harmony

[a]Used simulation to analyze PRO tuning algorithm.

large amount of expertise both in knowledge of the application and also in the appropriate methods for creating tuning mechanisms for each type of performance problem. In the literature, two example use cases have been provided. Both use cases adapt parameters used by a master process as it assigns tasks to workers. Neither case deals with the issues of synchronizing updates to multiple processors. There is almost no overlap in practice between the published uses of MATE and the work described in this thesis, except for the conceptual idea of gathering measurements and using them to reconfigure a program.

One published use case of cooperative tuning in MATE is to balance the load across multiple heterogeneous workers in a master/worker system [99, 100, 103, 101, 102]. Such load balancing is not the focus of this dissertation, as much research has been performed in the past on load balancing. The other use case of cooperative tuning finds an optimal number of worker processes in a master/worker application [96, 99, 103, 102]. The number of worker tasks, and correspondingly compute nodes, is increased as long as it will result in a faster program. Expanding to larger amounts of resources is not the type of optimization that is the focus of this dissertation. In this dissertation, the goal is to run an application as fast as possible on a fixed set of dedicated processors.

13.2.4 Critical Paths

The tuning approach proposed in this dissertation makes extensive use of observed performance observations, both simple statistical measurements and complicated critical-path profiles. Some work has been done in the past to detect critical paths as a parallel program runs. The seminal work for critical path profiling describes how a critical path can be created as a distributed PAG [18]. Subsequent extensions to that seminal work detects critical paths in PVM programs by using a two phase approach [10, 16, 19, 20]. In the first phase, messages are sent along the edges of the program activity graph. This is done by sending a second message anytime a PVM communication call is made. When a detailed critical-path profile is needed, the critical-path profile is reconstructed through a backwards traversal to gather information about the tasks performed along the path. A similar

technique works with MPI programs [11]. These approaches are only used to produce profiles of the critical paths that are used in offline visualization or offline performance analysis tools. This dissertation demonstrates that critical path information can be used in a novel way, namely, to perform online, automatic tuning of parallel programs.

13.2.5 Other

Some other projects have investigated other types of dynamic adaptation of parallel distributed memory applications. These projects do not attempt to reconfigure running programs as is proposed in this dissertation. One system attempts to adapt its vector communication algorithms in response to an important factor, the load on the NIC within an SMP node [104]. Some language specific communication mechanisms have also been tuned at runtime. In the *inspector-executor paradigm*, collective accesses to distributed data structures are observed at runtime by an inspector then an optimized communication schedule is created for all subsequent uses of the collective operations in an HPF program [83, 84].

13.3 Novelty of Control Points for Automatic Tuning

The research presented in this dissertation is different than other existing autotuning approaches in three important ways:

- *The tuning framework encourages applications to provide information about the behavioral effects of each control point.*
- *The tuning framework observes characteristics of the executing program, other than just its overall performance or memory consumption, to make intelligent tuning decisions.*
- *The utility of the tuning framework has been demonstrated for use on large-scale distributed memory parallel platforms, not just in single node shared-memory programs.*

CHAPTER 14

Future Work

This dissertation is a part of a wider research effort to investigate the utility of automatic, dynamic adaptations of parallel programs on large distributed-memory platforms. The use of control points is one new piece of this approach that will continue in the future as these wider research efforts continue. As this dissertation is only a part of this larger effort, some problems and questions left open by this dissertation will be revisited in the future.

Although this paper proposes a large catalog of possible control points in chapter 5, many of them have not yet been examined fully within the context of real applications. In the future, application developers will incorporate more of these control points in their applications and corresponding new tuning algorithms will be developed and evaluated.

The control point tuning framework has been designed with respect to the plan of tuning of many control points within a single application as it runs. If multiple tuning algorithms are enabled then a random choice is made from among their plans generated by the different algorithms. In the future, an obvious next step is to develop a metric that specifies the predicted benefits for each plan. Then a more informed choice could be made, possibly using a weighted random choice. Or the configuration with greatest predicted benefit could be chosen for future examination. A completely different system for choosing the new planned configurations would be to perform different types of adaptations as the program runs. For example, data decomposition schemes are adjusted early on in the program while less impacting load balancing parameters are adjusted throughout the remainder of the program run. Designing such a system correctly requires a good understanding of

each possible tuning scheme. This dissertation attempts to improve the understanding of various independent tuning schemes so that in the future, more elaborate schemes can be developed.

