PICS - A PERFORMANCE-ANALYSIS-BASED INTROSPECTIVE CONTROL SYSTEM TO STEER PARALLEL APPLICATIONS

BY

YANHUA SUN

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, 2015

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor William D. Gropp
Professor Andreas Kloeckner
Doctor Pavan Balaji, Argonne National Laboratory

Abstract

Parallel programming has always been difficult due to the complexity of hardware and the diversity of applications. Although significant progress has been achieved over the years, attaining high parallel efficiency on large supercomputers for various applications is still quite challenging. As we go beyond the current scale of computers to those with peak capacities of an ExaFLOP/s, it is clear that an introspective and adaptive runtime system (RTS) will be critical to reduce programmers' tuning efforts by automatically handling the complexities of applications and machines. This is the motivation for my research on a Performance-analysis-based Introspective Control System - PICS. PICS intelligently steers parallel applications and runtime system configurations to achieve desired goals by utilizing expert knowledge to analyze performance data and adaptively reconfiguring applications.

This thesis designs a holistic introspective control system for automatic performance tuning that combines the real-time performance analysis and performance steering to effectively automate the optimization. A few techniques are explored to make the parallel runtime system and applications more adaptive and controllable. Control points are defined for applications to interact with PICS. Decision tree based automatic performance analysis is implemented to significantly reduce the search space of multiple configurations. Parallel evaluation and sampling techniques are exploited to reduce the overhead of the system and to improve its scalability. In addition, the result of automatic performance analysis can be visualized to help developers manually tune their applications. The utility of PICS is demonstrated with several benchmarks and real-world applications.

To my parents Guiyuan Sun and Guifang Dong for their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern parallel computer systems are becoming extremely complex because of complicated network topology, hierarchical storage system and heterogeneous processing units, etc. On the other hand, physics simulation models are expanded to match experimental results, which greatly increase the complexity of computation. Although computer scientists focus on computer models and implementation while physical scientists concentrate on physics models, they share a common goal - to maximize their application performance using available resources. This naturally raises the question of how to obtain the best performance of applications on the existing and future platforms.

In the past, the ideal solution for parallel programming sought automatic parallelization, that is to let compiler automatically generate an efficient parallel program from a sequential program without involving any efforts of the developers. Despite decades of work by compiler researchers, automatic parallelization has only achieved very limited success [1]. This is mainly due to the high complexity of compiler, the diversity of parallel programs and also the low efficiency of generated code. Nowadays, the mainstream parallel programming languages remain either explicitly parallel or partial implicitly parallel. For example, the most popular parallel programming model – MPI (Message Passing Interface [2, 3]) automates very little. Developers need to take care of every detail of parallelization. OpenMP [4] is the most widely used parallel programming model of the partial parallelization, in which the developers provide the compiler directives for partial automatic parallelization. Besides

these two parallel programming models, the other major ongoing work tries to provide more features in programming models and underlying runtime systems to reduce the programmers' burden as well to improve the parallel performance. For example, communication optimization in Partitioned Global Address Space (PGAS) is supported by GASNET runtime [5]. Cilk is designed as an efficient runtime system by incorporating the work stealing scheduler [6, 7]. The new popular languages of Cray Chapel [8] and IBM X10 [9] both have their own powerful runtime systems to do partial automation. The research in this thesis belongs to this category of parallel programming.

The research in this thesis aims at improving both the parallel program performance and productivity. I emphasize two principles to achieve this goal. The first principle is that the reasonable application-specific knowledge provided to a runtime system may greatly reduce the complexity of the runtime system and improve the overall performance. For example, when tuning an application, it can be quite useful for the runtime system to know the start and the end of one simulation time step. It is complicated and potentially inaccurate for the runtime system to automatically detect the execution pattern of the application. However, it is easy for application developers to provide such information to the runtime system. Therefore, we believe application developers should provide some hints to help the runtime system to make decisions to improve performance. Of course, these hints should be designed such that they do not increase too much of the developers' programming burden. The second principle is that a powerful runtime system is crucial to enhance the performance of programs. This is due to that the runtime system has both the knowledge about application execution behaviors and the underlying architectures. This empowers the runtime system to make better decisions to improve performance. Overall, we believe that the best performance can be only achieved by efforts of both applications developers and the underlying runtime system of the programming models.

Based on the above two principles, this thesis proposes a design of an introspective control system that supports dynamic performance steering. The application developers provide tuning knobs and related knowledge to guide the control system. The tuning knobs are represented as control points, which are the interface between the control system and the

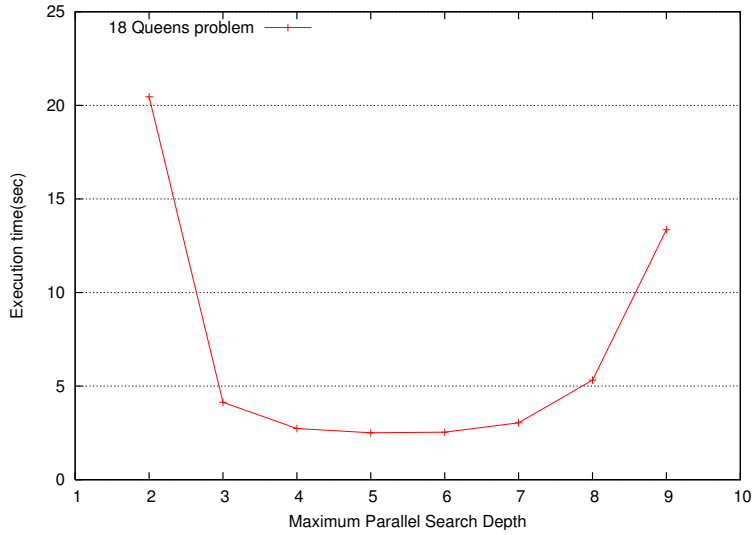
applications or runtime system. Meanwhile, the control system collects the program behaviors, analyzes the performance and dynamically steer the program execution. Expertise knowledge rules based performance analysis are applied to accelerate the process of automatic performance steering at runtime. The research also investigates how the parallel runtime system and applications can become more adaptive and controllable by exposing abstractions to the introspective control system. In order to make approach more feasible in practice, scalable techniques are developed so as to handle large-scale run analysis and steering. We have developed PICS - Performance-analysis-based Introspective Control System - to integrate these techniques [10]. We demonstrate the utility of this approach with several applications and show its effectiveness.

A few examples are discussed below to show how the overall performance is affected by different configurations of applications and the runtime system. This motivates the necessity and importance of the performance steering.

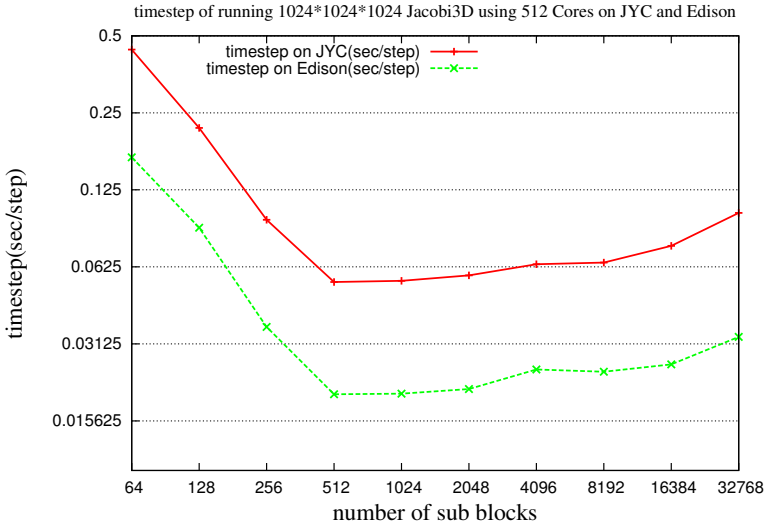
1.1 Motivation

The first example involves the grain size of parallel tasks, which greatly affects the parallel performance. This is a common problem in parallelizing applications, including both iterative scientific applications and state space search applications. In general, when the tasks are too large-grained, there is not enough parallelism to keep all resources busy. As a result, resources are idle, leading to performance degradation. On the other hand, when the tasks are too fine-grained, the overhead associated with each task becomes too high, leading to excessive useless work. The overhead usually includes the time of spawning tasks, scheduling tasks, communication, etc. Therefore, the grain size should be within a reasonable range to obtain good performance. Therefore, choosing the optimal grain size is necessary and useful.

Figure 1.1 shows how the performance varies with the grain size in two types of parallel problems. Figure 1.1(a) is the result for solving 18-Queen problem using different values, which is the depth threshold of executing sequential tasks versus parallel tasks. It can be easily seen that the optimal or close-to-optimal performance is achieved only with a



(a) 18-Queen problems



(b) Jacobi3D running on 512 cores

Figure 1.1: Performance variance due to grain size

few configurations while most of other configurations provide very bad performance. This problem represents the category of tree-structured state space search problems.

Figure 1.1(b) shows the execution time of one step of solving $1024 * 1024 * 1024$ Jacobi3D problem using different sub blocks running on 512 processor-cores. When the number of sub blocks is less than 512, insufficient parallelism leads to larger amount of idle time. On the other end, excessive sub blocks also lead to bad performance due to high overhead. Both examples demonstrate the importance of choosing the proper grain size. The grain size problems are also found in real-world scientific applications. For example, in molecular dynamics applications, the number of atoms in each task can greatly affect the performance when running on different number of processors. The complexity of real-world applications makes it even more challenging to find the optimal grain size.

The above example showed how the grain size affects the overall performance by affecting the degree of parallelism and the ratio of overhead. Moreover, the grain size can also affect performance in the means of cache effects. Figure 1.2 demonstrates such effect by running a Jacobi3D example on one core. It shows the execution time of one iteration using various decomposition schemes. Compared with only one single task, using more sub-tasks achieves better performance because the data of each block fits in cache. Besides the grain size, many other factors may affect the parallel performance, including memory contention, network contention, etc. Depending on the specific problem, these affects can interleave together and create even more challenges.

Not only computation, the configurations in communication can also significantly affect the overall performance. Figure 1.3 shows the execution time of transferring 1M bytes data by splitting into different small messages. In Figure 1.3(a), only communication is performed without any computation. It is seen that with the number of small messages increasing, the total time increases too. This is due to the latency of small messages so that the network bandwidth can not be fully utilized. As a result, transferring the data in one big chunk provides the best performance. In contrast, Figure 1.3(b) demonstrates the case of performing both computation and communication on the sender and receiver sides. Before data is sent out, the associated computation is carried out. Meanwhile when

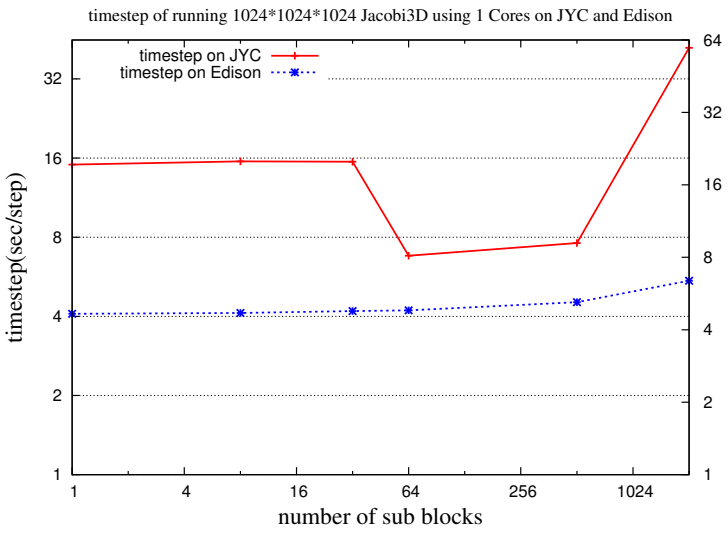
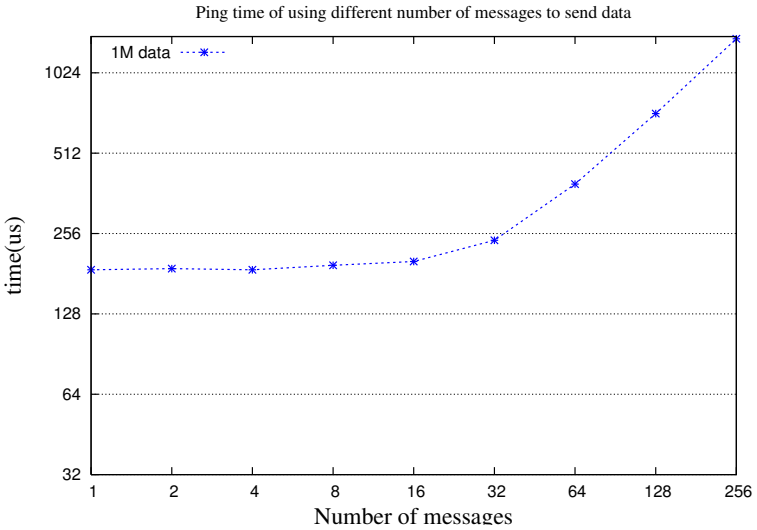


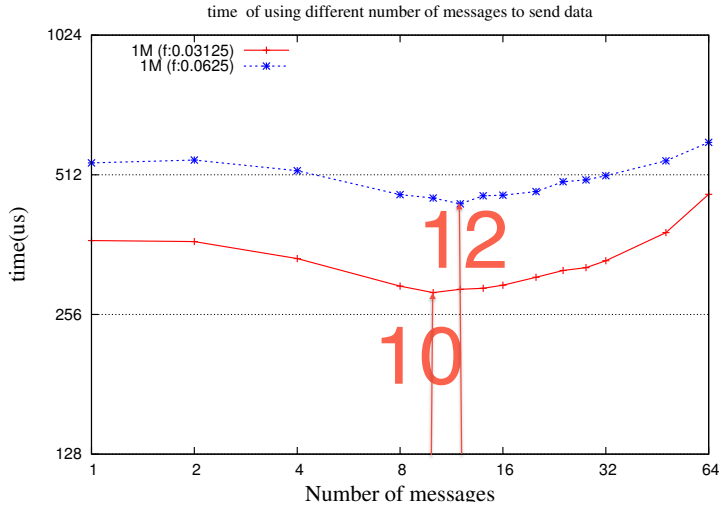
Figure 1.2: Timestep of Jacobi3D using different number of sub blocks on 1 core

data is received, computation is performed too. In this way, by splitting big data into small chunks, the computation can overlap with communication. This explains for the first part. When the number of chunks increases, the performance becomes better. However after a threshold, the overhead of communication over-weighs the benefit of overlapping so the performance drops. For the above two cases of transferring 1M bytes data, choosing the optimal number of chunks is a question. It needs to be adapt to the specific execution. This requires runtime automatic tuning.

Not only the application configurations can affect performance, but various configurations in the the runtime systems can also affect the overall performance. The collectives algorithms in MPI runtime are such examples [11]. Depending on the message size, the number of processors, etc, the runtime system is configured to choose different routing algorithms to achieve the best performance. Traditionally, the configurations for the runtime system are chosen based on a priori knowledge from experimental results. However, with the systems becoming more complex, the best configurations that work at a small scale might not work well at the large scale. Also the best configurations for some problems might not perform well for the others. For example, for the spanning tree algorithm used in reduction, the best branching factors vary for different application inputs, which is illustrated in Figure 1.4. It shows the time step of running Jacobi2D on 1024 processor-cores using different reduction



(a) Transfer 1M bytes data



(b) Transfer 1M bytes data with computation on sender and receiver

Figure 1.3: Communication variance

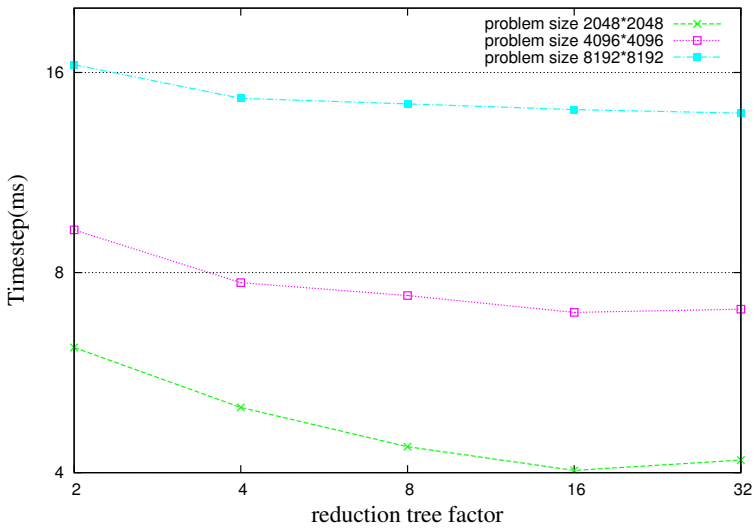


Figure 1.4: Timestep of Jacobi2d when using different reduction branching factors

branching factors. It can be seen that the optimal performance is almost 50% better than the worst.

Moreover, nowadays the hardware has evolved to be more controllable and adjustable. For example, the CPU frequency and memory frequency are tunable on Intel Sandy Bridge processors to satisfy power constraint or to maximize energy efficiency. Related work can be found in [12], where the authors illustrate how to reduce the overall energy cost by lowering CPU frequency. This makes it very practical to steer a runtime system for hardware to adapt to the applications to achieve the best performance.

1.2 Statement and Scope of Thesis

The thesis explores the hypothesis that, given a rich adaptive/introspective runtime system, and given a set of *control points* in the application as well as in the runtime system itself, it is possible to design a control system that can optimize application performance during execution.

The thesis investigates to steer the parallel applications running on large-scale supercomputers. The applications include both scientific applications with time steps and the state space search applications with tree structures. The runtime system underneath the appli-

cation has the features of over-decomposition, adaptivity, and migratablity. The hardware is mostly the large-scale supercomputers with multi-cores, high network bandwidth and low latency. Interconnection topology might also have impact on performance. Details are discussed in chapter 2.

1.3 Contribution of Thesis

In this thesis, an introspective control system has been developed to steer both runtime systems and applications. With the help of certain application specific knowledge, the control system can automatically find the optimal or close-to-optimal configuration to improve performance. This leads to the following research fronts:

1. This thesis studies the techniques of an introspective control system that supports dynamic performance steering. One major focus of this thesis is to study various rules to perform automatic analysis to identify problems. The depth and width of the expertise system significantly affects the overall performance.

2. We categorize the applications and propose their corresponding control points to reconfigure the applications. We identify many control points within the runtime system itself, and develop techniques for tuning them to optimize some aspect of the performance.

3. We exploit a decision tree based scheme that identifies program characteristics and correlate them to the effects of control points, so as to decide which control points to tune and in which direction. This is an attempt to eliminate wasteful combinatorial search over parameter space. The results of decision tree based analysis are used not only for performance tuning, but also can be visualized so as to guide developers to understand and manually tune their applications.

4. Various techniques including partial collection, parallel evaluations are proposed and developed to improve the scalability of the system.

5. All the techniques are implemented in the PICS framework. How the control system helps enhance real-world application performance is illustrated and discussed. PICS is evaluated in the context of synthetic benchmarks and real-world applications.

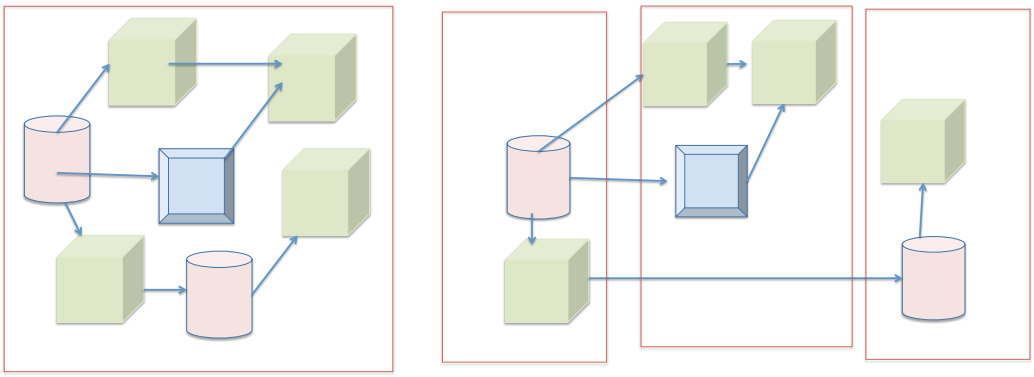
Chapter 2

Background

Nowadays more and more research focuses on the runtime system to improve parallel programming productivity and performance. The PICS is built for programming models with the runtime systems. The methodology in this thesis can be applied in various runtime system. However, in my research I focus on Charm++, which has been existing for almost 30 years and proven to be efficient and useful in real world applications. This chapter first describes this runtime system. One important part in PICS is to trace the performance data. The general tracing framework in Charm++ is presented then. Multiple benchmarks and applications are used to test various features of PICS in later chapters. The background of several parallel applications is also discussed.

2.1 Charm++ Runtime System

Charm++ [13] is a machine independent parallel programming language that runs on most shared and distributed memory machines. It employs an object-oriented approach to parallel programming. The programmer decomposes the problem into collections of objects that embody its natural elements. These objects are migratable and message-driven. Their number is independent of, but typically much larger than, the number of physical processors used to run the application. Figure 2.1(a) shows the user's view of objects while Figure 2.1(b) is the mapping of objects to physical resources. This over-subscription of processors is termed *object-based virtualization*. The migratable objects are



(a) User's view of objects (b) Machine's view of objects

Figure 2.1: Automatic mapping in Charm++

assigned to processors by the underlying adaptive runtime system. Objects communicate by asynchronously invoking an *entry method* on the callee object. Together, asynchronous messaging and object-based virtualization enable the dynamic overlap of communication and computation: a processor may overlap the messaging latency with not just the sending object's succeeding computation, but also with useful computation of other objects on that processor.

In Charm++'s execution model, each processor maintains a queue of messages to be delivered to Charm++ objects placed within it. This is called the *incoming queue.* There is a corresponding *outgoing queue* that holds messages generated by the processor's objects. The main control loop can be described simply as follows: a scheduler repeatedly picks a message from the incoming queue and retrieves the callee object and the entry method associated with the message; the entry method is then executed and the scheduler will pick the next message from the queue. Note that the entry method is executed to completion and is non-preemptive.

In order to control the applications, it is desirable for a runtime to be able to observe application behaviors and control applications. Charm++ provides such features, since Charm++ has the full control of program execution, including scheduling, communication and memory management. Therefore, it is feasible for the runtime to observe all the program activities and steer both the runtime and application behaviors. This programming model and runtime system has the following features.

Over-decomposition: Usually the problem is divided into more tasks than the physical resources. This has the advantages of better data locality, better task to resource mapping, and enables other features like load balancing. On the other hand, due to this over-decomposition, it naturally raises the questions of the optimal over-decomposition ratio. This makes automatic steering of grain size possible and necessary.

Migratablity: During the execution, tasks can migrate from one resource to the others. This makes dynamic load balancing, check pointing possible. This feature also makes it challenging to decide how and when the tasks are migrated in order to achieve the best performance.

Asynchrony: The computation continues without explicitly waiting for the remote tasks starting or finishing. The whole execution is driven by available messages with data inside. There is a central scheduling to be responsible for the task execution. Therefore, it raises the question of how tasks are scheduled. For example whether they are scheduled in FIFO or LIFO or priority order. The control system can help make the decision.

2.2 Control Point System

The control points system in Charm++ is first proposed and developed in Dooley's research [14]. His thesis describes the automatic tuning mechanisms within Charm++. He develops a preliminary framework that automatically reconfigures the behavior or structure of the program through one control points. Multiple benchmarks are studied to show that control points are useful mechanisms for dynamically reconfiguring applications to improve their performance.

This thesis is an extension with three significant novelties. The first novelty of this thesis is that the control system is based on decision tree based automatic performance analysis. The performance analysis results provide significant insight about what to steer and how to steer. The deeper the analysis is performed, the easier and more accurate the steering will be. Moreover, the rules to construct the decision tree are written in a plain text file. That makes expansion of the rules easy and clear. Therefore, one significant difference of this work is to perform steering based the expertise performance analysis

Listing 2.1: The core of the runtime scheduler

```
while(1) {
  msg = CsdNextMessage(&state);
  if (msg!=NULL) { /*A message is available-- process it*/
    if (isIdle) {isIdle=0;CsdEndIdle();}
    BEGIN_EXECUTION()
    SCHEDULE_MESSAGE
    END_EXECUTION()
  }
  else
  {
    if (!isIdle)
    {
      isIdle=1;
      CsdBeginIdle();
    }
    else
      CsdStillIdle();
  }
}
```

system. The second difference is that the steering framework can be applied to both of the application reconfiguration and runtime reconfiguration while most other work only reconfigures applications. Thirdly, this thesis proposes and develops scalable approaches to handle large-scale runs. The performance analysis and steering can be carried out in scalable way. The configurations of control points can be evaluated in parallel. This is discussed in detail in Chapter 6.

2.3 Charm++ Tracing Framework

In order to perform analysis of execution, performance data need to be collected. Our approach is based on the tracing framework in the runtime system. As described earlier, the runtime system takes full control of communication, task execution, memory allocation, etc. The main code of scheduler is shown 2.1. Therefore, it is possible to observe different behaviors of applications and trace the performance data. By inserting the tracing code in the runtime system, we are able to gather the information we need. The following events are important to our performance analysis. The related code is inserted in the runtime system.

Begin Idle, End Idle: when there is no available task to be scheduled, the system

begins idle. The schedule keeps polling the task queue. When a task becomes ready, the idle time ends. The total idle time for this period is marked as the time from the time of begin idle event. The higher the idle time is, the worse the overall performance is. One goal of performance optimization is to minimize the total idle time.

Begin Execution, end execution : When a message is picked up for execution, a Begin Execution event is recorded. The related information includes the time, the function name, the associated object id, etc. Besides the computation, the data in the message is also recorded including data bytes, sender. When the task associated with the message is finished, an End Execution event is recorded. The time between these two event is the time of task execution.

Overhead : This includes anytime that is neither idle time nor execution time is included in the overhead spending in the runtime. The overhead in the runtime system includes the communication overhead, the queuing overhead, scheduling overhead, all the software related overhead. The goal of performance optimization is to minimize the overhead.

Memory allocation : All the message allocation is done by calling internal CmiAlloc. This enable memory tracing to record how much memory is used.

The above are the most important events we care. These events are common to any tracing framework. The actions associated with the events can be overloaded in the new defined modules. The next chapter describes how these events are used to collect the performance data.

2.4 Applications and Benchmarks

In this thesis, a few real world applications are studied as examples. Their background is discussed here for references.

NAMD: NAMD [15] is a molecular dynamics application that was developed in the mid-1990's. The parallel structure of NAMD is based on a unique object-based hybrid decomposition, parallelized using the CHARM++ programming model. Figure 2.2 illustrates the decomposition in NAMD. Atomic data is decomposed into spatial domains (called "patches") based on the short-range interaction cutoff distance such that in each dimension

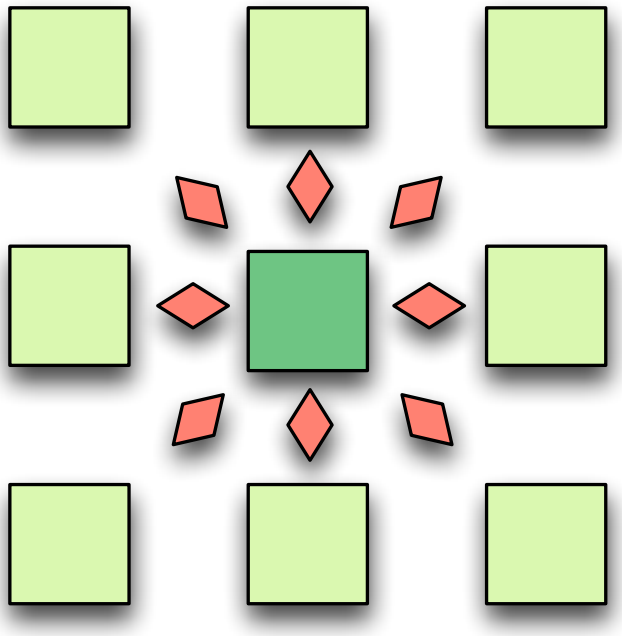


Figure 2.2: NAMD decomposition

only atoms in one-away or, when necessary to increase concurrency, one-away and two-away neighboring domains will interact directly. These equally-sized domains are distributed as evenly as possible across the machine and are responsible for accumulating forces and integrating the equations of motion asynchronously via per-domain user-level threads. Patches are represented on other cores by proxies and all position and force data sent between cores passes via these proxy patches.

The parallel structure of NAMD is shown in Figure 2.2. The calculation of short-range interactions is orthogonally decomposed into "compute objects" representing interactions between atoms within a single domain, between pairs of domains, or for groups of neighboring domains for terms representing multi-body covalent bonds. Compute objects are scheduled by local prioritized Charm++ messages when updated position data is received for all required patches. Longer-running domains are further subdivided by partitioning their outer interaction loops to achieve a grain-size that enables both load balancing and interleaving of high-priority PME or remote-atom work with lower-priority work that does not require off-node communication.

The long-range interaction in NAMD is implemented via the Fast Fourier Transform

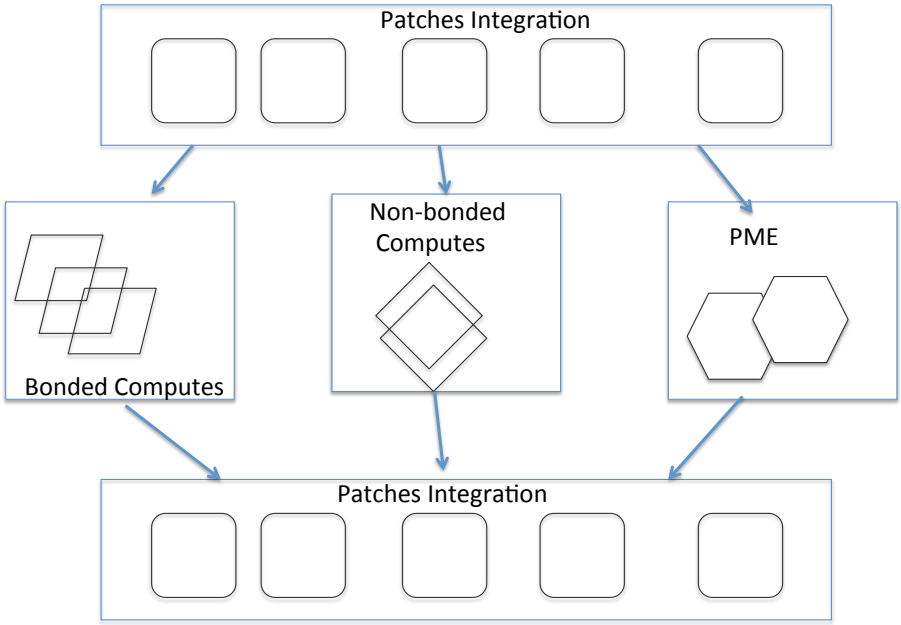


Figure 2.3: NAMD parallel structure

(FFT) based particle-mesh Ewald method (PME). PME calculation, due to the data transposes required in three dimensional Fourier transforms, is highly communication intensive [16], and therefore very challenging to scale. NAMD supports both slab (one dimensional decomposition) and pencil (two dimensional decomposition) PME.

ChaNGa : Simulation is widely applied in astronomy to and the calculation of the gravity among particles is the key to the simulation. N-body simulation is the most popular techniques for it. Nowadays, tree codes for gravity are popular following the publication of the Barnes and Hut paper [17]. This method soon includes gas dynamics using the Smooth Particle Hydrodynamics (SPH) method. As well as allowing for any level of adaptation, the tree method is not restricted to a particular geometry. This makes it particularly useful for isolated galaxies and clusters of galaxies. Tree codes have been extended to include periodic boundary conditions, perhaps most successfully in hybrid tree/particle- mesh methods where the long-range gravity is solved using an FFT while the tree is used for short-range forces. The irregular nature of the tree-walk is a significant challenge for parallel implementations. The N-body/Smooth Particle Hydrodynamics (SPH) code ChaNGa implemented in Charm++ has been developed to address these challenges Charm++ [18–20]. The

program composes of four phases, domain decomposition, tree build, tree traversal and gravity.

In decomposition phase, the particles are organized in space filling curve. The curve is split into TreePieces, which are contiguous sets of leaves (or buckets) containing particles. The ancestor nodes of those leaves up to the root of the entire tree. Once the particles are on the TreePieces, each TreePiece builds its tree independently in a top-down manner. The building starts from the root node that contains the entire simulation domain. During this phase, each TreePiece has the information about the extents of the domains of all other TreePiece to determine which nodes are boundary nodes. The traversal of the tree to calculate gravity is done in third phase. During the tree traversal the local and remote particles are accessed. While the particles are collected the gravity calculation is performed ,which costs most computation in the execution.

State space search problems : The state space search technique has varied applications. The traveling salesman problem and various scheduling problems are commonly encountered in the field of operations research and artificial intelligence. Other examples include floor-plan design in VLSI, genetic search for optimization, and game-playing programs such as chess solvers. Given that no polynomial-time algorithms are known to exist for these problems, they are solved through a systematic exploration of all possible configurations of their inherent elements. Each such configuration is termed a *state*, and the set of all possible configurations is called a *state space*. Generally, an operator is available to transform one state into another through the modification of the former's configuration. The objective of the state space search problem is to find a path from a *start* state to a *goal* state (or to each among a set of goal states). Most often, the search is seen to be tree-based – at each step of the search, we transform a stored *parent* state into several *children* states that do not violate the constraints specified by the problem. In this sense, we also refer to states as *nodes* of a search tree.

Classical examples include N-Queens problem, travel salesman problem, game problem, etc. This thesis focuses on N-Queens and UTS problems. *N-Queens* is a backtracking search problem in which N queens must be placed on an $N \times N$ chess board so that

they do not attack each other. We search for all solutions to the N-Queens problem, for various values of N. The *unbalanced tree search (UTS)* [21] problem is to count the number of nodes in an implicitly constructed tree. The tree is parameterized in shape, depth, size, and imbalance. Each node has all necessary information to construct its children. Therefore, starting from the root, the tree can be traversed in parallel in any order as long as each parent is visited before its children. The imbalance of a tree is measured by the variation in the size of its subtrees. Highly unbalanced trees pose significant challenges for parallel traversal because the work required for different subtrees may vary a lot. As a result, an effective and efficient dynamic load balancing strategy is required to achieve good performance. This benchmark stresses the ability of parallel runtime in terms of grain size control and load balancing.

2.5 Experimental Machines

The machine models where the experiments are performed in this thesis are supercomputers with large number of nodes, low network latency and high network bandwidth. Each node is composed with multiple sockets of processor cores. The nodes are connected in some topological way. Particularly, we have used two systems to perform our experiments - IBM Blue Gene/Q system and Cray XE6/Xk6 system. IBM Blue Gene/Q system uses low power embedded Power PC cores to scale to the 100 PF configuration. Each BG/Q node has 18 Power ISA A2 64-bit embedded PowerPC cores running at 1.6 GHz. One core is dedicated for operating system processing and one core is a spare core, leaving 16 cores for application processing. Each core has four hardware threads that have their own register files but share other resources such as the L1 and L2 caches, compute units and load/store resources. The A2 core can execute two concurrent instructions per cycle, one fixed and one floating point, but each thread can issue only one instruction per cycle. So, to fully saturate the core's resources, at least two threads per core must be used. Blue Gene/Q has a data network [22] with a 5D torus topology, where each link is capable of simultaneously sending and receiving at 2GB/s. Due to packet header overheads the maximum achievable throughput is 1.8GB/s. The 5D torus results in lower latency to furthest nodes and higher bisection throughput as

compared with a 3D torus on BG/L and BG/P. On BG/Q the point-to-point network, the collective and barrier networks all share the same torus network. The particular machine we used is *Vesta* at Argonne National Laboratory. The PAMI (Parallel Active Messaging Interface) machine layer in Charm++ [23] was used on IBM Blue Gene/Q system.

The Cray XE/XK system is connected using the Gemini Interconnect with latency as low as to $1\,\mu s$ and bandwidth as high as $8\,GBytes/s$. It provides hardware support for one-sided communication. One Gemini ASIC serves two nodes by connecting each node to one network interface controller (NIC) over a non-coherent HyperTransport(TM) 3 interface. The NIC provides two hardware components for network communication: the *Fast Memory Access* (FMA) unit and the *Block Transfer Engine* (BTE) unit. It is important for developers to properly utilize both of them to achieve maximum communication performance. Each XE6 node contains two sockets of sixteen 2.2GHz AMD Bulldozer processor cores while each XK6 node contains one socket of sixteen processor cores and one GPU. UGNI (user Generic Network Interface) [24, 25] was used on these systems. The particular system our experiments are performed on is *JYC* at National Center for Supercomputing Applications.

Chapter 3

Framework of Performance-analysis-based Introspective Control System

This chapter describes the main components in the PICS framework. The overview of PICS framework is firstly discussed. The performance metrics to evaluate the performance are then discussed. The core components of automatic performance analysis and tuning process are presented at the end.

3.1 Control System Framework

By definition, "Control theory is a theory that deals with influencing the behavior of dynamical systems" [26]. It is an interdisciplinary subfield of science, which originated in engineering and mathematics, and evolved into use by the social sciences, such as psychology, sociology, criminology and in the financial system." [26] Typically "control systems may be thought of as having four functions: Measure, Compare, Compute, and Correct. These four functions are completed by five elements: Detector, Transducer, Transmitter, Controller, and Final Control Element. The measuring function is completed by the detector, transducer and transmitter. In practical applications these three elements are typically contained in one unit. A standard example of a measuring unit is a resistance thermometer. The compare and compute functions are completed within the controller, which may be im-

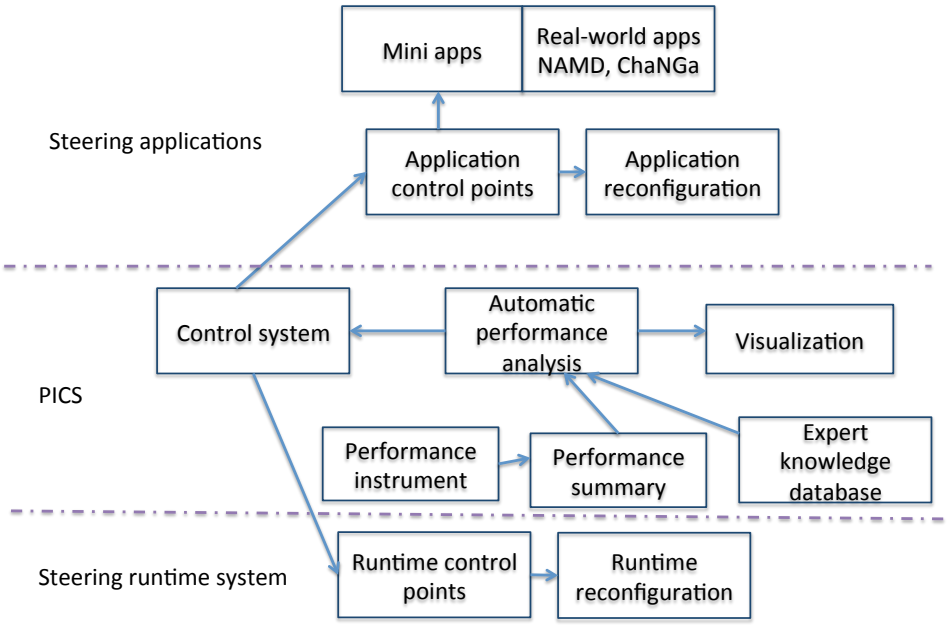


Figure 3.1: Framework of PICS

plemented electronically by proportional control, a PI controller, PID controller, bistable, hysteric control or programmable logic controller. Older controller units have been mechanical, as in a centrifugal governor or a carburetor. The correct function is completed with a final control element. The final control element changes an input or output in the control system that affects the manipulated or controlled variable". For computer applications to be controlled, two related notions are observability and controllability. Both of these features are essential. Only when its behavior is observable, related performance data can be instrumented and analyzed. In order to instrument, measurable metrics need to be proposed. In our system, we categorize these metrics into three types based on their relationship with the ultimate objective functions. After the performance is observed, the control system should have a scheme to propose better values for the tunable parameters. Whether the tuned values can be used to steer applications finally depends on the controllability of the applications. That means the application should be able to reconfigure or adapt to the new values while the correctness is guaranteed.

Similar with the typical control system, our PICS contains the components of measuring, comparing, computing and correcting. Figure 3.1 shows its infrastructure. The control

system steers both applications shown on the top of the figure and the runtime system at the bottom of the figure. Both applications and runtime can register tunable parameters to the control system using the same mechanics. The tunable parameters are encapsulated by control points, which will be described later in detail. Once the configuration of control points are adjusted, the applications and runtime system need to adapt to the new configurations. The difference between control points in an application and the runtime system is that control points in the runtime system are registered by the system developers while the application control points are registered by the application developers. Moreover, the runtime system control points affect all the applications running on it without application modification. The application control points only affect the specific application.

The core components of PICS are shown in the middle of the figure. PICS records configurations of the application and the runtime system, monitors application behaviors, and collects the application performance data. Meanwhile, the control system has a set of expert knowledge rules, which define various application characteristics and corresponding solutions to solve a particular performance bottleneck. Utilizing both the performance data and expert knowledge rules in our system, the system performs online automatic analysis to detect performance problems to determine the control points that need tuning and the mechanism to adjust them. New configurations are fed back to the application or runtime system to adapt to the new values.

The goal of the control system is to find the values for the tunable parameters to achieve better performance metrics. The performance metrics are a function of the application control points, the runtime system control points and the hardware control points. For some applications this function can be obtained using modeling techniques. For others, it might be impossible to derive the function. Other heuristic techniques will be used to serve the purpose of steering.

In order to steer applications or the runtime system, the first question we need to discuss is to define the performance metrics.

3.2 Performance Metrics

Performance metrics are used to evaluate how the program runs. We have extracted two categories of performance metrics, depending on whether it is used for users or for the system, ultimate metrics and intermediate metrics. The metrics that developers require to optimize are defined as as the **ultimate performance metrics**. This type of metric is directly from application developers. The most traditional and important metric is the **time cost to complete execution**, The goal of most applications is to run as fast as possible without considering other cost. This metric is also the fundamental reason why parallel computing is used. With the energy cost becoming a problem, another important metric is the **energy efficiency**. Due to some limit, it might be desirable to lower the overall supercomputer power limit. In this case, the execution time is not the only metric to judge. It is desirable to consider the overall performance. Another important metric is **network communication volume**. On many supercomputers, network resources are shared among concurrently running jobs, so it is inevitable to have interference among jobs. Therefore, in some cases in order to maximize the overall system performance, it might require to constrain the network communication volume of each application. Even it might hurt the execution time of one particular application. In practice, one or multiple performance metrics may need to be satisfied. In this thesis, we focus on the first metric, which is the time cost to complete execution. Meanwhile, we take the energy cost and network communication volume as secondary metric if application developers require.

The above metrics are the final objectives that application developers want to achieve. However, these metrics provide few hints about how to steer the applications. Therefore, we propose better intermediate metrics based on our study of parallel applications. Intermediate metrics are the ones that can be measured during program run. They are used to evaluate the effectiveness of the performance steering. We have concluded the following intermediate metrics.

- **Time cost to complete one step**: this is commonly used in iterative scientific applications. Less time cost to complete one step shows the effectiveness of the per-

formance steering.