As parallel languages become increasingly more complicated, it is conceivable that compilers would be able to generate control points, or insert other types of useful instrumentation into a parallel program. The languages themselves could also provide better interfaces for exposing information about static control flow or data flow. With dynamic compilation in managed languages such as Java, control points could additionally modify the behavior of the JIT optimizer.

The case studies described in this dissertation are HPC style applications. Other fields of applications might benefit from the use of control points. It will become especially important to develop control point systems for desktop applications if future desktop computer architectures become similar to the distributed memory parallel systems of today.

APPENDIX A

Derivation of Optimal Load Balancing Period

Assume that the execution time for a step that occurs in the perfectly load balanced state is the constant t_{min}. Assume further that the execution times for steps degrade at a rate of $m\frac{\text{sec}}{\text{step}}$. Assume the cost of each load balancing operation is a constant value c sec. Assume that after each load balancing step has an execution time of t_{min}. Below we prove that the optimal load balancing scheme performs load balancing operations at fixed intervals of steps. Further, the optimal choice for load balancing period is $\sqrt{\frac{2c}{m}}$.

Let $n_i, i \in \{1, 2, ...q\}$ represent the number of steps between the $i-1^{th}$ and i^{th} load balancing operations. Then $N = \sum_{i=1}^{q} n_i$ represents the total number of steps in the execution of the program of which q steps involve a load balancing operation.

The total execution time for the program is thus:

$$E_{total} = \sum_{i=1}^{q} \left(c + \sum_{j=1}^{n_i} \left(t_{min} + (i-1)\, m \right) \right) \tag{A.1}$$

$$= q \cdot c + \left(t_{min} - \frac{m}{2} \right) \sum_{i=1}^{q} n_i + \frac{m}{2} \sum_{i=1}^{q} {n_i}^2 \tag{A.2}$$

We prove by contradiction that the minimum value of the execution time E occurs when $0 \leq max\{n_1, n_2, \ldots, n_q\} - min\{n_1, n_2, \ldots, n_q\} \leq 1$, that is, all the different n_i are all nearly equal. For the sake of contradiction, assume $n_i \geq n_j + 2$ for some i, j. Then we can select different steps for the load

balancing operations by decreasing n_i to $n_i - 1$ and increasing n_j to $n_j + 1$. The total execution time for this configuration would differ from the original by the following amount:

$$\begin{aligned} & \left(t_{min} - \frac{m}{2}\right)\left((n_i - 1) - n_i + (n_j + 1) - n_j\right) + \\ & \frac{m}{2}\left((n_i - 1)^2 - n_i^2 + (n_j + 1)^2 - n_j^2\right) && \text{(A.3)} \\ = \;& m\left(-n_i + n_j + 1\right) && \text{(A.4)} \\ \leq \;& m\left(-2 + 1\right) = -m && \text{(A.5)} \end{aligned}$$

Thus because $m > 0$, the performance of this new set of load balancing steps will improve the performance of the program by m. Then we have contradicted the fact that original configuration was optimal. Hence any optimal execution must not have $n_i \geq n_j + 2$ for any i, j. Thus because the q values n_i must sum to N, it is necessary that $n_i \in \{\lfloor \frac{N}{q} \rfloor, \lceil \frac{N}{q} \rceil\} \forall i$.

Now the total execution time can be rewritten as:

$$\begin{aligned} E_{total} &= q \cdot c + \left(t_{min} - \frac{m}{2}\right)\sum_{i=1}^{q} n_i + \frac{m}{2}\sum_{i=1}^{q} {n_i}^2 && \text{(A.6)} \\ &\approx q \cdot c + \left(t_{min} - \frac{m}{2}\right)N + \frac{m}{2}\sum_{i=1}^{q}\left(\frac{N}{q}\right)^2 && \text{(A.7)} \\ &= q \cdot c + \left(t_{min} - \frac{m}{2}\right)N + \frac{mN^2}{2q} && \text{(A.8)} \end{aligned}$$

Next we find the q that minimizes the total execution time by finding the q such that $\frac{d}{dq}\left(E_{total}\right) = 0$

$$\begin{aligned} \frac{d}{dq}\left(E_{total}\right) &= 0 \implies && \text{(A.9)} \\ \frac{d}{dq}\left(q \cdot c + \left(t_{min} - \frac{m}{2}\right)N + \frac{mN^2}{2q}\right) &= 0 \implies && \text{(A.10)} \\ c - \frac{mN^2}{2q^2} &= 0 \implies && \text{(A.11)} \\ q &= \sqrt{\frac{mN^2}{2c}} && \text{(A.12)} \end{aligned}$$

Thus the minimal execution time occurs for $q = \sqrt{\frac{mN^2}{2c}}$, which yields an optimal load balancing period of $\frac{N}{q} = \sqrt{\frac{2c}{m}}$.