- **Utilization percentage**: the time that CPU is busy. This can be used for any application. In general, high utilization percentage stands for good performance.
- **Useful work percentage**: For some applications, even when CPU is busy, it is doing some runtime or parallel overhead work instead of useful work. What contributes to the real progress of the applications is the useful work. The more percentage the useful work accounts for, the better the application runs. This metrics requires the developers to mark the useful work function.
- **Idle percentage**: the time that CPU is idle. Higher value hints worse performance.
- **Overhead of the applications**: this is the total execution time subtracting the idle time.
- **User defined metrics** : For some applications, users can define their own intermediate metrics. For example, for graph or tree search, the number of nodes traversed in a time period is often used as the metrics to evaluate the parallel performance..

3.3 Control Points

In our control system, we define the tunable parameters as control points. Different from the general variable in optimization models, the control points have a few features. First, the values of the control points should be adjustable. The applications or the runtime system should be written in a way to use different values of control points. Secondly, the control points have some **effects** on the performance. That means when we change the values of control points, the performance will be affected in that aspect. We have concluded that the effects of control points on the applications include:

1. Degree of parallelism: this is affected by any parameters which relate to the number of parallel tasks. For example, for a fixed problem size of Jacobi program, the number of sub-blocks control the degree of parallelism. In molecular dynamics, the number

of patches affects the parallelism. In pencil 3DFFT, the number of pencils control parallelism.

2. Grain size: this controls the amount of computation per parallel task. This metric is converse with the degree of parallelism.

3. Memory consumption: this is affected by any parameters that control memory consumption.

4. Number of messages and message size: proper message size can both better utilize network bandwidth and also overlap computation with communication. When the performance problem is related to message size, corresponding parameters affecting it should be chosen and tuned.

5. Load balance frequency: the time period of performing load balancing. The more frequent it is performed, the more balanced the load is on all processors but the higher the overhead is.

The effect of control points are the most important property. It links the results of performance analysis with the performance steering process. Details are discussed in the next chapter.

3.4 Automatic Performance Analysis

The most general way to choose optimal configurations among various ones is to perform an exhaustive search. Although it guarantees to find the optimum values, it is typically infeasible for real applications due to the huge search space and the complexity of the searching.

One novelty of the PICS is that it is built on automatic performance analysis. This runtime system takes control of the application scheduling and communication. Therefore, it is easy to instrument, record and track application behaviors. Based on the instrumentation data, performance analysis can be performed. When possible performance problems are detected, we relate the performance problems to the effects of the control points. Therefore,

we only tune the control points which affect the performance while we ignore the others. This significantly reduces the number of control points we need to optimize. The other advantage is that based on the effect of control points and performance problems, the steer direction is guided instead of proceeding randomly. This thesis identifies various common application characteristics. Then it proposes a decision tree based automatic performance analysis to to detect performance deficiency. The results of analysis are fed back to the controller to guide performance steering. Details are discussed in the next chapter.

3.5 Improving Performance by Reconfiguring Runtime and Applications

The essential requirement for our methodology to work is that applications and the runtime system should be reconfigurable during execution. In order to reconfigure applications, some efforts are needed from the applications developers. First, the applications need to provide control points to PICS to tell what are tunable. These control points are registered in the runtime system and then their effects on application performance are observed, stored, analyzed. Secondly, when the configurations of application change, applications or runtime systems need to correspondingly adapt to the new configurations. This might be easy or hard. Some applications may require no change to use new configurations. However, some might need significant changes, like global data redistribution. The details are discussed in section 5.5.

3.6 The Tuning Process

Having discussed the components of PICS, figure 3.2 shows the process of applying PICS to steer applications. At the beginning of executions, control points are registered. The program runs with default or user provided configurations of control points. During program execution, the program is instrumented. A performance database is maintained to keep track of the history performance metrics. When new performance result is collected, it is compared with history result to tell whether the tuning improves or hurts performance.

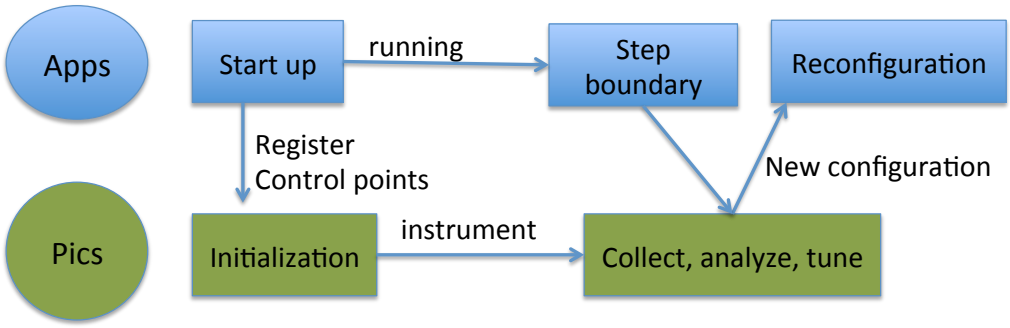


Figure 3.2: The process of PICS tuning

If the performance for current step is better than ever, the configuration for this step is recorded to database for future use. Performance analysis is conducted based on the performance summary to derive the possible reasons for deficiency. Based on the results of analysis, corresponding solutions are suggested by the expert knowledge system. Since these solutions are also related to the effects of control points, the corresponding control points are tuned. The tuning process is based on the ratio of current performance to previous performance and effect of control points. When the values are tuned, they are broadcast to all other processors to update their local configurations. In this way, the future run will use new local values instead of querying the remote processors every time.

Chapter 4

Decision Tree Based Automatic Performance Analysis

The goal of the control system is to find the optimal configuration of all the control points. Due to the complexity of the runtime system and application, many control points will be registered with PICS. This leads to a huge search space of configurations. As a result, performing direct optimization (such as hill climbing [27], genetic algorithm (GA) [28]) can be time consuming. When we examine control points closely, we notice that some control points may have more impact than others. If we can determine which control points have the most impact on the overall performance, the process may be accelerated.

The approach we take is to perform automatic and comprehensive analysis to detect a performance deficiency. Since the runtime system takes control of the application with regard to scheduling and communication, it is easy to instrument, record, and track application behaviors. Based on the instrumentation data, performance analysis can be performed. When possible performance deficiencies are detected, we can steer the control points whose effects are related to these performance deficiencies instead of searching all possible configurations. This significantly reduces the search space. The other advantage is that based on the effect of control points and performance problems, the direction of performance steering is guided instead of proceeding blindly.

This chapter first describes the scheme to collect performance data and gather the sum-

mary. Based on the summary data, the decision tree based performance analysis is exploited. After the analysis is performed, the results are fed into the control system in the close loop. In addition, the results can be visualized to help developers understand and manually tune their applications. The techniques for visualizing the results are exploited at the end.

4.1 Gathering Performance Data

The automatic performance analysis composes of three phases, tracing, summarizing and gathering.

Performance data is traced during program run on each processor. Our control system is based on the CHARM++ runtime system. The CHARM++ runtime system takes full control of the program execution. It controls the mapping of the tasks to physical resources, memory management, scheduling of function executions, and communication. Therefore, it is easy to add instrumentation in the runtime system to monitor the behaviors of the programs. Currently, the runtime system can record the events of *begin idle, end idle, begin of function execution, end of function execution, message creation, message sending, message receiving, assignment of tasks* . Instrumentation of these events are designed as a linked module. It requires no source code change from the applications.

Based on these events, our control system summarizes and gathers performance summary as needed. To get the summary, the traced performance data is accumulated together and pre-processed. Currently, we summarize the following performance data. For each of the performance data, we get the maximum, minimum and the average.

High level performance data : idle percentage, overhead percentage, utilization percentage

Function execution data : function execution duration, number of functions executed

Object data : load per object, number of object per processor, communication per object

Communication data : send/receive number of messages, bytes of data, external bytes

In phase 3, upon some boundary of applications, the performance data on each processor is gathered to a central processor or a group of processors. A reduction algorithm is used to gather all the performance data. Once the global performance data is collected, performance analysis is performed. During the gather process, one important question is to determine when to perform analysis for various type of applications. The principles to determine the time to perform analysis are as following: The first is to maximize the quality of performance analysis. When the data is collected and analyzed, it is better to have associated application boundary with it. Therefore, the profile data for a short period should capture the most important features and the pattern of the whole application. Secondly, it should have minimum interference and overhead on the application when performing analysis. For most applications, performing analysis during idle time is a good choice to reduce the overhead. An even better way is to use a helper thread to do it. Thirdly, it should be easier for application developers to modify their application. Finally, after performance data is collected, it should be easy to evaluate the performance so as to guide the runtime system for steering.

Based on the above principles, the following three scenarios are summarized.

1. Synchronization point to perform analysis: application developers insert code at a global synchronization point. Although it is hard for the runtime system to recognize the time-step of applications, it is obvious for developers to tell it and insert code. In this case, the root processor requests all processors to reduce their performance data.

2. In some iterative applications, there is no explicit global synchronization point. Each processor knows its own time step but does not reach global barrier for each step. In this case, application developers can mark time step locally instead of globally.

3. For non-iterative applications, it is difficult for developers to mark the time step. State search applications belong to this category. In this case, the runtime system will decide when to performance analysis. It can be performed either periodically or based on performance results. One possibility is to perform analysis when some processor has low utilization. This processor will trigger data collection and analysis.

Now the performance summary data is gathered on one processor, which can carry out the analysis.

4.2 Decision Tree Based Automatic Performance Analysis

In order to determine the application performance problems and possible solutions, we need to identify the characteristics of the program. Based on our study of various applications and the runtime system, we classify the program characteristics in to three main categories, which are problem decomposition, task mapping and scheduling.

Problem decomposition is about how to decompose the whole problem into small problems, which can be solved in parallel. This is mainly the responsibility of application developers. Problem decomposition directly determines the grain size of computation and communication, the degree of parallelism. Good problem decomposition is an essential factor to achieve high performance.

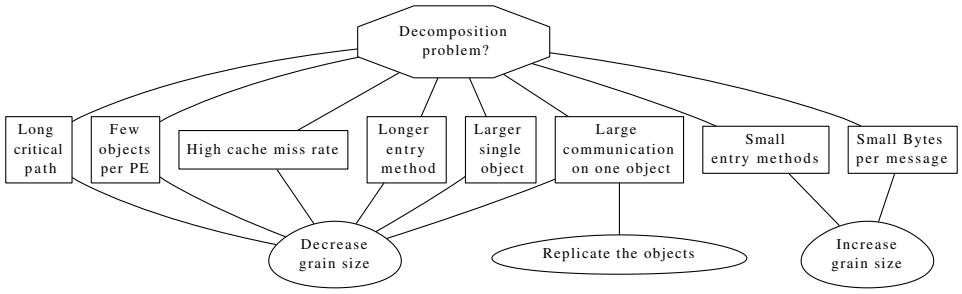
The characteristics about problem decomposition include:

1. A single entry method takes too much time.
2. A single object takes too much time. Because in our model one object can be only executed on one processor at one time, this causes an execution dependency.
3. There is too much runtime system overhead when average grain size is too tiny.
4. There is not enough parallelism to overlap communication with computation. When these four characteristics are identified, it hints a problem of grain size.

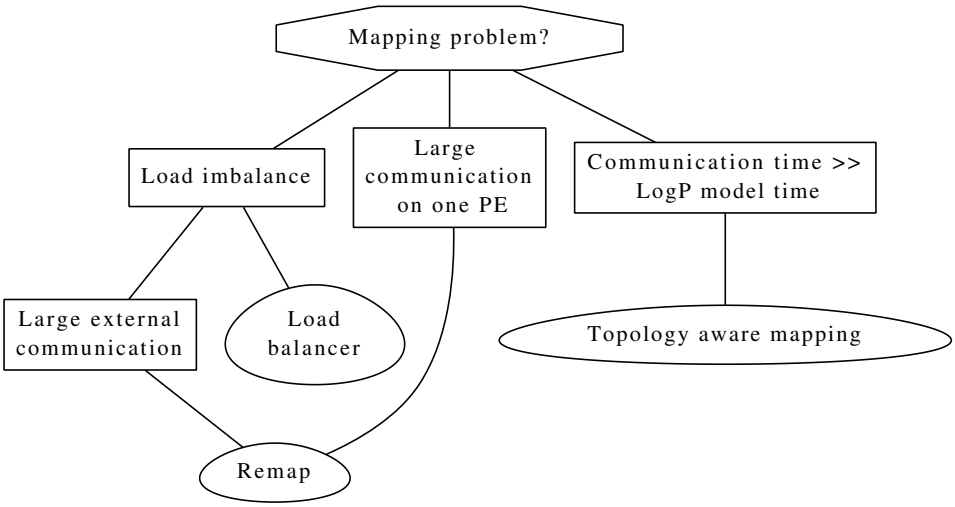
The category of the problems is represented in Figure 4.1(a).

Task mapping is about how to map tasks to physical processors. Task mapping affects the communication cost. Related work includes communication topology aware mapping [29–31]. It also affects the load balance. Besides these two most important effects, it might also affect memory usage and I/O usage.

When mapping is a problem, a possible result is high idle time. The specific characteristics shown during program run can be summarized: (1) There is load imbalance among all



(a) Decomposition characteristics



(b) Mapping characteristics

Figure 4.1: Decomposition and mapping characteristics

processors. This can be easily identified by calculating the ratio of maximum load to average load. Corresponding solution is to perform load balancing. (2) Even when the load is balanced, a bad mapping can still lead to high communication in the system. Whether communication is high or not can be evaluated by the metric of the number of hops multiplying bytes. Both maximum hops and average hops may impact the performance. The corresponding solution is to perform a topology-aware and communication-aware load balance. This is illustrated in Figure 4.1(b).

Scheduling is about the order to execute the available tasks on processors. How the scheduling performs directly determines the application performance.

When scheduling is a problem, the direct result is the high idle percentage. The main characteristic is that critical tasks are delayed. Therefore, processors that depend on the critical task become idle. The other potential problem caused by scheduling is out of memory. If only tasks which consume memory are scheduled while the tasks free the memory are not, it will consume all memory.

Besides the above three main categories of program characteristics, the other characteristics which relate to performance can be collective operations, like how the broadcast, reduction algorithms are implemented, How the runtime system deals with the intra-node and inter-node communication. These also greatly affect overall performance.

Combining all the program characteristics and corresponding solutions discussed above, the full decision tree is represented in Figure 4.2. In this figure, starting from the performance summary data, the decisions are made based on the performance characteristic and the specific run performance data according to the tree. The triple-octagon shapes represent the most top level of performance metrics. Based on it, there are three branches. Under each branch, we check whether the corresponding performance characteristic exists. The performance characteristic is represented in boxes. When a characteristic is satisfied, the corresponding performance solution is proposed at the leaves of the tree, represented by eggs. The corresponding performance solution has two meanings. The first is which aspect of the applications should be steered. The another is the direction of the steering. For example, if the solution is to decrease the grain size, we need to tune all the parameters

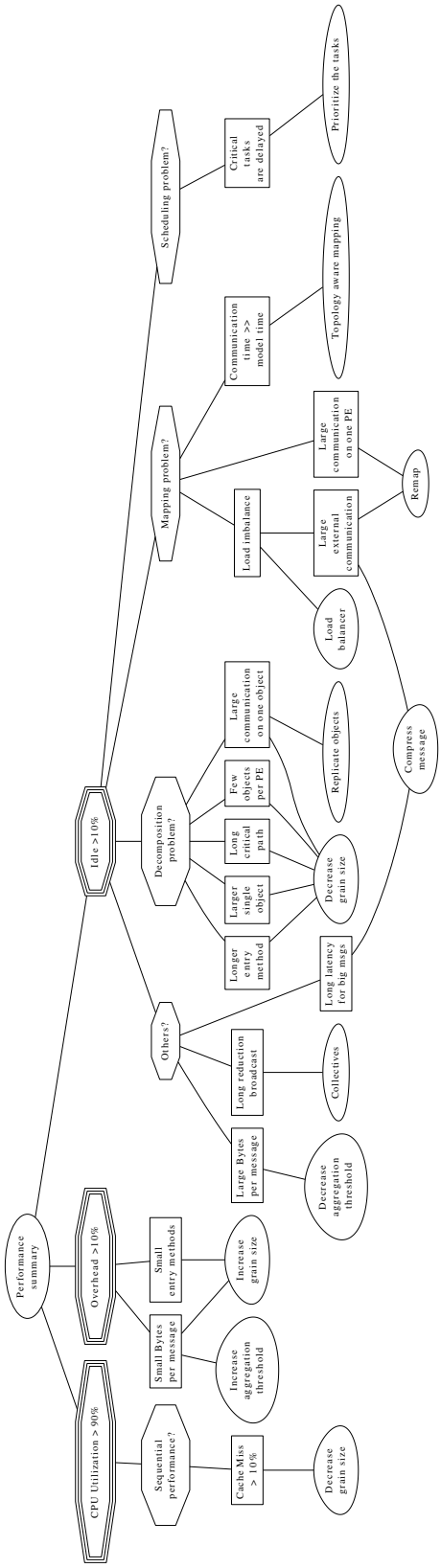
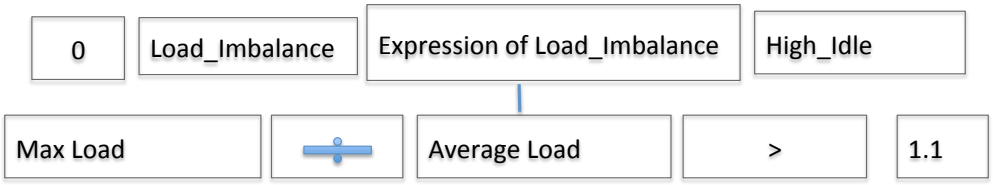


Figure 4.2: Performance analysis decision tree

(a) Condition node



(b) Example of load imbalance

Figure 4.3: Definition and example of condition node

which have the effects of grain size. The direction is to adjust the values so that the grain size becomes smaller.

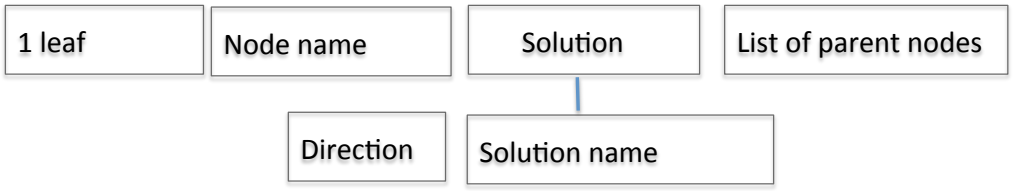
The process of performance analysis is traversing the tree. Whenever a leaf node satisfies, it is saved for tuning. As a result, we have a list of performance solutions. We feed these solutions into the control points database so as to figure out what control points to tune and in which direction.

Defining Rules for Decision Tree

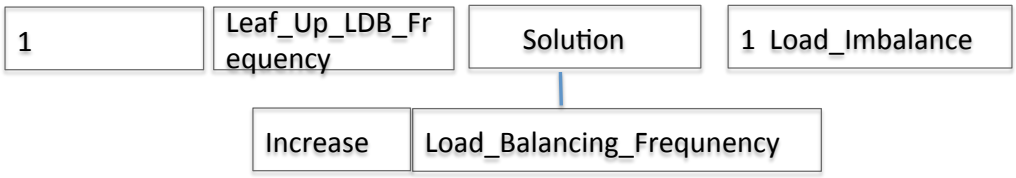
In order to make our analysis scalable and easy to expand, we have defined a set of rules in the configuration files. These files are read during the startup of the program to construct the decision tree. The configuration files are plain text files and easy to read and modify. Therefore, rules can be easily added or modified.

The configuration file contains multiple lines of information. Each line represents one or more tree nodes. There are two types of nodes, condition nodes and solutions nodes. Condition node represents a performance characteristic while solution node represents the solution to solve the parent problem. Both nodes have the following formate.

- The definition of condition node is shown in Figure 4.3. $NodeType$ is 1, which represents an internal node of tree. $NodeName$ is the identifier to recognize the node. It is unique for each internal node. The purpose is for child to refer to it. Following it



(a) Solution node



(b) Example of increasing load balancing frequency

Figure 4.4: Definition and example of solution node

is the expression of performance characteristic. At the end is a list of parent nodes. The expression of performance characteristic contains a few fields. *Performance data 1* tells which performance data this internal node corresponds to. *Performance data 2* tells the base of performance data, which can be a constant number defined by the system or a input source. *Operator* operates on the two items of performance data, which can be *add*, *subtract*, *multiple*, etc. *Compare operator* compares the result of the above with the *threshold value* defined by the system.

- When $NodeType$ is 0, this is a leaf node of tree, which represents a solution to solve the performance problem shown in the parent node. The format of solution node is shown in Figure 4.4. Comparing with internal node, leaf node as a solution is much easier. Following the $NodeName$ is the solution to apply, which contains the *direction* and the *name*. Direction tells whether to increase or decrease specific effect. The name is for the solution, which is also the effect name of control points. *Parents* link this solution to its performance problem nodes. Figure 4.4(b) shows the solution of increasing load balancing frequency to solve the load imbalance problem.

Building the Decision Trees

Corresponding to the configuration files, there are two types of nodes we need to build in the tree. The internal nodes represent the constraints, which are also the performance

bottlenecks. The leaf nodes are the solutions to solve the problems. The building process is straightforward. Each line is read and one node is constructed and added into the decision tree. Depending on the type of the nodes, either an internal condition node is added or a solution node is created as the leaf. Notice that in configuration files, we have various names for the performance data. However, in order to save the space used by the tree, we do not have to store the string of the names. Instead, we map the names into integers. The integers are used in the tree file.

Certainty Tree and Fuzzy Tree

For the expert knowledge rules, some rules are certain to be true while some are possible to be true. For example, when the execution time of specific entry method is much longer than the average processor load, it certainly means high idle time. In order to solve it, the entry method should be split to small ones. Decreasing the grain size of that object will certainly solve this problem. These rules are called **certainty rules**. All the certainty rules compose the certain tree.

Besides the certainty rules, there are some possible rules, which can be very likely true but not necessarily true. For example, decreasing the grain size of object will increase the number of total object. It is likely to solve the load imbalance. However, whether this will really solve the problem also depends on the result of the load balancer. It is not necessarily true because if load balancer does not migrate objects the load imbalance still exists. Another example is that increasing the grain size of object might decrease the overhead percentage. However, if the overhead is caused by too frequent load balancing. Increasing grain size will not help much. We call these types of rules **fuzzy rules**. The fuzzy rules compose a fuzzy tree.

During the process of automatic performance analysis based on these decision trees, the certain tree has the higher priority to be searched. The solutions derived from this tree are added to solution set. When this search is done, the fuzzy tree is searched next. If the solution from the fuzzy does not conflict with any from the certain tree, it will be added. Otherwise, it will be ignored.

Performance Analysis by Traversing the Decision Tree

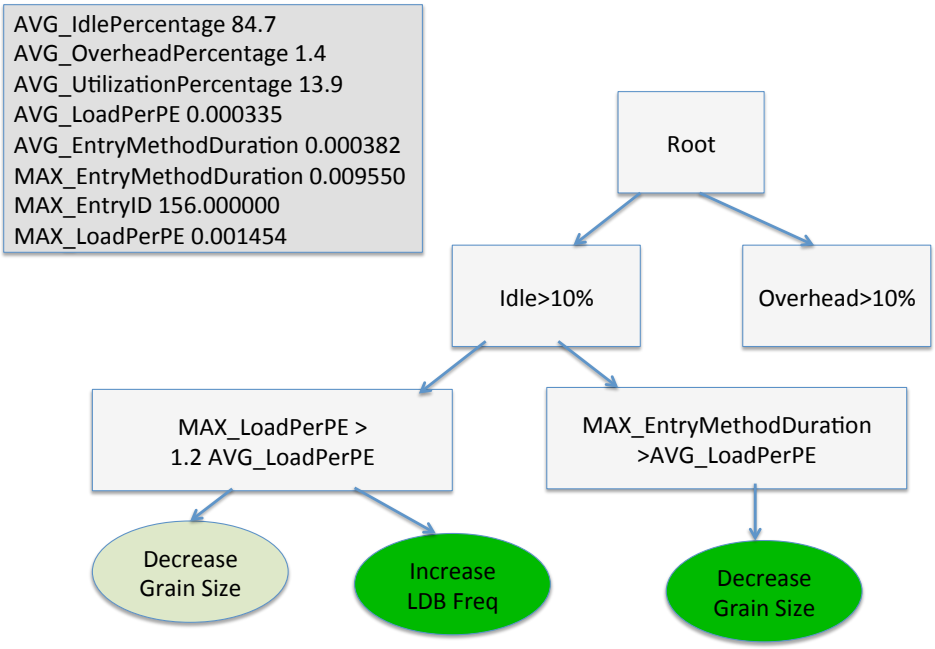


Figure 4.5: Performance analysis tree traversing

During the program run, we collect the performance data as discussed above. Using the data as the input, we traverse the decision tree. Here, we perform a depth-first-search(DFS). Different from the regular DFS, we do not necessarily search all the nodes. Before a node is searched, we use the input performance data and the condition to test whether the condition for the node holds true. If it is true, we continue search its children. Otherwise, we skip this node and all of its children. The algorithm is shown as below. When the traversing is done, a set of solution are found. Each solution corresponds to one effect of some control point. Figure 4.5 shows an example of input and the output of traversing the tree.

Learning for the Decision Tree When the rules in our decision tree are small, it is possible that traversing the tree gives us no results. In this case, we will use exhaustive search by adjusting some control point. We compare the previous and current configuration. Once performance is improved, we also compare the previous and current summary data to check which performance characteristic goes away. We link this performance characteristic with the control point we tuned to form a rule. This rule is added to the decision tree. We call this process **learning**.

Listing 4.1: Traversing the decision tree

```
void DFS( double *input,
       vector<IntDoubleMap>& solutions,
       int level,
       std::vector<Condition*>& problems)
{
  stack.push(root)
  while(!stack.empty()) {
    node = stack.top();
    stack.pop();
    for(each child of node) {
      if(child.isLeaf()) {
        solutions.push(child);
      }
      else if(child.test(input)) {
        stack.push(child);
      }
    }
  }
}
```

4.3 Phase by Phase Analysis

Currently the analysis and tuning is based on steps. However, in most scientific applications each step might contain multiple phases. Each phase has different computation and communication pattern. Therefore, phase by phase analysis should be performed to improve the accuracy of analysis. Meanwhile, some configurations only affect some specific phases not all phases. Therefore, in the phases where some control points do not matter those control points can be filter so as not to search their configuration. One good example of this is that in NAMD non-PME phases, all PME-related control points can be ignored. Figure 4.6 illustrates this point. Every 4 steps, there is one PME step which takes much more time than the other three non-PME steps. If we do not distinguished these steps, the performance for the PME step will be considered to be much worse. Without careful examination, the control point configurations in this step might be processed in a wrong direction. With phase by phase analysis the performance in this step will not be compared with the previous step. Instead, it will be compared with the previous forth step.

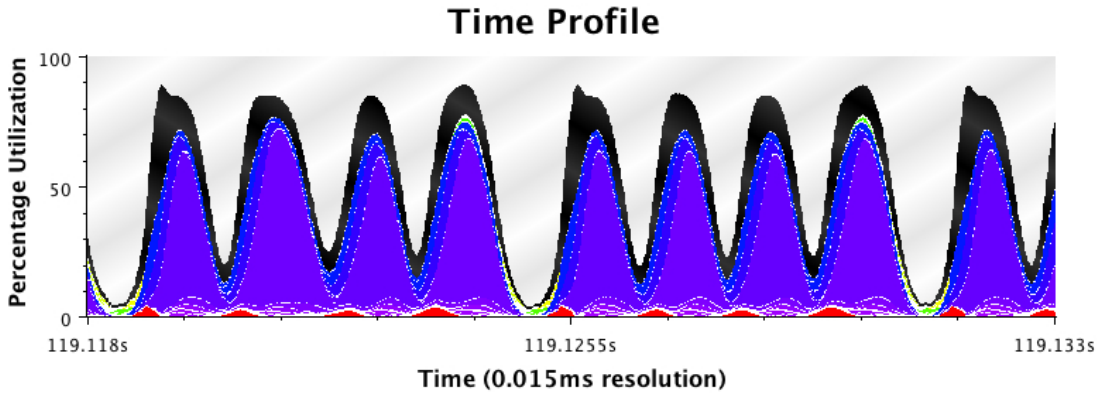


Figure 4.6: Timeprofile of ApoA1 simulation on 1024 nodes

4.4 Display Performance Analysis Results

The results of performance analysis are mainly used to feed into the control to guide the performance steering. Besides this, it can be directly written into plain text files to help the developers figure out the program performance problems.

Projections

Projections [32], is a stand-alone tool to visualize and analyze performance of Charm++ applications. During program execution, performance is traced. The details of events are recorded in plain text files, which are read by Projections. Projections composes a set of tools.

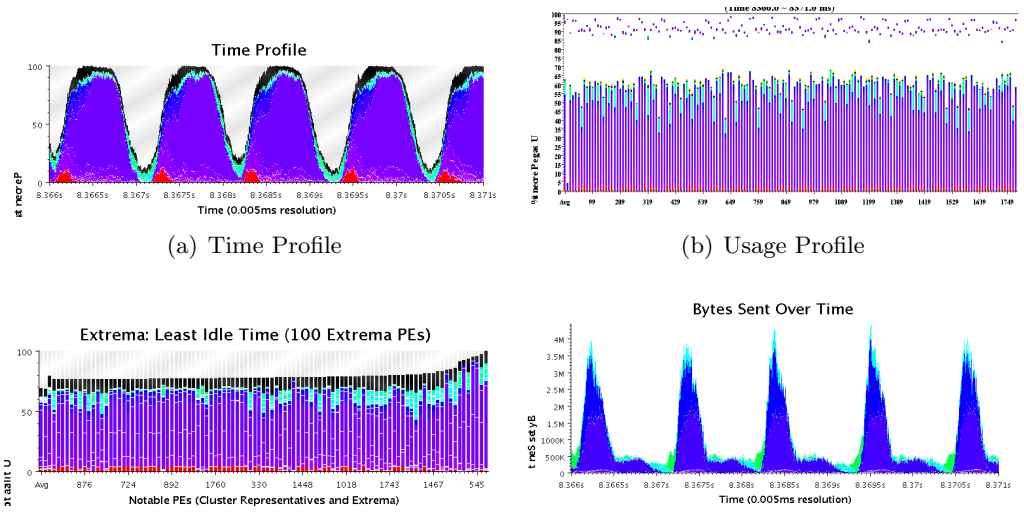
1. *Time Profile* tool : it shows the profile of various functions, idle and overhead time for all processors over the time. The clearly tells how well the program runs.

2. *Usage Profile* tool : it displays each processor's CPU usage for a time period. It is useful to tell how the load is distributed over all processors.

3. *Communication Over Time* tool : it displays the communication over the time.

4. *Extrema Analysis* tool : it sorts the processor based on idle time ascending or descending, utilization descending. This is useful to find processors with interest.

5. *Timeline* tool : it displays every detail of the program execution. Figure 4.8 is an example of using Timeline tool to visualize NAMD performance. It displays a set



(a) Time Profile

(b) Usage Profile

(c) Extrema Profile

(d) Communication Profile

Figure 4.7: Tools in Projections

of processors that users choose. For each processor, it shows the execution of each function, idle period in different colors. The begin time, end time and associated messages to entry methods are also displayed. More features include tracing message back to source, tracing message forward to the destination. Timeline is the ultimate tool to visualize and analyze program behaviors and performance.

All these tools are extremely powerful and useful to analyze the performance. However, there are a few drawbacks of Projections. With the number of processors that the program running on increasing, the data files generated for Projections can be huge, as high as hundreds of Gigabytes or even more. This costs a lot of storage. Also it takes time to download all data from supercomputers to local machines to process. Even worse, Projections becomes very slow to visualize or analyze the huge data due to slow loading of files, files not fitting into the memory. The worst problem is to analyze the complicated data manually. For example, with hundreds of thousands of processors finding the least idle processor, the longest entry method takes long time, which can be tens of minutes for 10K processor data set. It becomes almost impossible to manually figure out the root performance bottleneck from the massive data. This motivates us to use PICS to help performance analysis and visualization.

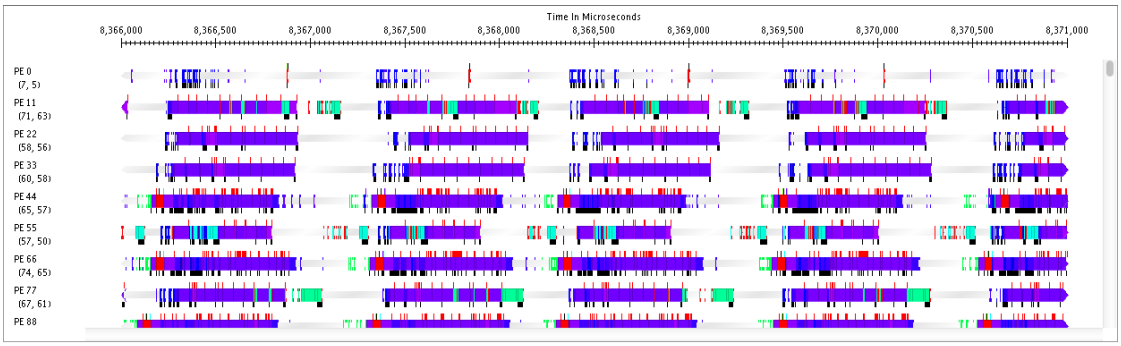


Figure 4.8: Timeline tool of NAMD

PICS and Projections

In order to overcome the drawback of Projections, we have applied PICS to help. The following approaches are used.

Output Processors with Interest : in order to reduce the data, we only output the data for the processors with interest. For example, we output the processors with least idle time, most utilization, most overhead, etc. This significant reduces the amount of data.

Generate a PICS summary file : A PICS summary file is generated to help Projections. For example, instead of finding the least idle time in all processors, this information is read from this PICS summary file.

Output performance bottleneck : The results of performance analysis is written into plain text files to help users understand the performance.

The output of PICS is organized in blocks of data shown in List 4.2. The data contains multiple blocks of *DataEntry*. Each entry is firstly identified by the step id, the begin timer. Following it are the performance summary data for this step, the performance problems and the solutions.

Features in Projections

The PICS output file is read by Projections to reconstruct the data structure for analysis use. In projections in Timeline tool, a few new features are added to display the PICS output. For each feature, a binary search based on the timer is applied to find the proper entry of data. And then the related information is displayed based on the requirement of different features.

Listing 4.2: Struct of PICS output

```
class Node {
  bool isSolution;
  //--- functions
}

class Condition{
}

class Solution{
}

class DataEntry{
  int step;
  int entries;
  long timer;
  double summary[][];
  vector<Node> CondSol;
}

Vector<DataEntry> dataset;
```

Now we have added the following features in Timeline.

Least idle processors The least idle processor is automatically loaded into Timeline.

Highlight the entry methods of longest duration : When there is a performance problem associated with entry method duration, it is important to highlight this entry, the object it belongs, the processor it runs on. The information can all be extracted from PICS output. After it is obtained, the associated entry methods are highlighted for users to clearly see them.

Display the performance problems : All the output from PICS analysis can be displayed in plain text to the users.

Plot the performance summary data : All the performance summary data can be visualized in plot in line, bar, or area formate shown in Figure 4.9.

4.5 Examples of Performance Analysis

In this section, a few examples are discussed to show how PICS can help Projections.

Figure 4.10 is time profile, which shows the CPU utilization is as low as 30% in average. This can be also easily seen from PICS output file shown in table 4.1. In order to figure

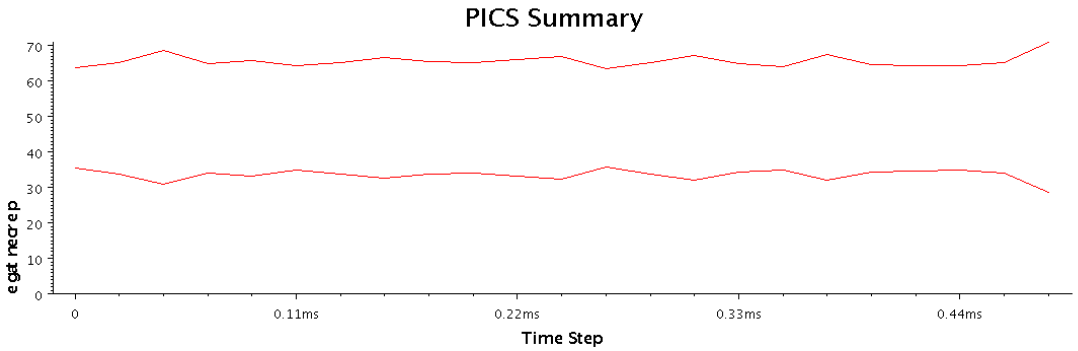


Figure 4.9: Plot of average idle and utilization percentage in Projections

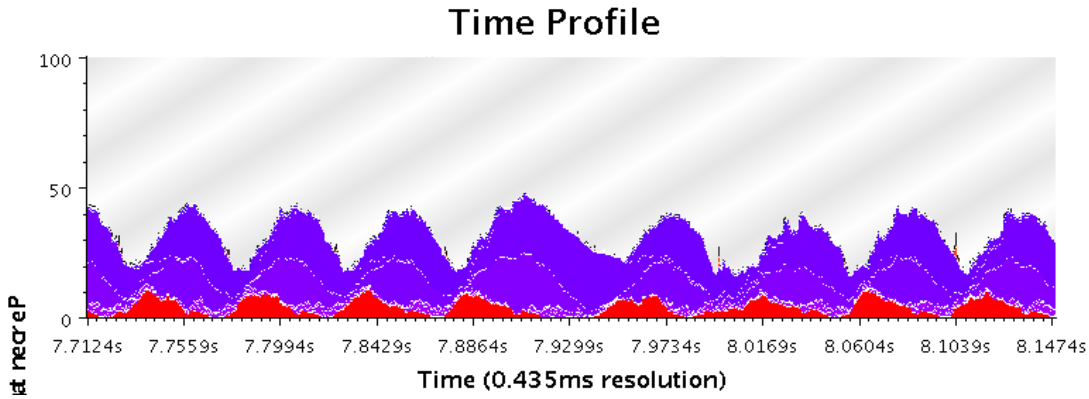


Figure 4.10: Time profile of simulating DHFR using 256 processors

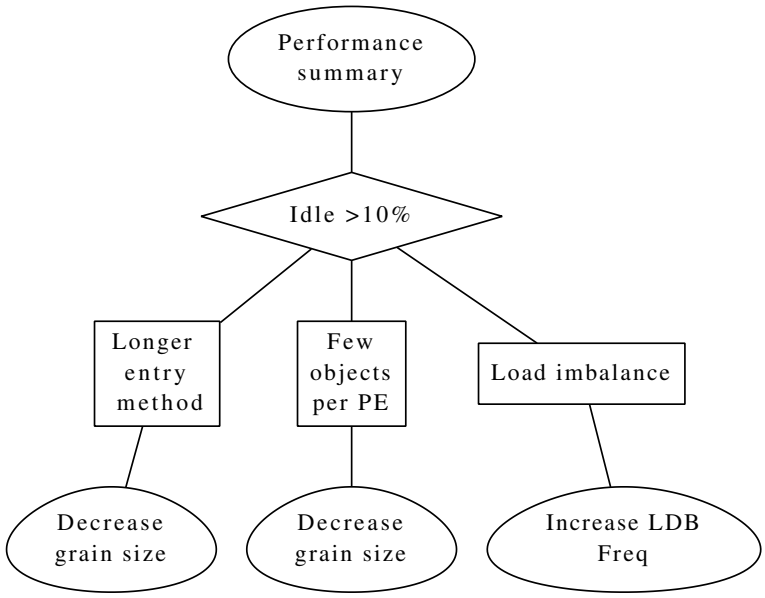


Figure 4.11: Tree nodes visited for NAMD analysis

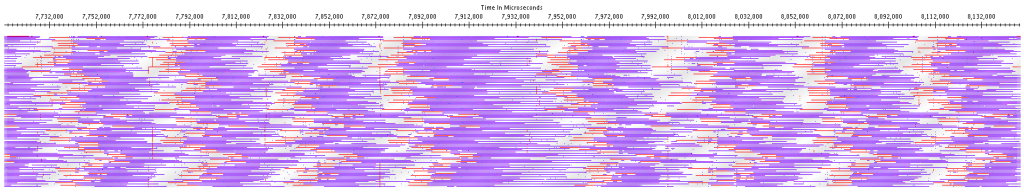


Figure 4.12: Super compact view of time profile of simulating DHFR using 256 cores

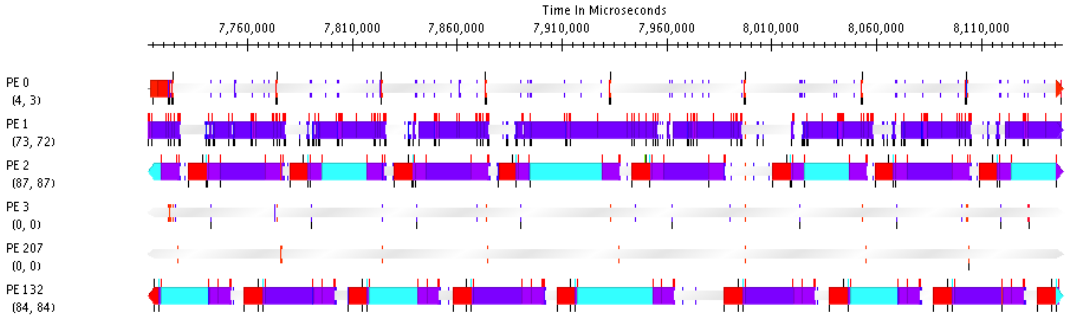
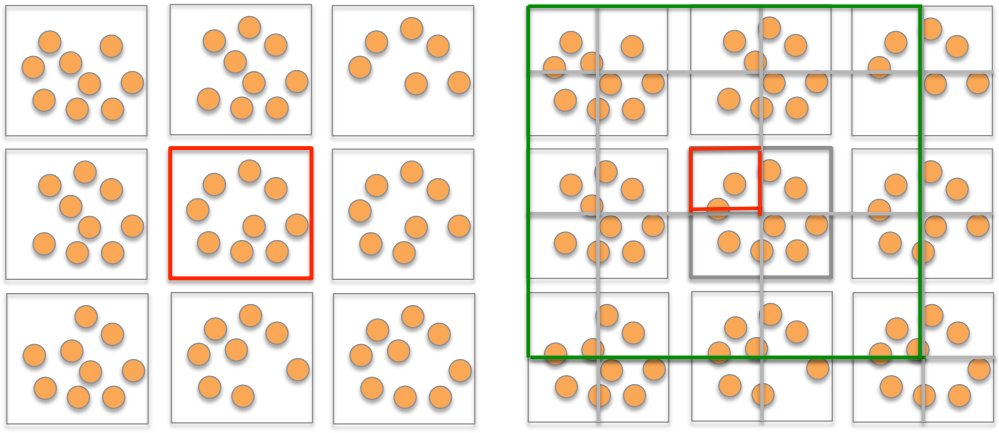


Figure 4.13: Using PICS to load least idle PE, longest entry method

out the problems of causing low utilization, Timeline tool is used. Figure 4.12 shows the compact view of time line of all 256 processors. In high level, we can see that the load on processors is extremely unbalanced. This is one clear reason. Besides, it is hard to tell the other reasons.