REFERENCES

[1] Isaac Dooley, Chao Mei, Jonathan Lifflander, and Laxmikant Kale. A study of memory-aware scheduling in message driven parallel programs. In *Proceedings of 17th Annual International Conference on High Performance Computing*, 2010.

[2] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, Jan 2003.

[3] Parry Husbands and Katherine Yelick. Multi-threading and one-sided communication in parallel LU factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[4] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[5] Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, IL. *The Charm++ Programming Language Manual, (Version 6.1.3)*, 2010.

[6] 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.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.

[8] Toshio Endo, Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. High performance LU factorization for non-dedicated clusters. In *Proceedings of the 4th International Symposium on Cluster Computing and the Grid (CCGrid 04)*, pages 678–685, 2004.

[9] Isaac Dooley and Laxmikant Kale. Detecting and using critical paths at runtime in message driven parallel programs. In *12th Workshop on Advances in Parallel and Distributed Computing Models (APDCM 2010) at IPDPS 2010.*, April 2010.

[10] J.K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):1029–1040, Oct 1998.

[11] M Schulz. Extracting critical path graphs from mpi applications. *Cluster Computing, 2005*, pages 1 – 10, Sep 2005.

[12] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[13] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[14] 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.

[15] Parry Husbands and Katherine Yelick. Multi-threading and one-sided communication in parallel lu factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[16] Jeffrey K. Hollingsworth. An online computation of critical path profiling. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 11–20, New York, NY, USA, 1996. ACM.

[17] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[18] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[19] J. K. Hollingsworth and B. P. Miller. Parallel program performance metrics: a comprison and validation. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 4–13, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[20] J Hollingsworth and Barton Miller. Slack: A new performance metric for parallel programs. *cs.umd.edu*, Jan 1994.

[21] 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.

[22] Eric Bohm, Sayantan Chakravorty, Pritish Jetley, Abhinav Bhatele, and Laxmikant V. Kale. CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm . In *Proceedings of International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, August 2009.

[23] M. J. Sottile and R. G. Minnich. Supermon: a high-speed cluster monitoring system. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 39–46, 2002.

[24] 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.

[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[26] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[27] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–282, New York, NY, USA, 2009. ACM.

[28] J A Nelder and R Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[29] A. Gursoy and L.V. Kale. High-level support for divide-and-conquer parallelism. In *Proceedings of Supercomputing '91*, pages 283–292, November 1991.

[30] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 207–216, Santa Barbara, California, July 1995. MIT.

[31] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[32] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[33] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[34] B. Shen, M. Hubler, G. H. Paulino, and L. J. Struble. Functionally graded fiber-reinforced cement composite: Processing, microstructure, and properties. *Cement and Concrete Composites*, 30(8):663–673, 2008.

[35] Matthew T. Tilbrook, Robert J. Moon, and Mark Hoffman. Finite element simulations of crack propagation in functionally graded materials under flexural loading. *Engineering Fracture Mechanics*, 72(16):2444 – 2467, 2005.

[36] M. Wosko, B. Paszkiewicz, T. Piasecki, A. Szyszka, R. Paszkiewicz, and M. Tlaczala. Application and modeling of functionally graded materials for optoelectronic devices. In *Photonics and Microsystems, 2005. Proceedings of 2005 International Students and Young Scientists Workshop*, pages 87–89, July 2005.

[37] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235, September 2006.

[38] Sayantan Chakravorty, Aaron Becker, Terry Wilmarth, and Laxmikant V. Kalé. A Case Study in Tightly Coupled Multi-Paradigm Parallel Programming. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC '08)*, 2008.

[39] NVIDIA. *CUDA 2.0 Reference Manual*. NVIDIA Corporation, Santa Clara, CA, USA, June 2008.

[40] NVIDIA. *CUDA 2.0 Programming Guide*. NVIDIA Corporation, Santa Clara, CA, USA, June 2008.