Based on the PICS decision tree 4.11, there are a few performance problems. Now we display the PICS output of automatic performance analysis. In text file, the reasons are shown in 4.13. In order to verify it, only some processors' Timeline is loaded. The first 4 processors are loaded. And then based on the result of PICS, the least idle processor, the most idle processor is loaded. It can be easily seen a large variation of processor load. PICS tells the load imbalance is one problem. Besides this, PICS also identifies some entry method takes much longer time than the average load. Based on its results, the related PE with longest entry method is loaded as well as the longest entry method is highlighted. From the figure, it is the 'self short-range calculation' that takes longest time. From our experience, this is due to the decomposition method in NAMD. PICS suggests to increase the Away decomposition. Therefore, we tried another experiment with same input but using 2 Away XY decomposition shown in Figure 4.14(b). In 2Away XY decomposition,



(a) 1Away decomposition (b) 2Away XY decomposition

Figure 4.14: Decomposition in NAMD

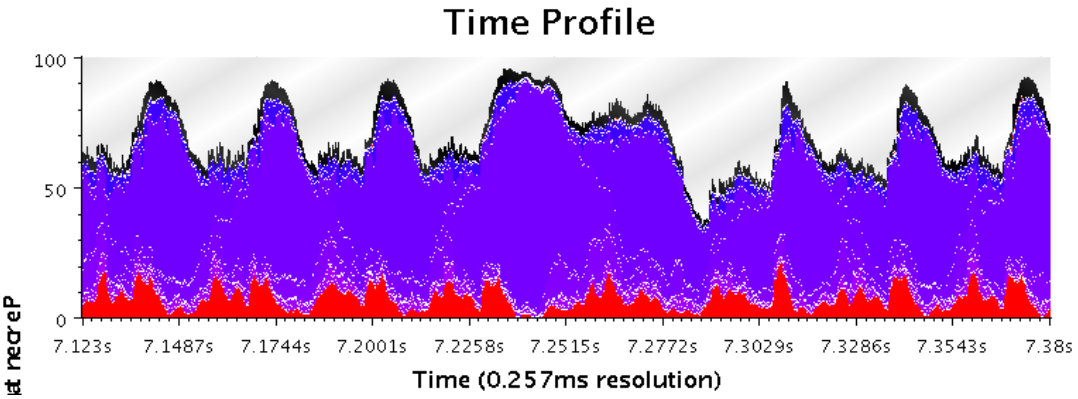


Figure 4.15: Time profile of running NAMD simulation of DHFR with 2 Away XY on 256 processors on BGQ

each patch is split into 4 smaller ones. As a result, atoms in each patch interact with atoms in neighbor patches that are 2 patches away. In this way the amount of computation in both patches and computes decreases. Figure 4.15 shows the time profile of the 256 processor run with 2 Away XY decomposition. It is clearly seen that the idle percentage becomes much lower and the utilization gets much higher. The detail data is shown in the table 4.1. The super compact time line view is displayed in Figure 4.16.

The results of PICS analysis are also used to load the least idle processor, most idle processors, and the processor with the longest entry method execution in Figure 4.17. Studying the table again, we can see that the longest entry method with 2 Away XY is

Name of performance data	1Away	2Away XY
average idle percentage	0.679998	0.263319
average overhead percentage	0.006755	0.071367
average utilization percentage	0.313265	0.665345
maximum entry method duration	0.022648	0.004039
entry method ID with longest duration	172	178
PE where longest entry method happens	132	24
least idle PE	2	1

Table 4.1: PICS performance summary for NAMD simulating DHFR on 256 processors using 1Away and 2Away XY decompositions

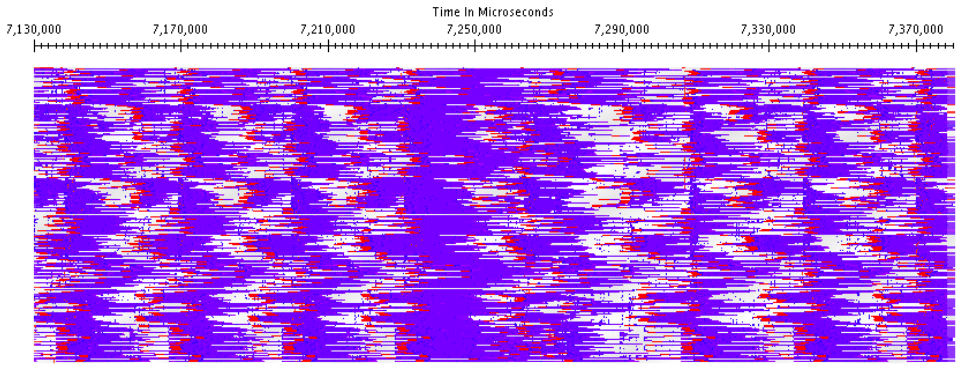


Figure 4.16: Super compact view of time profile of running NAMD simulation of DHFR with 2 Away XY on 256 processors on BGQ

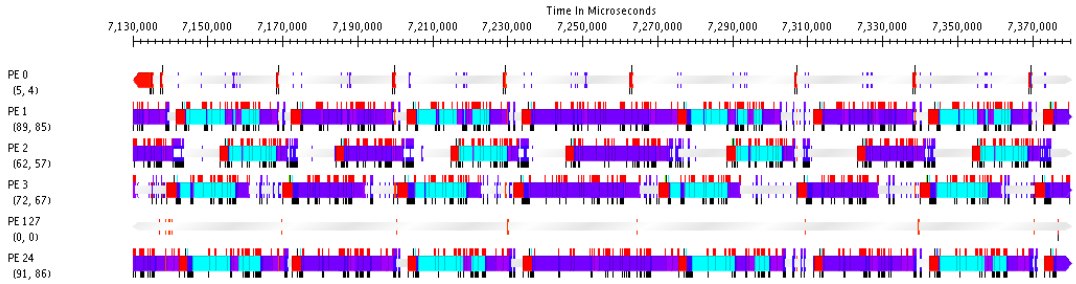


Figure 4.17: Using PICS to load least idle PE, longest entry method

much less than the one with 1 Away XY.

Chapter 5

Steering Runtime System and Applications

In the previous chapter, we have discussed how to perform automatic analysis so as to figure out the program problems and the corresponding solutions. This chapter describes how the performance steering is carried out based on the results of analysis. A few techniques are discussed. And then the categories of runtime system and application control points are listed.

5.1 Control Point Steering

As it is described in Chapter 4, the output of performance analysis is a set of possible performance problems and their corresponding solutions. Each solution contains two parts, an effect and the direction of that effect. On the other hand, when the control points are registered by the applications or the runtime system, they are stored in the controller system database. The solutions are associated with the control points. Therefore, we know what control points to tune and in which direction. The main algorithm is listed:

Steering Strategies

Steering direction : We have proposed and developed three strategies to determine the steering direction.

1. The steering direction is first guided by the performance analysis result. As it is discussed earlier, performance deficiency can be reduced by a set of solutions, which

Listing 5.1: Control point steering

```
void tuneParameters(vector<IntDoubleMap> &effects) {
  for(each eff in effects )
  {
    vector<TunableParamters*> vec_with_effect = database[eff];
    for(each tp in vec_with_effect)
    {
      tp.tune(eff);
    }
  }
}
```

include the control points and steering direction.

2. If the steering direction is not known, we check whether performance of current iteration is better or worse than the previous. Being better means that the previous steering is correct and making progress. The steering will go toward this direction. Otherwise, it means the steering is in the opposite direction so that the steering direction is reversed.

3. There is an exception. When the effect of control point is *exhaustive*, there is no direct relation between the performance and configurations. Therefore, all configurations are searched and best configuration is saved for future use.

Steering Values. When the control points are registered, the minimum modification unit is set by users. We have a few strategies to modify the values.

1. The simplest one is to modify the value by the minimum unit every time. It is slow but tries to search as many configurations as possible.

2. To speed up the search process, we can modify the values by a larger extent if moving toward one direction continuously improves performance. Once the performance degrades, we reduce the modification extent.

3. The other option is to do binary search based on steering direction.

4. Coarse grain first to determine the rough range where the optimal values might be. And then a finer-grain search is performed where smaller steps are moved.

Accelerating the Steering

As we have described, many control points can be inserted into the runtime system and applications, this makes the search space of configurations of control points extremely large. The main approach we take to prune the search space is to perform analysis to narrow down the possible useful control points. Another way that we find useful to prune the search space is to record whether control points are accessed during one step or one phase. Since we have directed all access to control points to a function call which is in the control of PICS, it is easy to record whether it is accessed. In this way, when we tune a control point, if it is not accessed at all, nothing will be done to that control point. If it is accessed and also one possible solution to improve performance based on analysis, that control point is tuned and the flag for access is reset. This seems to be a minor point but it is very useful since it can prune a lot of control points, which are registered in the runtime system but are not relevant to a particular execution.

Handling Oscillation

During our steering, when the same configurations are repeatedly checked for a few times, we might end up in repeating the same process by going forward and backward. In this case, in our system after a few times of same configurations, we continue the run by applying the configurations which provide best performance. In most cases, this works well. However, for applications with dynamic behaviors where best configurations might change during run, PICS handles it by resetting search. In the case that the current performance becomes worse using the previous best configuration, we reset the values for the control points and restart searching. Since our performance analysis could detect the deficiency, this handles the oscillation in applications.

5.2 Handling Conflicting Effects

The output of automatic performance analysis is a set of solutions to eliminate the performance problems. It is very likely that some solutions conflict with each other. For example, if overhead is larger than a threshold, possible solution is to increase the grain size. On the other hand, the idle time is larger than the threshold too. Possible reason is load imbalance.

To handle load imbalance, one solution is to decrease the grain size so that the difference between maximum load and average load is reduced. In this case, the solutions about grain size conflict with each other. We have proposed and developed two strategies to handle this case.

First strategy: determine the dominant effect. By dominant effect, we mean that by solving the problem, it provides larger potential performance improvement. For example, in the above case, assume that the overhead percentage is 20% while idle percentage is 10%. To be extreme, removing overhead will potentially improve the performance by 20% while removing idle will give 10% maximum improvement. Therefore, we choose the effect associated with eliminating overhead. To be general, we calculate the upper bound of potential performance improvement for each performance problem. Based on it, we keep the effects that provide larger improvement.

Now, we describe how the upper bound performance improvement is calculated. In the decision tree, the nodes in the second level include CPU utilization, idle percentage and overhead percentage. The potential improvement for these nodes can be easily calculated. For the utilization node, the maximum improvement percentage is $1 - R_cpu$. For the overhead and idle nodes, the maximum is $R_overheador R_idle$. For the intermediate nodes, the upper bound of potential improvement is no more than that in the parent node. Besides this upper bound, some nodes have its own constraints. For example, for the load imbalance node, if load is perfectly balanced, the maximum improvement is $max_load/avg_load - 1$. In this case, the upper bound of this node is the minimum of parent upper bound and its own constraint. In this way, when traversing the decision tree, each node is also associated with the improvement. The improvement of leaves is inherited from their parents. For each obtained effects, we check whether its counter-wise effects exist or not. If the counter-wise ones exist and have larger impact than the newly obtained ones, we just ignore the new ones. Otherwise, we keep the new ones.

Second strategy : pick up the effects which have no conflicts. In this strategy, we ignore all the control points with conflict effects. Only the ones with single direction of effects are steered. The reason behind this is that it is likely to be wrong prediction for the

effects with conflicts.

Third strategy : For each category of problem, pick up the one without conflict first. When there are conflicts, keep the effect that is relevant with the problem, which has no solution yet.

When a solution is found, we first check whether the opposite solution exists in the higher level solution set. If it does exist, this solution is ignored. Otherwise we store the solution in the corresponding problem set.

Once we have the solutions for each problem category, we process them. If one solution does not conflict with any others, it is kept and the corresponding problem is marked as *hasSolution*. If the solution exists in the higher level solution set, the same action is taken. If one problem category has only one solution, it is kept without checking conflicting with others.

Next phase for the solutions which are uncertain, we process them again. First we check whether it conflicts with the *must keep* set of solutions, it does not conflict, it is kept and added.

5.3 Control Points in the Runtime System

With the applications and hardware becoming more and more complex, it is widely accepted that a powerful and intelligent runtime system should handle more complexity to improve performance and facility the programming.

Traditionally, in order to achieve the best application performance, it is well known that the application configurations should be adapt to the particular execution. The underlying runtime system might not adapt. When studying the runtime system carefully, we find that many runtime configurations are chosen as constant values based on experience. These configurations usually do not change with applications or particular run setting. However, with the systems becoming more complex, the best configurations that work at a small scale might not work well at the large scale. Also the best configurations for some problems might not perform well for the others. Therefore, we believe that the runtime system should also be more adaptive. The runtime system should be steered too. This chapter describes

how to steer the runtime system and what can be adjustable. And then different categories of application control points are presented. A few techniques to steer the performance are discussed in details here.

In order to steer the runtime system, we need to first understand what a typical runtime system performs? As an example, CHARM++ is a representative of this type of runtime systems. Based on our knowledge we have concluded that it is mainly responsible for the lower level communication, task/thread scheduling, resource management and mapping, etc. Now I will describe how each component should be steered and the corresponding control points.

At the bottom of a runtime system is the lower level communication libraries. Depending on the underlying platform, the libraries can be varied a lot. This layer is mainly responsible for communication, including sending/receiving network data, making network progress, and performing some collectives. Depending on the underlying network, the communication protocol and libraries are chosen at the build time. However, some parameters should be adaptive during run time. For example, on Cray machines with Gemini or Aries network, uGNI protocol is used as the low level communication library. In this library, in order to send different size of data, different protocols are applied. For smaller messages, direct connected channels are set up at the initialization phases. Memory space is reserved to store the data before sending to the network and before delivering to the user space. There is a tradeoff about the size of the memory reserved. In small scale, reserving more memory will speed up the small data transferring. However, in large scale run, due to the big size of channels, it takes much memory. As a result, it should not be too large. Also depending on the communication patterns in the applications, it might require or not need to reserve high memory. Therefore, the threshold between defining the small messages and large messages is a tunable parameter in the runtime.

In CHARM++, Converse is a component of the CHARM++ runtime system that provides a unified functionality across all machine layers to CHARM++. For portable and efficient implementations on a variety of parallel machines, this runtime system needs a minimum set of capabilities from the parallel machine, its communication infrastructure, and its node-

control points	Effects	Use Cases
broadcast algorithm selection	communication	most applications
broadcast/reduction branch factor	critical path	most applications(NAMD)
compression algorithm	communication, overhead	NAMD, ChaNGa
seed load balancer period	overhead, load balance	NQuees, UTS
seed load balancer topology	overhead, load balance	NQueens, UTS
fault tolerance frequency	overhead, memory usage	most applications
load balancing frequency	overhead, load balance	most applications
tracing data disk write frequency	memory usage, overhead	most applications
number of AMPI virtual threads	grain size	AMPI applications

Table 5.1: Runtime system control points

level operating system. Therefore, it is desirable to separate machine specific components of Converse into a low-level runtime system, i.e., LRTS [24]. Different machine-specific LRTS implementations can share common implementations such as collective operations (e.g. broadcast) to construct a full Converse layer.

For a supercomputer vendor, LRTS serves as a concise specification of the minimum requirements to implement Charm++ software stack on their platform. This simplifies the work of porting the Charm++ runtime to a new platform since the vendor only needs to implement the functions defined in the LRTS. These LRTS functions are classified into capabilities needed for communication, node-level OS interface (including memory allocation, virtual memory functions, topology information, and timers), support for user level threads, external communication, and fault tolerance.

In Converse layer, after carefully studying the runtime system, we have abstracted a list of control points, which affect the system performance. It is shown in Table 5.1. Among these control points, we will study the following.

Broadcast Based on the unified API functions, we have designed broadcast operations using spanning tree and hypercube virtual topology. However, depending on the problem and system size, the branch factor in spanning tree needs to be tunable. Whether to use spanning tree or hypercube should be tunable too.

Reduction Reduction is performed in the structure of reduction tree. The degree of reduction tree affects the performance on different scales. Therefore, it should be tunable. In pipelined reduction implementation, the size of each fragment affects performance too. Thus, it should be tunable too.

Message compression. Compressing network message reduces the network load and

speeds up the communication, possibly reducing the idle time. However, whether it benefits the overall performance depends on multiple factors, compression/decompression speed, compression/decompression rate, and also network bandwidth. Equation 5.1 represents the time cost of sending one message without compression.

$$T = l_s + \frac{S}{B_w} \tag{5.1}$$

$$T_c = l_s + t_c S + t_d S + \frac{SR}{B_w} \tag{5.2}$$

Compared with the regular communication, the time of transferring data of S Bytes with compression is described in Equation 5.2. Here, l_s is the constant time cost to process the message. t_c is the per Byte cost for compression. t_d is the per Byte cost for decompression. R is compression rate. B_w is the network bandwidth.

Intuitively, applications benefit from compression on the machines with the fast processor and slow network. However, these two factors are fixed to the specific machine. The other factor is compression rate. For one particular message, there is nothing to tune. However, in a real application messages have different compression rate. The time cost of transferring N messages of S_N bytes is shown in Equation 5.3. If messages with low compression rate are transferred, the cost of compression/decompression might be more than the saving of network time. Only when compression rate is higher than some threshold, the overall performance will get better. Therefore, one possible control point is the threshold of the compression rate. The time cost function is represented in Equation 5.4. α is the ratio of messages that have compression rate larger than threshold T.

$$T_c = N \times l_s + (t_c + t_d) \times S_N + \sum_{i=1}^{N} \frac{S_i \times R_i}{B_w} \tag{5.3}$$

$$T_c = N \times l_s + (t_c + \alpha \times t_d) \times S_N + \sum_{i=1}^{\alpha \times N} \frac{S_i \times R_i}{B_w} \tag{5.4}$$

The other control point in compression is the compression algorithm. Many algorithms

can be used with their own advantages and disadvantages. Some algorithms have high compression rate but slow speed while others have low compression rate but fast speed. Depending on the hardware and application data patterns, some algorithms might perform better than the others. So far I have studied four compression algorithms, including ZLIB, LZ4, byte compression, and float compression. ZLIB has very good compression rate but quite slow. LZ4 has both reasonably good compression rate and compression speed. Byte similarity compression performs good when the compression data have higher similarity among bytes. We make this algorithm selection as a control point.

5.4 Control Points in Applications

Most parallel applications are written in a way with many constant or empirical values for performance related parameters. To some extent, the performance of the applications highly depends on the values of these parameters. For some applications, when running on small scales, these values might work reasonably well. However, when scaling to large runs, they might give bad performance. Therefore, these parameters should be tunable by the control system so as to steer applications toward the direction of the better performance. In order for applications to be tunable, we require the application developers to provide some hints for the runtime system.

In this section, a few common control points in the applications are discussed. And then the general categories of control points for various scientific applications are listed.

Steering grain size One crucial control points in most scientific application are the number of parallel tasks (degree of parallelism). When the total amount of problem work is fixed, the degree of parallelism is the inverse of the grain size.

In 8 section, I will present the result of using PICS to steer the Jacobi to find the optimal grain size. Besides Jacobi problems, all the other stencil applications have the similar problem. In matrix related applications, how metrics are decomposed belong to this category. In real applications for example molecular dynamics, how the particles are grouped into patches(the unit of task) is the problem of grain size control. In NAMD, depending on the number of processors running, we carefully choose between 1-Away, 2-

Away X, 2-Away XY, 2-Way XYZ. So far how these are chosen totally depends on user's experience. Instead, our introspective runtime system should automate the choice.

All the above examples are about scientific applications with regular computation pattern. The grain size control also exists in state space search problems, where the search is a tree structure. Determining whether to perform sub tree search in parallel or in sequential significantly affects the performance. I have proposed and designed an adaptive grain size control algorithm to steer this type of applications. Main idea is to split the sequential tasks into parallel ones when the computation granularity of one task is more than some threshold. This work is published in [33] and [34].

Steering for message aggregation

The other important category of control points is in message aggregation. When we strong scale an application, the granularity of each task is decreasing. Meanwhile, the communication data of each task is decreasing too. Especially with the over decomposition runtime system, the communication data of each message can be very small, as small as a few bytes. In this case, the overhead of sending, processing and receiving each small message is relatively high. In worst case, the overhead can dominate the overall execution time. One way to decrease the overhead is to aggregate multiple small messages into one big message. However, how many messages should be aggregated together is a problem. Our introspective runtime should be able to adaptively choose the best aggregation size on different platforms. In 8, I will discuss how the runtime system help find the optimal aggregation size and improve application performance.

Other application control points are summarized in table 5.2. We list the type of tunable parameters, their effects on applications and their use cases.

5.5 Application Reconfiguration

As described above, the values of control points are tuned by PICS and ready to be fed back in to the runtime system and applications. The last step is that the runtime and applications should be able to adapt to the new values. Only when the new values are used, all the tuning will be effective.

control points	Effects	Use Cases
sub-block size	parallelism,grain size	Jacobi,Wave,Stencil
parallel threshold	parallelism,overhead,grain size	state space search
stages in pipeline	number of messages,message size	pipeline collectives
algorithm selection	degree of parallelism,grain size	plan/pencil 3DFFT
software cache size	memory usage,communication	ChaNGa
mapping scheme	number of msgs/Bytes, hops	most applications
ratio of GPU CPU load	computation,load balance	NAMD, ChaNGa

Table 5.2: Application control points

Depending on the specific control points and applications, adapting to the new configurations can be easy or difficult. We have summarized the following categories about how the new values are adapted.

Direct Use

For some control points, the applications are written in the way that the values can be easily changed during execution. Without using PICS, that values can be a fixed number or from the user input. With PICS, just minor modification is required to be adaptive. Instead, the new values are obtained from the PICS and directly applied for next execution. For example, in a message pipeline program, list 5.2 compares the regular function without using PICS and that with PICS. Comparing two function, it can be seen that the only modification is to add get function to get the value from PICS. Everything else is same. This is the easiest case to use PICS.

Data Redistribution

For a lot of applications, it requires more actions to be adapt to the new configurations. Most important action is to redistribute the data. Especially when the decomposition is changed, the number of tasks are changed, the data associated with each task should be re-distributed. An example is Jacobi program. As we described earlier, the sub-block size or the number of tasks should be adaptive to achieve the best performance. During execution, once the sub-block size is changed, the objects should be rebuilt, the data in the old object should be distributed correctly to the new objects. The applications should be written in the way to be able to correctly handle the data movement. Figure 5.1 shows how one task is split into four tasks and how data are moved to the new tasks.

Listing 5.2: Direct use of new configurations

```
//-------------------------Without PICS -----------------
  void regularSend() {
    int frags = 1;
    int size = payload/frags;
    for(int i=0; i<frags; i++)
    {
      sum += doWork(workLoad/frags);
      PingMsg *msg = new (size) PingMsg();
      memset(msg->x, sum, sizeof(double));
      thisProxy[1].recv(msg);
    }
  }

//------------------------- With PICS ------------------
void sendWithPICS() {
  int frags = (int)PICS_getTunedParameter("PIPELINE_NUM", &valid);
  int size = payload/frags;
  for(int i=0; i<frags; i++)
  {
    sum += doWork(workLoad/frags);
    PingMsg *msg = new (size) PingMsg();
    memset(msg->x, sum, sizeof(double));
    thisProxy[1].recv(msg);
  }
}
```

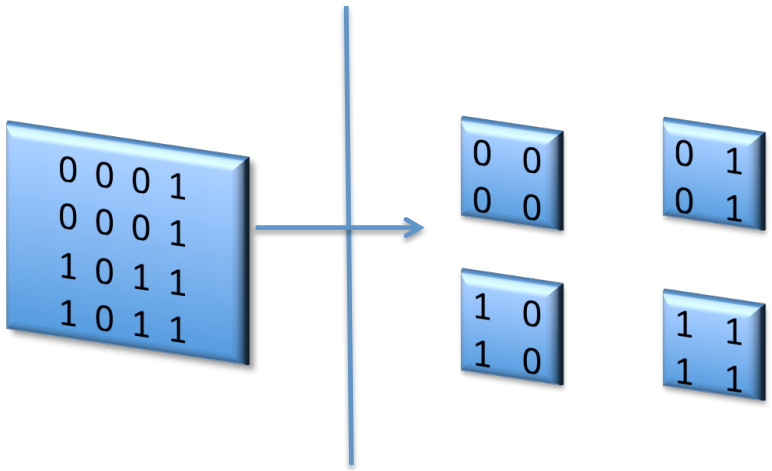


Figure 5.1: Data redistribution when one task is split into four

Data Structure Reconstruction

In this category of control points, once their values are changed during executions, it requires a lot of modifications in the applications, including applying new values, data redistribution, and even data structure rebuild. In this case, in case of doing the complicated reconfiguration during execution, we pretend to exit the program and restart with new configurations. One example of this type of applications is NAMD.

In NAMD the decomposition is an important control point. Usually it is read from the simulation config file. The value is based on users' experience. However, it is not necessarily the best. Once it is read, the whole program is built based on the values, it determines the number of patches, the number of compute objects, the number of pencils, the multi-cast tree construction. Almost everything in the program depends on the decomposition. They are constructed at the start up phase. Once the program is running, it is very complicated to change the decomposition. In this case, we just simply record the tuned values from PICS and exit the program. And then the program is restarted with the new values that is recorded from PICS.

Chapter 6

Scalable and Distributed Framework

In the above chapters, we have discussed the techniques for performance analysis and tuning. It is designed in the centralized fashion, where the data is collected to one processing unit, analysis and tuning is performed on it. This is good in the aspects that it provides most accurate results. However, when we scale to large number of processors, the analysis and tuning itself can be the performance bottleneck. In order to solve this problem, this chapter discusses the ideas and techniques to make PICS more scalable and distributed.

6.1 Scalable Performance Analysis

After the data is measured, it is collected and analyzed. The next question is who performs analysis and steering. For data collection, we use spanning-tree based reduction algorithm. All the processors are organized in a tree structure. Each processor except the root has some parents. All processors except the leaves have some children based on a branch factor. The reduction starts from the leaf processors, which contribute its data to parents. The internal parent nodes wait for data from all children to arrive. The data is combined and processed. The parent continues to contribute the processed data till reaching the root of the spanning tree. The straightforward and traditional way is to collect all data to the root of the spanning tree, which is called the central server. Since this central server has the knowledge from all processors, it performs the most accurate and comprehensive analysis. This scheme works well for small scale runs. Although it is easy to implement and provides

accurate results, all data is processed by one processor using sequential analysis. This poses a bottleneck when scaling to large number of processors.

Here we discuss a few modes to overcome this bottleneck.

Types of control points

First, let us exam the features of the control points which need to be steered. In general, there are two categories of control points that we need to handle differently.

- Control points with different configurations on different processors. That means that the values of control points on different sets of processors can be independent. They do not have to be the same. In this case, each processor can steer the configurations independently. As a result, the performance analysis and steering process can be fully distributed.

- Control points with same configurations on all processors. That means configurations must be same any time on all processors. In this case, even though we can steer configurations independently, only one set can be used.

Options for analysis and steering Based on the types of control points described above, the number of configurations generated by PICS once, whether the performance data is fully or partially collected, the following options are summarized as shown in figure 6.1.

- **Full or Partial**: When the analysis is performed, either performance data from all processors or from partial processors is collected. In *Full* mode, data from all processors are collected and analyzed. It provides more accurate analysis but does not scale well. In *Partial* mode, only data from a set of processors are collected. It can reduce the amount of communication in the system and speed up the collect process. The partial information collection overcomes this drawback in the satisfying the accuracy.

- **Single or Multiple**: When performance steering is carried out, new sets of configurations are generated. Either single set or multiple sets of new configurations are generated. Traditionally, when performance analysis is done, the controller generate

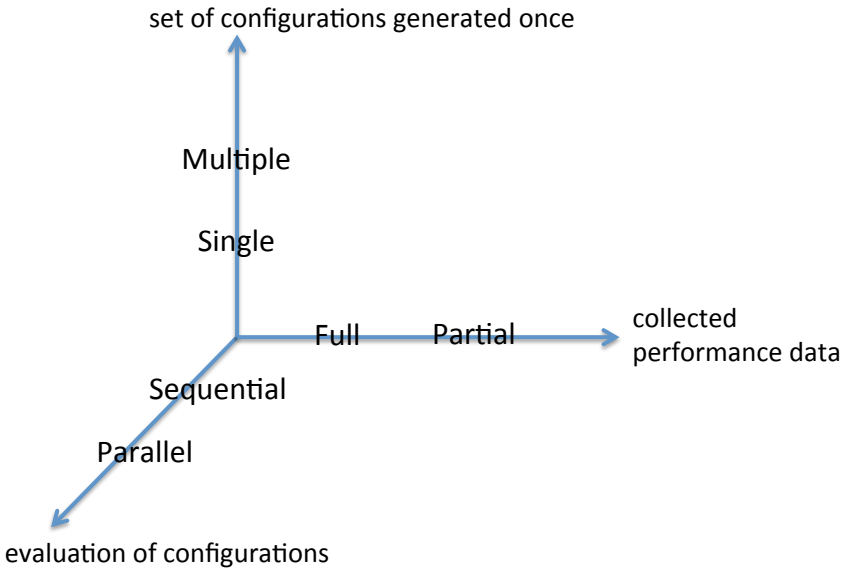


Figure 6.1: Modes of analysis and steering

one set of configurations of control point for next execution. There is another option. When performance analysis is performed once, the controller generates multiple sets of configurations. The multiple configurations can be evaluated in parallel or in multiple steps. This reduces the time of performance collection and analysis. It is more efficient. The drawback of this mode is that it is possible that some configurations are in the direction of degrading performance. It is a waste of time to evaluate these sets of configurations.

- **Sequential or Parallel**: As described, there are two types of control points. For some control points, the values must be same for all processors. This is due to application requirement. While, for other control points, the configurations can be different on each processor. For the first case, at one time only set of configurations are applied on all processors and evaluated. For the second case, multiple sets of configurations can be evaluated in parallel within multiple groups.

Although in theory there are eight total combinations of the above options, some of them does not make sense. For example, if single configuration is generated once, parallel evaluation is impossible. Due to some application requirement, if the configurations have

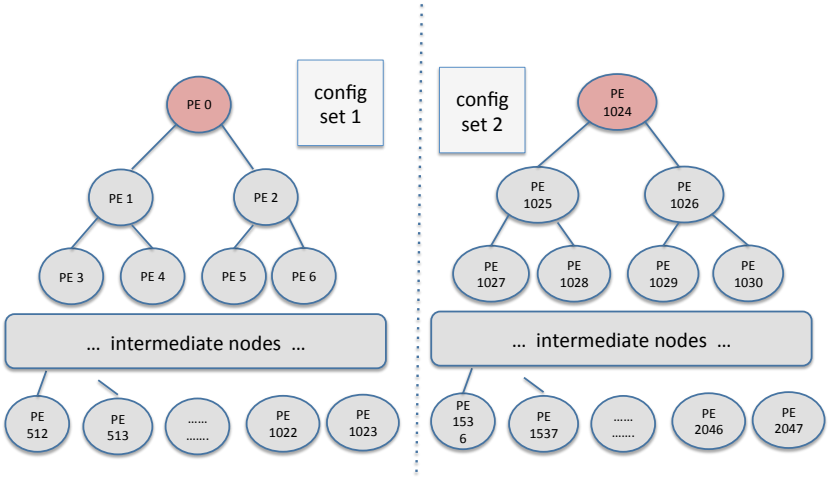


Figure 6.2: Independent configurations

to be same on all processors, the parallel evaluation does not make sense.

6.2 Distributed Analysis

In the above options, there is one option that we can make future optimization. When multiple configurations are generated and evaluated in parallel, we can do it in a totally distributed way. The processors are divided into groups based on the group size. Each group has a group leader, where all the other processors in this group contribute the data to. The group leaders are responsible for analyzing the performance data and deciding the next configurations for the control points. When the decisions are made, the new configurations are multicast to the processors within the group. This is illustrated in Figure 6.2

Choose among multiple configurations

Here rises a problem - since each group leader independently makes its own decision, the configurations of same control point on different processors are very likely different shown in Figure 6.2. Depending on the scenario, this can be right or wrong.

If the configurations are required to be same, we need to tweak this strategy. The idea is to do a global steering as shown in Figure 6.4. After each group leader makes its own decision, the results are contributed to a global root processor. This processor determines which configurations are best to choose. This set of configurations are broadcast to every

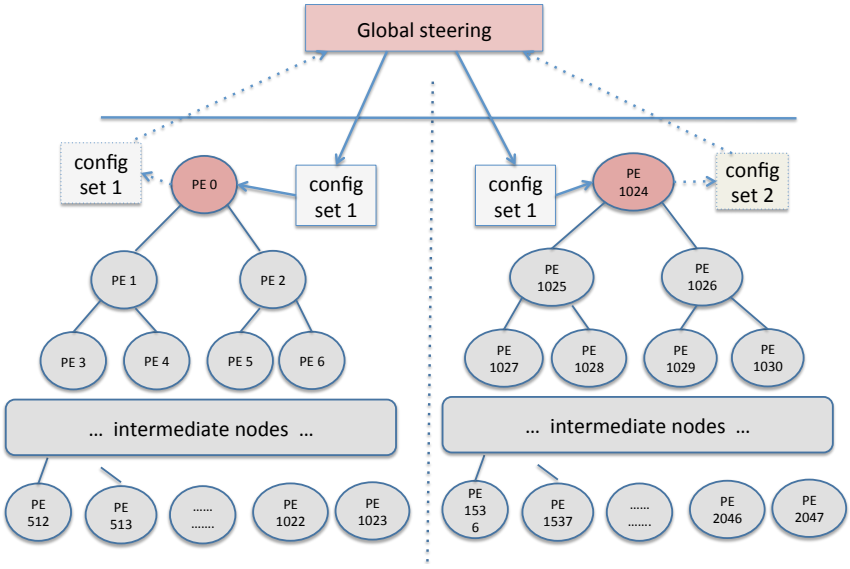


Figure 6.3: Independent configurations with global steering

processor so that every one has the same configurations.

If the configurations on different processors are not required to be same, multiple sets of configurations can be evaluated in parallel. This is shown in Figure 6.4. A global steering is performed to generate multiple sets of configurations. Each set of configuration is sent to one group leader, which is responsible to distribute to all group members.

6.3 The Process of Scalable Tuning

Based on the different modes described above, in PICS, there are the following steering processes.

1. Performance analysis is performed, and then steering is carried out to generate one set of configurations. This set of configurations is applied on all processors. The process is illustrated in Figure 6.5(a).

2. Performance analysis is performed, and then steering is carried out to generate multiple sets of configurations. These configurations are stored in the control system. In the next a few rounds, each time one set of configurations is applied on all processors and evaluated. Multiple sets of configurations are evaluated sequentially. And one

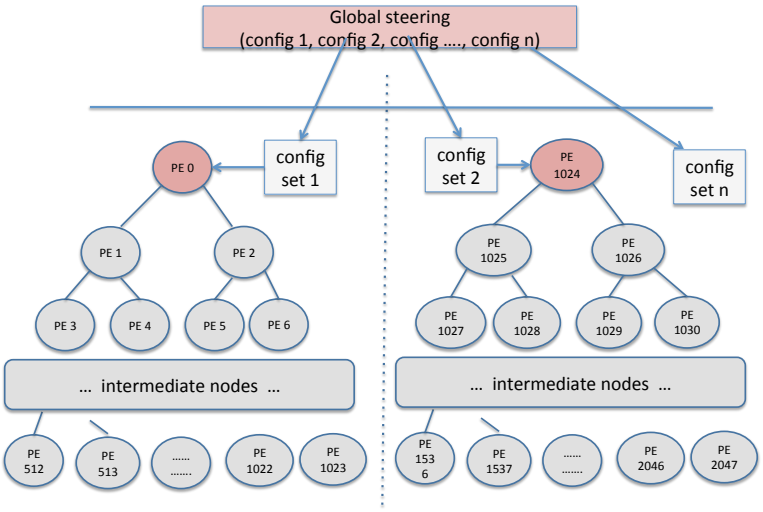
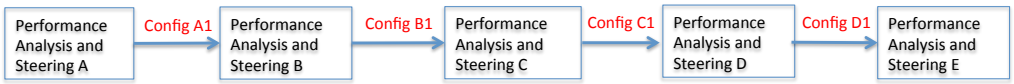


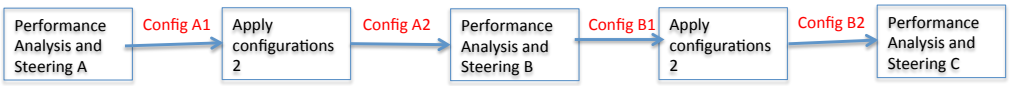
Figure 6.4: Multiple configurations with global steering

set is chosen among them to be the base of next performance steering. This is shown in Figure 6.5(b).

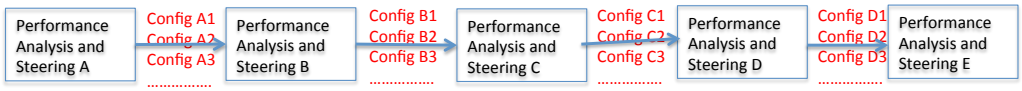
3. Performance analysis is performed once, multiple sets of configurations are generated once. These multiple set of configurations are evaluated in parallel. It is shown in Figure 6.5(c).



(a) Single analysis, single set of configurations, single evaluation



(b) Single analysis, multiple sets of configurations, single evaluation



(c) Single analysis, multiple sets of configurations, parallel evaluation

Figure 6.5: Three types of analysis and steering processes

Chapter 7

Implementation in Charm++

To investigate the appropriate mechanisms required to add control points to parallel applications, PICS has been developed within the CHARM++ runtime system. The framework is capable of observing performance characteristics across the parallel machine and storing that information along with the past history of control point configurations for a running program. Once the framework decides how to adapt the behavior of a parallel program, it can enact the changes through a callback to the program.

7.1 PICS Framework in Charm++

CHARM++ is a message-driven parallel programming paradigm. CHARM++ programs are written mostly in C++, with portions in Fortran, C, or other languages if necessary. A CHARM++ program consists of collections of worker objects called *chares* that are mapped onto processors by the runtime system. The chares communicate with each other predominantly by invoking *entry methods* asynchronously and remotely on each other. The runtime system can instrument the computation and communication loads and can remap chares to processors in order to perform dynamic load balancing. The standard practice in writing CHARM++ programs is to over-decompose the problem so that there exist many chares on each processor. A scheduler on each processor executes the available entry method invocations once at a time. We have implemented PICS system in CHARM++ parallel programming system. The PICS framework is implemented as a set of chare objects, one

instantiated on each processor. This allows the communication and computation performed by the framework to be automatically interleaved with the execution of the program, leveraging the message-driven scheduling.

Measurement Gathering. CHARM++ contains mechanisms to measure certain performance characteristics of a running program. To gather measurements that are useful for automatic performance analysis and tuning, we have developed a new custom tracing module. The new tracing module records the amount of time spent in each type of entry method, time spent idle, time spent in overhead (the remaining unaccounted for time) on each processor, number of messages, and communication volume. The overhead time represents time spent in the runtime system for handling communication and scheduling. The measurements produced by the tracing module are used by the control system when it tries to make automatic performance analysis. Thus it is important to gather measurements that will likely inform the decision making process. These measurements are general and abstracted away from the behavioral effects produced by varying a control point.

7.2 Control Point API

In our framework, a control point has the structure as shown in Listing 7.1. It includes its name, value type, value range, the unit of change, and the approach to change its value. Besides these, the important fields are the effect and direction of effect as described in Section 3.3. The effects of control points are the bridge between automatic performance analysis and performance tuning. The result of performance analysis is correlated to the effect of some control points. The strategy field is used to select the search algorithms for the control points, which have no obvious effect. For example, when the possible configurations of control points are quite few, exhaustive search can be used for accuracy. The *arrayID* field is used to locate what tasks are affected by this control point. This is useful for real applications, which contain various types of tasks. After control points are defined, they are registered to the control system by calling *registerTunableParameter(ControlPoint *tp)*. This interface is uniform for registering both runtime system and application control points.

The effects of control points are defined in the PICS system as shown in list 7.2.

Listing 7.1: Struct of control point

```
typedef struct __controlpoint
{
  char name[30];
  enum TP_DATATYPE datatype;
  double defaultValue;
  double currentValue;
  double minValue;
  double maxValue;
  double bestValue;
  double moveUnit;
  int moveOP;
  int effect;
  int effectDirection;
  int strategy;
  double effectScale;
  int arrayID;
} ControlPoint;

void registerControlPoint(ControlPoint *tp);
```

Listing 7.2: Effects in PICS

```
enum Effect_t {
  PICS_EFF_PERFGOOD=0,
  PICS_EFF_GRAINSIZE,
  PICS_EFF_AGGREGATION,
  PICS_EFF_COMPRESSION,
  PICS_EFF_REPLICA,
  PICS_EFF_LDBFREQUENCY,
  PICS_EFF_NODESIZE,
  PICS_EFF_MESSAGESIZE,
  PICS_EFF_UNKNOWN
};
```

```
double getTunedParameter(const char *name, bool *valid);
```

Listing 7.3: APIs for steps and phases

```
void PICS_setNumOfPhases(int fromGlobal, int num, char *names[]);

void PICS_startStep();
void PICS_endStep();

void PICS_startPhase(int phaseId);
void PICS_endPhase();
```

The application acquires a new configuration for the control points by calling a simple function named *getTunedParameter*. This function takes the name of the control point and a bool pointer. When it returns, if the value of the bool is true, it returns the tuned value. Otherwise it means the control point does not exist yet. The API is as follows.

Besides registering control points, application users also need to tell the control system about the pattern of the application. We have provided APIs for two types of applications, including applications with steps and without steps. Most scientific applications generally compose of a sequence of steps. At the end of a step, corresponding performance data is collected, analysis is performed, and any required tuning is done. The API for applications to mark the steps of applications is *startStep* and *endStep*. For applications that contain multiple phases in one step, *startPhase* and *endPhase* are provided to mark these phases. The API is shown in Listing 7.3.

Applications also need to provide a callback to tell the control system how to continue when the performance steering is done. The callback is a standard CHARM++ callback provided at startup by the application through a registration call such as:

When there is no obvious step boundary in the applications, performance analysis and steering is done periodically. In this case, the applications do not have to tell the step boundaries. Neither the callback is required. The PICS performs all these in the background.

Besides the APIs exposed to the applications, there are also relevant runtime options

```
void registerAutoPerfDone(CkCallback cb);
```

available to control PICS.

1. +auto-pics : PICS analysis and steering happens periodically instead of steps. This is mainly useful for applications without iterations.

2. +picsGroupSize : This is for the scalable analysis to determine the number of processor in one group.

3. +picsCollectionMode : This is also for scalable analysis to determine whether to perform data gather on all processors or partial processors.

4. +picsOutput : Set the filename to store the results of performance analysis.