[41] Aaron Becker, Isaac Dooley, and Laxmikant Kale. Flexible hardware mapping for finite element simulations on hybrid cpu / gpu clusters. In *SAAHPC : Symposium on Application Accelerators in HPC*, July 2009.

[42] Gengbin Zheng, Michael S. Breitenfeld, Hari Govind, Philippe Geubelle, and Laxmikant V. Kale. Automatic dynamic load balancing for a crack propagation application. Technical Report 06-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2006.

[43] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[44] J. Nagib, C. Árebe, A. Beguelin, and Bruce Lowekamp. Dome: Parallel programming in a distributed computing environment. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[45] Rupak Biswas, Leonid Oliker, Sajal K. Das, and Daniel J. Harvey. Portable parallel programming for the dynamic load balancing of unstructured grid applications. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 338–342, Washington, DC, USA, 1999. IEEE Computer Society.

[46] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[47] Isaac Dooley, Chao Mei, Jonathan Lifflander, and Laxmikant Kale. A study of memory-aware scheduling in message driven parallel programs. In *PPL Technical Reports 2010*, number 10-05, March 2010.

[48] B Hendrickson and E Womble. The torus–wrap mapping for dense matrix calculations on massively parallel computers. *Siam J Sci Stat Comp*, Jan 1994.

[49] Edgar Solomonik and Laxmikant V. Kale. Highly Scalable Parallel Sorting. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[50] L.V. Kale and Sanjeev Krishnan. Medium grained execution in concurrent object-oriented systems. In *Workshop on Efficient Implementation of Concurrent Object Oriented Languages, at OOPSLA 1993*, September 1993.

[51] Vikram A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference (5th DMCC'90)*, volume II, Architecture Software Tools, and Other General Issues, pages 994–999, Charleston, SC, April 1990. IEEE.

[52] 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.

[53] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.

[54] Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, New York, NY, USA, 2008. ACM.

[55] D H Bailey, J Chame, C Chen, J Dongarra, M Hall, J K Hollingsworth, P Hovland, S Moore, K Seymour, J Shin, A Tiwari, S Williams, and H You. Peri auto-tuning. *Journal of Physics: Conference Series*, 125:012089, 2008.

[56] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[57] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005.

[58] M. Frigo and S.G. Johnson. Fftw: an adaptive software architecture for the fft. *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, 3:1381–1384 vol.3, May 1998.

[59] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3 – 35, 2001.

[60] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[61] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[62] S Williams, K Datta, J Carter, L Oliker, J Shalf, K Yelick, and D Bailey. Peri - auto-tuning memory-intensive kernels for multicore. *Journal of Physics: Conference Series*, 125:012038 (15pp), 2008.

[63] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[64] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14, April 2008.

[65] Michael Voss and Rudolf Eigenmann. Dynamically adaptive parallel programs. In *ISHPC '99: Proceedings of the Second International Symposium on High Performance Computing*, pages 109–120, London, UK, 1999. Springer-Verlag.

[66] Hans-Wolfgang Loidl and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1995.

[67] Prechelt. Efficient parallel execution of irregular recursive programs. *Parallel and Distributed Systems, IEEE Transactions on*, 13(2):167 – 178, 2002.

[68] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–83, Jan 1999.

[69] Bernd Freisleben and Thilo Kielmann. Automatic parallelization of divide-and-conquer algorithms. In *CONPAR*, volume 634 of *Lecture Notes in Computer Science*, pages 849–850. Springer, 1992.

[70] Tatiana Tabirca, Len Freeman, Sabin Tabirca, and Laurence Tianruo Yang. Feedback guided dynamic loop scheduling: convergence of the continuous case. *The Journal of Supercomputing*, 30(2):151–178, November 2004.

[71] Oleh Olkhovskyy. Feedback guided scheduling for two-dimensional loops. In *UKHEC Technical Reports*, Jan 2001.

[72] D.J. Hancock, J.M. Bull, R.W. Ford, and T.L. Freeman. An investigation of feedback guided dynamic scheduling of nested loops. In *International Workshops on Parallel Processing, 2000. Proceedings*, pages 315–321, 2000.

[73] J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 377–382, London, UK, 1998. Springer-Verlag.

[74] Christoph A. Schaefer, Victor Prankratius, and Walter F. Tichy. Engineering parallel applications with tunable architectures. In *International Conference on Software Engineering (ICSE)*, May 2010.