7.3 System Design of PICS

Figure 7.1 illustrates the design of PICS. *Trace* is a common class in CHARM++ to provide the tracing APIs to the runtime and applications. All the specific tracing modules are inherited from this class and implement the functions for their own purpose. In PICS, *TraceAutoPerf* inherits from *Trace* to trace different events to generate the performance data. The performance data is stored in the *PerfData* temporarily. Once the analysis is started, it is over-written. The old data is saved to *DataBase*. Here all the tracing happens on each processor. Once it is time to perform collection, *TraceAutoPerf* does the most work. It first collects all data to the group leader or one root processor. Then performance analysis is carried out. As described earlier, the analysis is based on decision trees with expert knowledge rules. The decision tree is implemented in *Decision Tree* class, which consists of Tree nodes from *TreeNode*. Once the analysis is done, the results are output for Projections use or it is fed into *TraceAutoTuner*. TraceAutoTuner has an instance on each processor, which is responsible to communicate with TraceAutoPerf and Control Point database. TraceAutoTuner finally calls *ParameterDatabase* to steer the control points. All the history of control points are stored in ParameterDatabase. Based on the input from TraceAutoTuner and the history data, performance steering is done. The interface for the runtime system and applications to use PICS is provided in *picsAutoPerfAPI* and

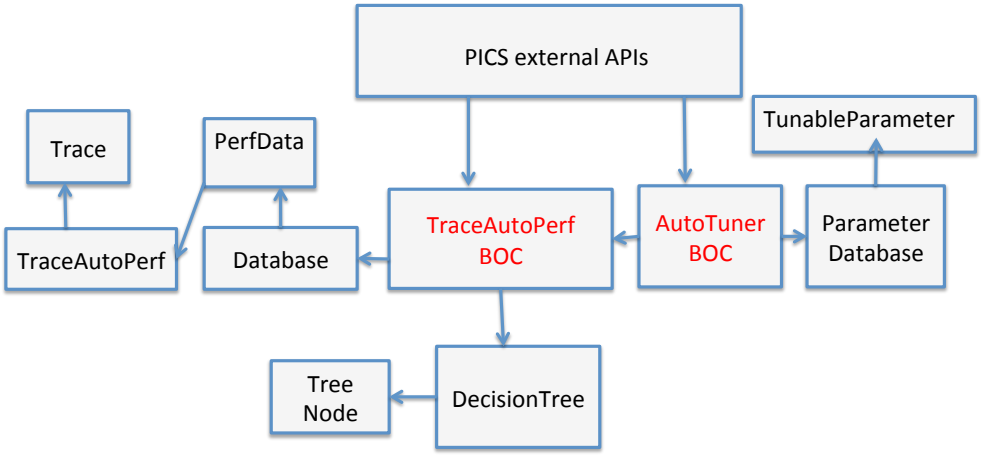


Figure 7.1: Design of PICS System

picsAutoTunerAPI.

Chapter 8

Integration and Validation with Benchmarks

In this chapter four synthetic benchmarks are studied to demonstrate how to apply PICS to tune them. For each benchmark, we describe what control points are added in the benchmarks, how these are related to performance, and how they are steered by PICS.

8.1 Message Pipeline

In this benchmark, processor A sends a 2MB message to processor B. Before sending the message, processor A performs some amount of computation. After B receives the message, it performs computation too. The 2MB message can be broken into multiple pieces to be pipelined. Whenever a portion of computation is finished, one piece of the message can be sent out. Whenever process B receives the message, it performs the corresponding portion of computation. In this case, the runtime must determine how many pieces the 2MB data should be broken into. Depending on the amount of work and the platform, the optimal value varies. Therefore, the control point in this benchmark is the number of pipeline messages. It affects the overlap of the computation and communication. Increasing its value improves the overlap, while it also increases the overhead. Every time the value is changed, the basic adjustment unit is 1. This means that the number of pipeline messages

can be increased or decreased by 1. List 8.1 shows the code to register control point, obtain new value from PICS, tell the applications patterns and callback function.

Listing 8.1: Control point in message pipeling

```
PICS_registerTunableParameterFields("PIPELINE_NUM", TP_INT, dv, minv, maxv
    , 1, PICS_EFF_GRAINSIZE, -1, OP_ADD, TS_SIMPLE);

int frags = (int)PICS_getTunedParameter("PIPELINE_NUM", &valid);

PICS_startStep(true);
PICS_startPhase(true, 0);

//do work

PICS_endPhase(true);
PICS_endStepResumeCb(true, CkCallbackResumeThread());
```

Figure 8.1 illustrates the process of using PICS to find the optimal number of pipeline messages for two cases with different amount of computation. When the number of pipeline messages is small, the program characteristic PICS observes is the high idle time, the computation is not overlapping communication enough. The corresponding solution is to increase the number of pipeline messages. Figure 8.2 compares how pipelining improves the overlap of computation and communication causing a decrease in time per step. In the figure, white represents the idle time. Blue represents work on the sender side and yellow stands for work on the receiver side. However, when the number is large, high overhead is observed which suggests that the number of pipeline messages should be decreased. During this process, PICS saves the configurations and their performance results for each tuning step. When configurations are repeatedly searched three times, the best configurations among the previous runs will be chosen. For the case with little computation, the optimal performance is achieved when using 4 pipeline messages. Meanwhile, for the case with more computation, the optimal performance is obtained using 9 pipeline messages. In both cases, the optimal values are found within 20 steps and the configurations are used for the rest of

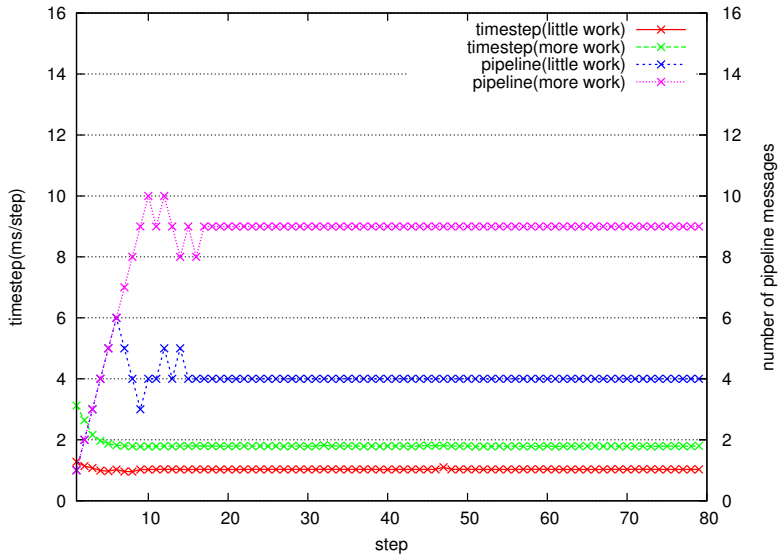
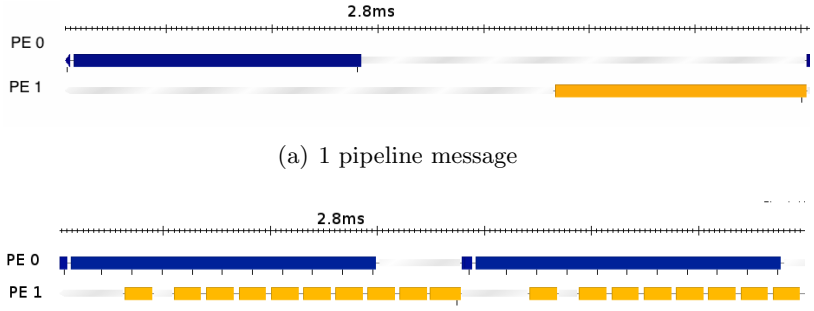


Figure 8.1: Tuning the number of pipeline messages to optimize performance



(a) 1 pipeline message

(b) 10 pipeline messages

Figure 8.2: Timeline of pipeline transferring using 1 message and 10 messages

the run.

8.2 Message Compression

Compressing communication data reduces the network load and accelerates the communication, possibly improving the performance. However, whether it really benefits the performance depends on multiple factors, compression/decompression speed, compression/decompression ratio, and network bandwidth. In order to demonstrate how PICS can be applied to tune message compression, we have developed a synthetic all-to-all benchmark. Each processor sends two messages to all the other processors. These two messages have

different patterns and potentially have different compression ratio based on their content. In our runtime system, we have 5 compression algorithms. Some of them have high compression ratio but slow speed, like zlib. Others have fast speed but low compression ratio. No compression is also considered as a possibility. The goal of applying PICS is to determine whether to use compression and what compression algorithm to use for each type of messages. These are control points in the runtime system. In this benchmark, we have two control points associated with these two types of messages.

Listing 8.2: Control point in message compression

```
TunableParameter *tp_compress_algo = new TunableParameter( "
    RTS_compression_algo", TP_INT, 1, 0, 4, 1, OP_ADD, UNKNOWN, 1,
    TS_EXHAUSTIVE);

CkpvAccess(allParametersDatabase)->insert( "RTS_compression_algo",
    tp_compress_algo);

int compress_algo_index = (int)PICS_getTunedParameter( "
    RTS_compression_algo", &valid);
```

Figure 8.3 shows the process of steering the benchmark and finding the optimal performance for 128KB all-to-all running on 128 cores of Vesta. Different curves represent cases of messages with different compression ratio, which is controlled by r parameter. The lower the r is, the higher compression ratio the messages have. The program characteristic PICS identified in this benchmark is that byte per message is high so as to suggest using compression to reduce communication. However, it is unclear how well each compression algorithm performs on each type of messages. Therefore, PICS tries exhaustive search for possible configurations. PICS tunes the first control point for one message type, and determines the best value for it. After this best configuration for one control point is fixed, PICS steers another control point for the best value. In all three cases shown in the figure, the final performance is stable and improved. However, the best configurations for using compression in three cases vary.

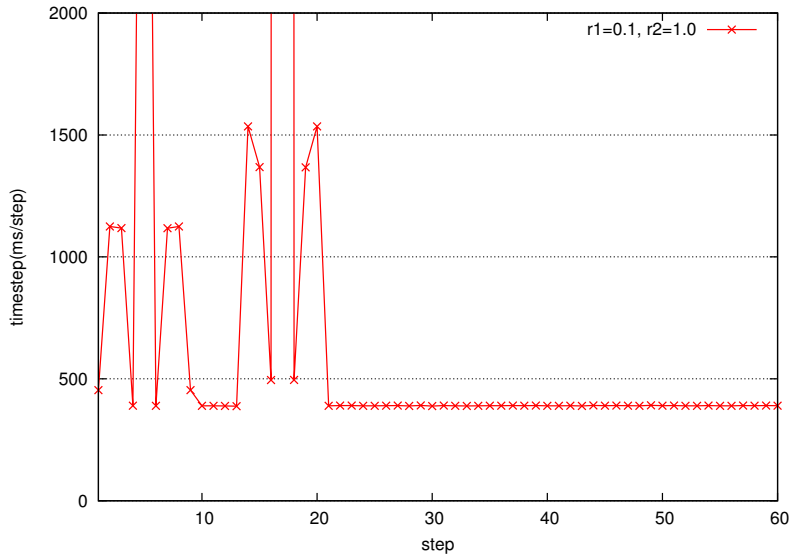


Figure 8.3: Steering the compression algorithm for all-to-all benchmark

8.3 Jacobi3D Stencil Code

This experiment is to steer the grain size of a Jacobi3D relaxation kernel code. Traditionally, the number of tasks equals the number of processor-cores. However, this might not provide the best performance in some cases.

In this experiment, three changes are made to the Jacobi3D code to use PICS. The first is to register the control points. The three control points are the sub-problem size in X, Y, and Z dimensions. The effect associated with the control points are the granularity. When the values of the control points increase, the granularity of tasks increases too. In this problem, every time when the value is changed, it either multiplies by 2 or 0.5 to make the sub-block size a power of 2. The second change is to call the *autoPerfGlobalNextStep()* API function to perform the steering at a global synchronization point. Third, the application needs to implement the *redistribute()* function to re-distribute data into the new decomposition. In this example, either the original data block is split into small blocks or multiple small blocks are merged into one bigger block. The data distribution is relatively regular and simple. So far, we ask the developers to implement this function because only developers have the knowledge of their data decomposition and layout and know how to redistribute the data. Figure 8.4 illustrates how PICS steers Jacobi3D in determining the best sub-block sizes.

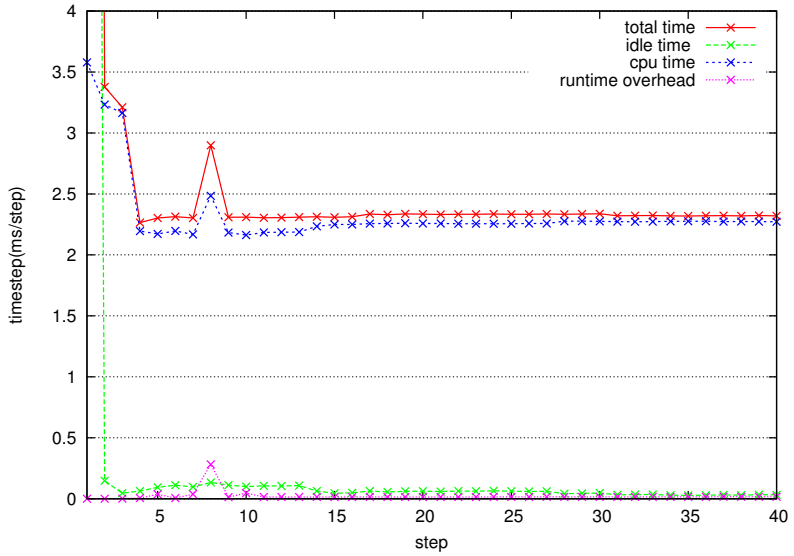


Figure 8.4: Jacobi3d performance steering on 64 cores for problem of 1024

The test is running on 64 cores JYC. Three major factors are taken into account for the performance steering: cache misses, idle time and the runtime overhead associated with parallel objects. When the grain size decreases, data may fit in the cache, which improves performance. However, as the grain size decreases and the number of tasks increases, the runtime overhead may dominate leading to degraded performance. In the figure, when there is not enough tasks for the 64 cores, the idle time is high. When the sub block size decreases, the idle time decreases due to better load balancing. Also due to small sub-block problem, cache miss decreases so that the CPU time decreases. However, when there are too many tasks, the overhead overcomes the benefit of locality and over-decomposition. Therefore, the overall performance decreases. At the end, the optimal value is 64 tasks per core, which gives best cache locality, least idle time, and relatively low overhead.

8.4 Message Aggregation Benchmark

Another important use case of PICS is to control communication, for example, for message aggregation. Message aggregation as well as a few other communication optimizations including dynamic routing have been implemented in TRAM library [35] in Charm++. TRAM combines multiples small messages into big messages and chooses the optimal virtual

Listing 8.3: Register control points

```
//register the call back function when performance steering is done
CkCallback cb(CkIndex_Main::newRun(), mainProxy);
registerAutoPerfDone( cb, false);

//register X dimension sub-block size as the tunable parameter
ControllableParameter blockXMsg;
strcpy(blockXMsg.name, "blockDimX");
blockXMsg.datatype = TP_INT;
blockXMsg.defaultValue = blockDimX;
blockXMsg.minValue = 1;
blockXMsg.maxValue = arrayDimX;
blockXMsg.moveUnit = 2 ;
blockXMsg.moveOP = OP_MUL;
blockXMsg.effect = GRAINSIZE;
blockXMsg.effectDirection = 1;
//This is important for multiple chare arrays --
blockXMsg.chareArray = jacobi;
registerTunableParameter(&blockXMsg);

ControllableParameter blockYMsg;
strcpy(blockYMsg.name, "blockDimY");
blockYMsg.datatype = TP_INT;
blockYMsg.defaultValue = blockDimY;
blockYMsg.minValue = 1;
blockYMsg.maxValue = arrayDimY;
blockYMsg.moveUnit = 2 ;
blockYMsg.moveOP = OP_MUL;
blockYMsg.effect = GRAINSIZE;
blockYMsg.effectDirection = 1;
registerTunableParameter(&blockYMsg);

ControllableParameter blockZMsg;
strcpy(blockZMsg.name, "blockDimZ");
blockZMsg.datatype = TP_INT;
blockZMsg.defaultValue = blockDimZ;
blockZMsg.minValue = 1;
blockZMsg.maxValue = arrayDimZ;
blockZMsg.moveUnit = 2 ;
blockZMsg.moveOP = OP_MUL;
blockZMsg.effect = GRAINSIZE;
blockZMsg.effectDirection = 1;
registerTunableParameter(&blockZMsg);
```

Listing 8.4: Obtain new configurations

```
blockDimX = (int)getTunedParameter("blockDimX");
blockDimY = (int)getTunedParameter("blockDimY");
blockDimZ = (int)getTunedParameter("blockDimZ");

newarray.redistribute(....)
```

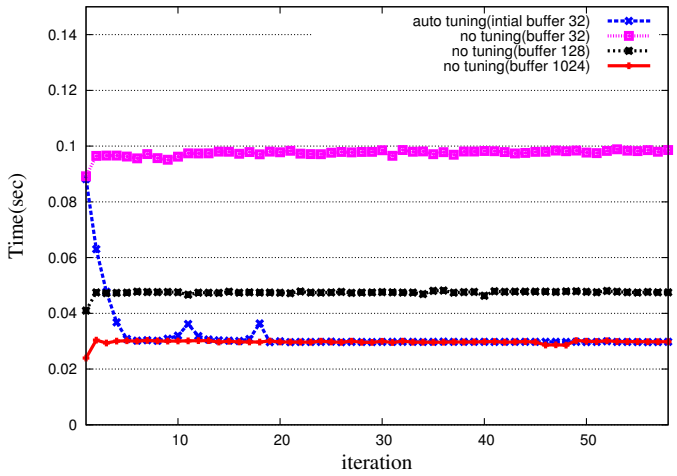
Listing 8.5: Data redistribution interface

```
void redistribute(CProxy_Jacobi newarray, double splitFactor) {
  if(splitFactor > 1 ) { //split
    distribute its data into sub array elements
  }else //merge
  {
    calculate the merged index and send data into merged elements
  }
}
```

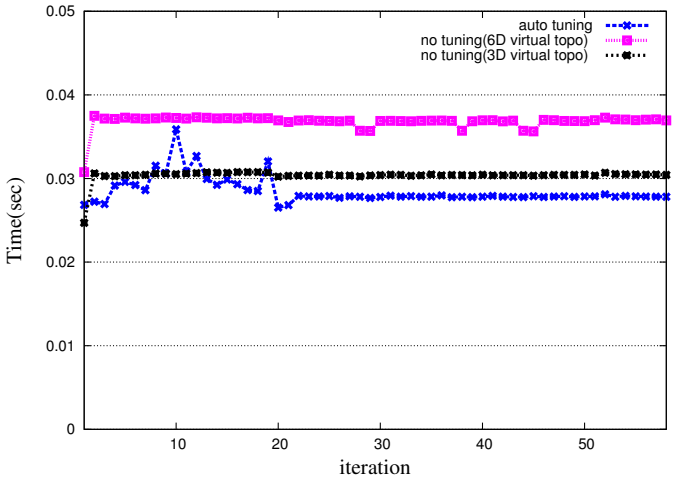
topology for routing. However, it can not automatically decide the optimal buffer size and virtual topology, instead, it relies on the user input.

In this experiment, we apply our PICS techniques to automate the selection of the following library parameters. The first parameter we add is to control the aggregated message size. When the size is small, it can not fully utilize the benefit of aggregation. When the size is big, the overlap of communication and computation can not be fully achieved. Figure 8.5(a) shows the results of using PICS to achieve optimized performance. We can see that without using PICS, the performance becomes stable and much better than the initial performance.

Figure 8.5(b) shows how we use PICS to select the optimal virtual topology. The test is run on a BGQ machine. The BGQ network is a 5-D torus. Based on this physical topology and the way to combine the dimensions, we have around total of 10 different virtual topologies to evaluate. It is challenging to apply the guided search approach to determine the best virtual topology because of the lack of correlation between a particular virtual topology and the overall performance. However, since the search space is relatively small with only 10 possible virtual topologies to compare, PICS can use the exhaustive search to find the optimal configuration.



(a) Automatically steering buffer size



(b) Automatically steering topology

Figure 8.5: Using PICS to automatically steer TRAM buffer and topology for all-to-all benchmark on BlueGene/Q

Chapter 9

Real-world Applications Study

The ultimate goal of PICS is to be applied to real-world applications. In the previous chapter we have shown that PICS can effectively tune simple benchmarks and mini apps. This chapter discusses the utility of PICS on several real applications to handle more complex scenarios.

9.1 ParSSSE Tuning

As described in Section 2.4, state space search problems are commonly encountered in various fields. Two important features of this type of application are that it requires huge amount of computation and they are inherently parallel. There has been a great deal of interest in developing parallel methods [7,36] for such problems. We have developed ParSSSE - a Parallel State Space Search Engine - to improve both the programming productivity and performance [33, 34].