[75] Thomas Karcher, Christoph A. Schaefer, and Victor Pankratius. Auto-tuning support for manycore applications - perspectives for operating systems and compilers. *Accepted for ACM SIGOPS Operating System Review. Special Issue on the Interaction among the OS, Compilers, and Multicore Processors*, April 2009.

[76] Christoph A. Schaefer. Reducing search space of auto-tuners using parallel patterns. In IEEE Computer Society, editor, *Proceedings of the 2nd ICSE Workshop on Multicore Software Engineering*, pages 17–24, May 2009.

[77] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In Springer Berlin / Heidelberg, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume LNCS, pages 9–20, August 2009.

[78] Otilia Werner-Kytölä. *Automatic Tuning of the Degree of Parallelism of Programs.* publikation, IPD Tichy, University of Karlsruhe, Germany, October 1999.

[79] Zhelong Pan and Rudolf Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2004. IEEE Computer Society.

[80] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[81] Ahmad Faraj, Xin Yuan, and David Lowenthal. Star-mpi: self tuned adaptive routines for mpi collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM.

[82] M.J. Koop, T. Jones, and D.K. Panda. Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[83] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4):303–312, 1990.

[84] Siegfried Benkner, Piyush Mehrotra, John Van Rosendale, and Hans Zima. High-level management of communication schedules in hpf-like languages. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 109–116, New York, NY, USA, 1998. ACM.

[85] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst.*, 18(1):175–187, 2001.

[86] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, Chicago, IL, July 1998.

[87] Daniel Reed and Celso Mendes. Intelligent monitoring for adaptation in grid applications. *Proceedings of the IEEE*, 93(2):426 –435, feb. 2005.

[88] I-Hsin Chung and Jeffrey K. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. In *International Symposium on High Performance Distributed Computing (HPDC)*, June 2006.

[89] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[90] I-Hsin Chung and Jeffrey K. Hollingsworth. Runtime selection among different api implementations. *Parallel Processing Letters*, 13(2):123–134, 2003.

[91] I-Hsin Chung and Jeffrey K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 30, Washington, DC, USA, 2004. IEEE Computer Society.

[92] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Comput.*, 35(8-9):475–492, 2009.

[93] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[94] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2(3):195–205, 1999.

[95] Vahid Tabatabaee, Ananta Tiwari, and Jeffrey K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 57, Washington, DC, USA, 2005. IEEE Computer Society.

[96] Anna Morajko, Eduardo César, Tomàs Margalef, Joan Sorribes, and Emilio Luque. Dynamic performance tuning environment. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 36–45, London, UK, 2001. Springer-Verlag.

[97] Anna Morajko, Oleg Morajko, Josep Jorba, Tomás Margalef, and Emilio Luque. Dynamic performance tuning of distributed programming libraries. In *ICCS'03: Proceedings of the 2003 international conference on Computational science*, pages 191–200, Berlin, Heidelberg, 2003. Springer-Verlag.

[98] Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque. Mate: toward scalable automated and dynamic performance tuning environment. In *Lecture Notes in Computer Science*, volume 3149, pages 98–107. Springer Berlin / Heidelberg, 2004.

[99] E. Cesar, J.G. Mesa, J. Sorribes, and E. Luque. Modeling master-worker applications in poetries. In *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, pages 22–30, 26 2004.

[100] Anna Morajko, Paola Caymes, Tomàs Margalef, and Emilio Luque. Automatic tuning of data distribution using factoring in master/worker applications. In *Lecture Notes in Computer Science*, volume 3515, pages 132–139. Springer Berlin / Heidelberg, 2005.

[101] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. Mate: Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurr. Comput. : Pract. Exper.*, 19:1517–1531, 2007.

[102] Anna Morajko, Tomàs Margalef, and Emilio Luque. Design and implementation of a dynamic tuning environment. *J. Parallel Distrib. Comput.*, 67(4):474–490, 2007.

[103] Anna Morajko, Eduardo Cèsar, Paola Caymes-Scutari, Tomàs Margalef, Joan Sorribes, and Emilio Luque. Automatic tuning of master/worker applications. In *Lecture Notes in Computer Science*, volume 3648, pages 95–103. Springer Berlin / Heidelberg, 2005.

[104] Costin Iancu and Steven Hofmeyr. Runtime optimization of vector operations on large scale smp clusters. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 122–132, New York, NY, USA, 2008. ACM.