In ParSSSE, adaptive grain size control has been proposed and implemented to improve the parallel performance. The idea of adaptive grain size control is to split the sequential execution based on a threshold value, which is the maximum allowable execution time for single task. Figure 9.1 shows an example of searching the tree and spawning the parallel tasks. Even we have developed adaptive grain size control technique, there is still one question about when to split the sequential queue to multiple tasks. If we split the queue when the amount of sequential execution is too small, there is too much overhead associated

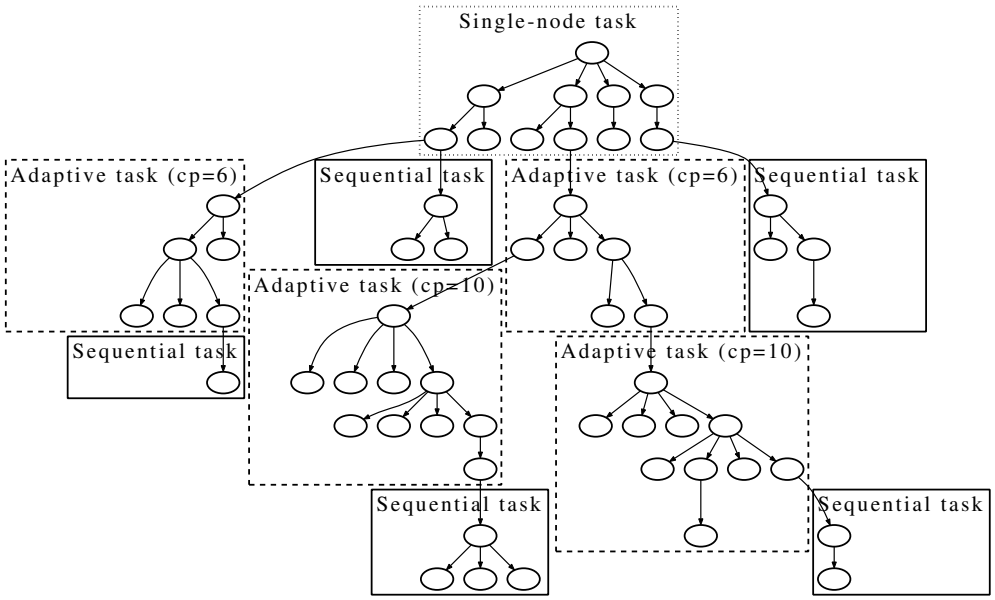


Figure 9.1: Adaptive grain size control in ParSSSE

Number of groups	1	2	4	8	16
Time(sec)	9.7	8.5	8.1	9.4	14.9

Table 9.1: Execution time using different number of groups

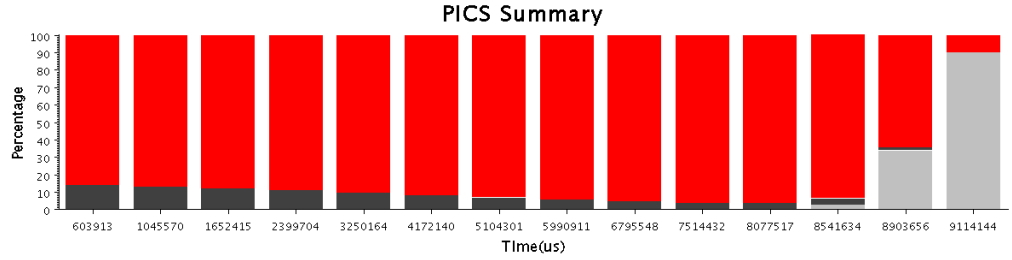
with splitting and creating new tasks. If we split the queue when the amount of execution is too large, there is not enough parallel tasks. As a result the idle time can be high. Therefore, this directly affects the granularity of task execution. We have added a library control point. List 9.1 shows the code to add this control point and how the tuned value is used in the library. Since this control point is added in the ParSSSE library, all state space search applications benefit from this feature without any modification.

Figure 9.2(a) illustrates the change of utilization and overhead percentage during steering. Figure 9.2(b) illustrates the change of the grain size control point. With the steering, the overall utilization increases while the overhead decreases.

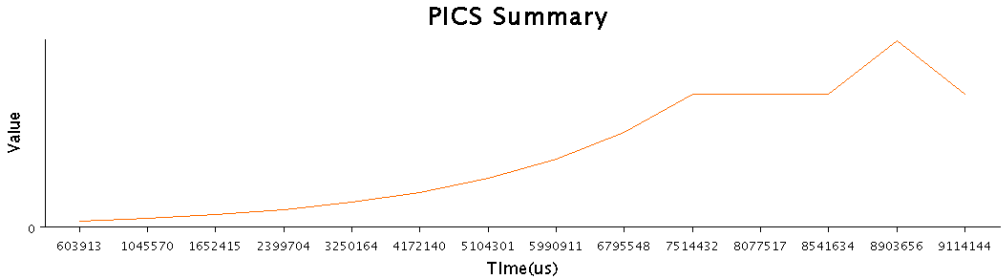
Here, the grain size value does not have to be the same on each processor. Therefore, we have applied parallel evaluation to steer. Table 9.1 compares the total execution time when using different number of groups for parallel evaluation. Figure 9.3(a) and 9.3(b) shows the control point change when using 1 group and 4 groups. It can be clearly seen that when parallel evaluations are used, the optimal value is found much quicker.

Listing 9.1: Control point in ParSSSE

```
PICS_registerTunableParameterFields("ParSSSE_entryGrain", TP_DOUBLE,
    preDefinedEntryGrain, 0.0001, 10, 1.4, PICS_EFF_GRAINSIZE, 1, OP_MUL,
    TS_SIMPLE, 2);
// use the tuned value
double entryGrain = PICS_getTunedParameter("ParSSSE_entryGrain", &valid);
while((state=solver->dequeue()) != NULL)
{
  if(processed_nodes == 20)
  {
    avgentrytime = (CkWallTimer() - instrument_start)/20;
  }
  f2(state, solver, parallel);
  accumulate_time += avgentrytime;
  if(accumulate_time > picsEntryGrain)
  {
    solver-> dequeue_multiple(avgentrytime, processed_nodes);
  }
  processed_nodes++;
}
```

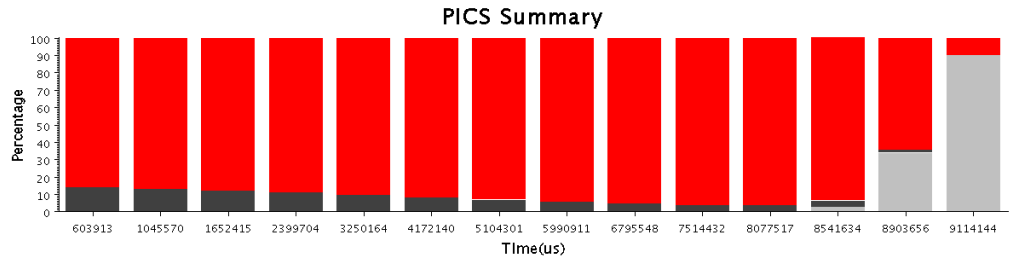


(a) Utilization and Overhead

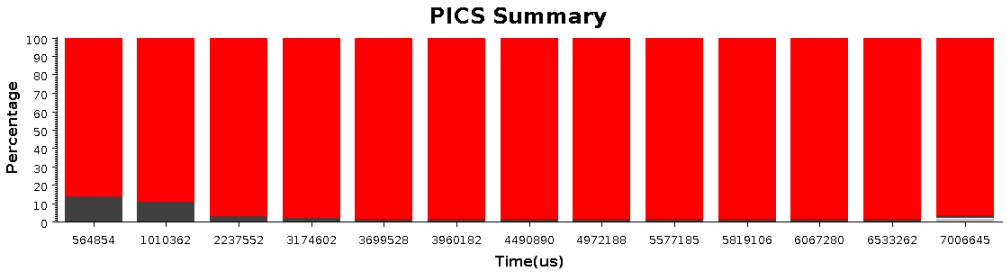


(b) Grain Size

Figure 9.2: Performance variance during Steering



(a) Using 1 group



(b) Using 4 groups

Figure 9.3: Control point steering using 1 group and 4 groups

9.2 Cosmology Application - ChaNGa

Communication bottleneck problem

In some applications, due to the physical requirement or task mapping, some processors get much more communication requests than the others. This is called the communication bottleneck. In order to solve it, one solution is to forward the requests to other processors to evenly distribute the communication. In our system, we generalized a "mirroring" idea to solve this problem. Besides the original tasks and data, we keep several copies of tasks and data. Therefore, when the data is requested, it is forwarded to mirror copies instead of all going to the original tasks. By carefully selecting the task processor mapping and the number of mirrors each task has, we can minimizing the communication deviation on various processors. Depending on the specific applications, how many mirrors each original task should have is a reconfigurable parameter in our system. We apply the PICS control system to find the optimal value for the number of mirrors for different applications.

Mirroring API design

We have designed a minimum set of APIs for application developers to use the mirroring

```
void setMirror(true);

void ArrayElement::syncMirror( CkCallback& cb);

entry [expedited,mirror] void fillRequestParticles( CkCacheRequestMsg<
    KeyType> *msg);

void ArrayElement::unmirrorData(CkMirrorSyncMessage* buffer, int size) { }

char* ArrayElement::mirrorData(int *size) { }

void TreePiece(CkMigrateMessage* m)
{
  useAsMirror = true;
}
```

solution. First, users need to annotate the object to tell that the objects can be mirrored. Secondly, users need to annotate what messages need to be forwarded to the mirror objects. The requirement for these functions are that they only read the data without modification. Thirdly, users need to implement two functions to pack the original data so that the runtime can send them to the mirrors, and to unpack them in the mirrors. Last, the application needs to insert code to call a sync API to synchronize the mirror data with the original objects.

The following shows the APIs for user to apply the mirroring idea. *SetMirror(bool) sets whether the array will be mirrored or not. By default mirroring is not used.* SyncMirror(CkCallback) implies the application to synchronize the data from the original array with their mirrors. The application is blocked until all the mirrors are updated. The callback function will proceed once the synchronization is done. Only the requests to the entry methods with the *mirror attribute are distributed among originals and mirrors. All the other requests to the methods without* mirrorr attribute are only sent to the original array. The *mirrorData() and* unmirrorData() need to be added to tell what and how to pack/unpack data from original array to the mirroring array. In order for the mirroring array not to migrate, we set the *useAsMirror* to be true while the original array to be false.

Design and implementation in Charm++

Here, we have generalized the idea and make the number of mirrors a runtime control point. One particular real application that shows this communication bottleneck problem is

```
class ArrayElement : public CkMigratable
{
 virtual char* mirrorData(int *size);

 virtual void unmirrorData(CkMirrorSyncMessage *buffer, int size);

 virtual void recvSyncMirrorData(CkMirrorSyncMessage *);

 void recvAck();

 void syncMirror(CkCallback& cb);
};

class MirrorUpdate : public CBase_MirrorUpdate
{
};
```

ChaNGa [18], which is a parallel N-Body cosmology simulation application implemented in CHARM++. The problem is found in calculating gravity phases and solved by replicating objects. In this experiment, we show how tuning the number of mirrors improves the performance. Figure 9.4 compares the time cost of calculating gravity without using mirror and with using various number of mirrors. The top red curve is the time cost without using mirror while the bottom green curve shows the cost of using mirrors. The optimal value we found here is to use 2 mirrors.

9.3 Molecular Dynamics Application - NAMD

NAMD [15] is a molecular dynamics application that was developed in the mid-1990's. Unlike its contemporaries at that time, NAMD was designed from scratch to be a parallel program. Over many years, a lot of efforts have been made to scale NAMD to hundreds of thousands cores. My previous work [25, 37] discussed how various manual optimizations helped improve NAMD performance. The improvement is significant but it requires a lot of expert knowledge and tuning efforts. The PICS is applied in NAMD. There are two goals of using PICS in NAMD. The first is to detect performance problems when there are no known control points to tune it. The second is to dynamically steer the applications by adjusting the values of control points, which affect the performance.

Visualize the performance and detect the performance problems

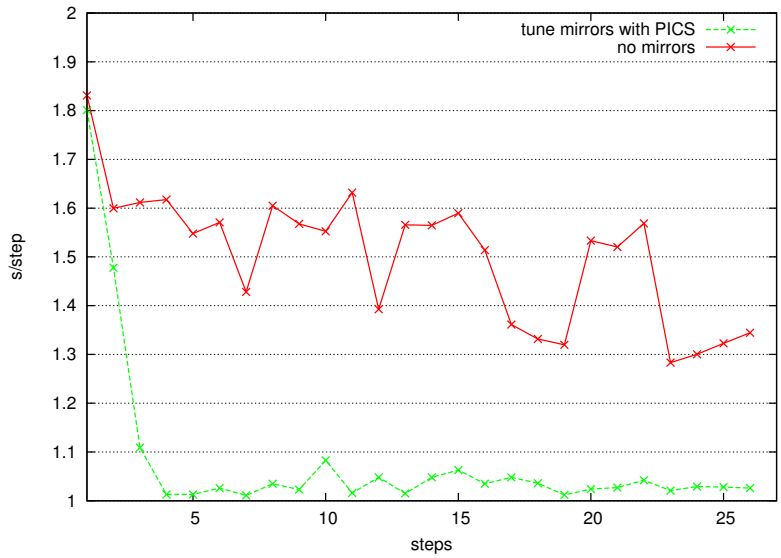


Figure 9.4: Time cost of calculating gravity for various mirrors and no mirror on 16k cores on Blue Gene/Q

Over the years we have been able to scale NAMD to hundreds of thousands cores [38]. At this scale, performance visualization and analysis becomes very challenging due to huge amount of data. For example, running NAMD on 64K cores can easily generate tens of Gigabytes tracing data. It takes both space to store the data and time to transfer data among computers. Even worse, it is almost impossible to figure out the performance problems by visualizing the data manually. Here, instead of generating detail tracing data, the performance summary data generated by PICS can be used to have an overview of the performance. The high level performance data including idle percentage, utilization percentage, communication are all included in the summary. The summary data can be visualized in Projections as described in section 4.4.

Secondly, as it is described in chapter 4, the decision tree based automatic performance analysis helps detect NAMD performance problems and provide possible solutions. During the execution the PICS has performed the automatic analysis. The results of analysis are recorded in the files, which can be read by developers or visualization tools. List 9.2 shows the results of analysis when running NAMD to simulate DHFR system using 512 cores on Blue Gene/Q Vesta. From the list, it can be seen that there are a few performance problems, including high overhead, load imbalance, communication imbalance. The possible solutions

Listing 9.2: Results of PICS performance analysis on BGQ

```
Condition High_Overhead 2 -1 AVG_OverheadPercentage 0.213-0.08>0.0 0.213
Condition High_Idle 1 -1 AVG_IdlePercentage 0.132-0.080>0.0 0.132
Condition Load_Imbalance 36 8 MAX_LoadPerPE/AVG_LoadPerPE> 1.10 0.13 70
Condition FEW_ENTRIES 5 8 AVG_EntryMethodDuration-AVG_LoadPerPE< 0.0 0.132
Solution Down PICS_EFF_GRAINSIZE
Solution UP PICS_EFF_LDBFREQUENCY
Condition Few_Obj_Per_PE 9 -1 AVG_NumObjectsPerPE 0.25-3.0<0.0 0.13
Condition Large_Bytes_Per_Obj 40 20 MAX_BytesPerObject/AVG_NumMsgRecv >
    1.2 0.13 0
Condition Load_Imbalance 36 8 MAX_LoadPerPE/AVG_LoadPerPE> 1.1 0.13 70
Condition Comm_Imbalance 50 20 MAX_NumMsgRecv/AVG_NumMsgRecv > 1.5 0.0 95
Solution Down PICS_EFF_GRAINSIZE
Solution UP PICS_EFF_REPLICA
Solution Down PICS_EFF_GRAINSIZE
```

Listing 9.3: Decomposition control point in NAMD

```
PICS_registerTunableParameterFields("DECOMP_AWAY",
             TP_INT, default_decomp, 0, 3, 1,
             PICS_EFF_GRAINSIZE_1, -1, OP_ADD, TS_SIMPLE, 1, "WorkDistrib"
                 );
```

include decreasing the grain size, increasing load balancing frequency, or using replicas for communication imbalance.

Steering NAMD with control points

Based on our previous study and experience, we have found that the performance highly depends on the setting of various configurations. The following control points are extracted and used to optimize performance.

1. Decomposition degree: including 1-Away, 2-Away X, 2-Away XY, 2-Away XYZ. In section 4.4 we have presented the performance of running same simulation with 1-Away decomposition and 2-Away decompositions. That example illustrates that properly chosen decomposition is required to obtain the best performance. We have make this as a control point, which has four possible values. The effect of this control point is the grain size. The code to register this control point is shown in List 9.3.

2. The number of PME pencils used in long range calculation. As described in the background, the long range calculation in NAMD is implemented using pencil decomposition of 3D FFT. When there is too few pencils, execution time of each entry

Listing 9.4: PME pencil control point in NAMD

```
PICS_registerTunableParameterFields("FFT_PENCIL_NUM",
              TP_INT, 1, 1, 2, 1, PICS
              _EFF_GRAINSIZE_2, -1, OP_ADD, TS_SIMPLE, 1, "PmePencil");
```

method might be too large. This might lead to heavy load on the processors with PME pencils. However, if the number of pencils is large, each pencil has small amount of computation, small amount of communication. This leads to small messages, small entry execution. This leads to high computation overhead and high communication latency. Therefore, depending on the molecular systems, the number of processor cores the system is running with, the number of FFT pencils should be adaptive. We have added this as a control point. The registration for it is shown in List 9.4.

In the above, both control points have the effects on the grain size. However, the associated objects they affect are different. Therefore, the name of the objects is added to tell the object they have impact on.

3. The compression related control points. Besides the complexity in compression itself, NAMD has its own challenge. There are two major kinds of messages in NAMD, short-range messages among neighbors and long-range messages over all the processors. Since short-range messages communicate with neighbors, they do not cause global network contention. Therefore, it might not benefit from compression. On the other hand, long-range messages might need to be compression to reduce the network contention. Therefore, we need to distinguish these two types of messages.

4. The load balancing frequency. Load balancing in NAMD can significantly impact the performance. When there is high load imbalance, load balancing should be performed to improve the performance. However, if it is done too frequently, the overhead of load balancing might over weigh the benefit of it. Also when running on large number of cores, load balancing overhead becomes worse. Depending on the degree of load imbalance in the system, the overhead of load balancing, the benefit of load balancing, load balancing frequency should be adaptive and dynamically tuned during execution. We have added this as a control point.

Listing 9.5: Load balancing frequency control point in NAMD

```
PICS_registerTunableParameterFields("LDB_FREQ",
            TP_INT, 400, 40, 10000, 1,
            PICS_EFF_LDBFREQUENCY, -1, OP_ADD, TS_SIMPLE, 1);
```

Number of cores	8	128	2048	8192
run 1 (ms/step)	444	42.2	17.7	19.5
run 2 (ms/step)	444	39.8	10.2	9.8
run 3 (ms/step)	444	40.2	6.9	6.5
run 4 (ms/step)	444	40.2	6.9	4.9

Table 9.2: Time step of simulating DHFR using different number of cores with PICS steering

Once we have registered the above control points, the next step is to reconfigure the applications to use the new values from PICS. For the control points of decomposition method, number of PME pencils, it is not easy to directly use the values. This is mainly because all the data structures depend on their values. The data structures are constructed at the beginning so they are difficult to change during execution. Although in theory we can destruct all the data and re-construct during execution, it costs too much programming efforts to write this part of code. Instead, we come out with an easier way. The simulation is run for small number of steps and PICS steering is applied for these steps. Then the run finishes and the results of PICS are used to modify the configuration file for NAMD simulation. The program is restarted with the new configurations in the simulation file. Table 9.2 shows the time step of simulating DHFR using different number of cores. For each set of processor number, we did four consecutive run. The first run starts with using 1 Away decomposition. And then the following run uses the tuned configurations from the previous run. It can be seen that on different number of processor cores, the time step decreases. Examining the configurations that are used, 1 Away decomposition is the configuration for the best performance. 2 Away X is best for the run using 128 cores. For 8192 cores run, 2 Away X Y Z is the best configuration. PICS is able to find all the best configurations regardless of the starting configuration.

The listing 9.6 shows the analysis result of simulating DHFR on 2048 cores with 1Away and PME being 32. It can be seen that the idle percentage and overhead is high. The high idle percentage is caused by the load imbalancing and long function execution of short range

Listing 9.6: Performance analysis of NAMD

```
Condition High_Overhead 2 -1 AVG_OverheadPercentage-0.070>0.0
Condition High_Idle 1 -1 AVG_IdlePercentage-0.10>0.000
Condition Long_Entry_1 60 12 MAX_EntryMethodDuration_1/AVG_LoadPerPE>1.20
Condition Load_Imbalance 40 12 MAX_LoadPerPE/AVG_LoadPerPE 0.0058>1.10
Solution UP PICS_EFF_LDBFREQUENCY
Solution Down PICS_EFF_GRAINSIZE_1
Condition Small_Entry_2 7 -1 AVG_EntryMethodDuration_2-0.00006<0.00
Solution UP PICS_EFF_GRAINSIZE_2
```

calculation. The high overhead percentage is caused by the fine grain PME calculation. The solutions for this are to decrease the grain size of short range calculation and to increase to the grain size of PME calculation. This example demonstrates that we are able to adjust the grain size differently for multiple objects.

Chapter 10

Related Work

Autonomic computing and adaptive systems have been proposed as one method to deal with the rising complexity of computer systems [39–41]. Adaptive techniques have been built to provide performance in web servers [42].

10.1 Compiler Autotuning on Multicore Architectures

Traditionally compiler based autotuning is the most common approach to optimize the parallel program performance. Such systems generate in parallel a set of alternative implementations of code. These implementations run in parallel and are evaluated to select the best one. The selection is either done manually or automatically. The examples of such systems include PERI [43], POET [44], SPIRAL [45], FFTW [46], ATLAS [47], and PHiPAC [48]. These system work with single node multicore execution.

PERI [43] studies a set of multicore optimizations for three programs of Sparse Matrix Vector Multiplication (SpMV), the explicit heat equation PDE on a regular grid (Stencil), and a lattice Boltzmann application (LBMHD). It then employs a code generate to produce multiple versions of the computational kernels using a set of optimizations with varying parameter settings. These implementation are tested to figure out the optimal version of code on the platform. They have demonstrated that auto-tuning is essential in achieving good performance.

Stencil computation is the kernel code for many scientific applications. Therefore, its

optimization and auto-tuning on multicore architectures has been widely studied by many researchers [49–53]. The authors in [49] described various optimizations including hierarchical blocking, unrolling, reordering, and prefetching, etc. Autotuning is applied to choose the best combinations of parameters. They believe the application-specific auto-tuners are the most practical near-term approach for obtaining high performance on multicore systems. Overall, a lot of progress is achieved for stencil code optimization.

10.2 Autotuning in Distributed System

Now there has been more and more work studying performance tuning and optimization for the large scale distributed systems and supercomputers. Due to the complexities of the hardware system and applications, so far no unified solution is found to be best. Each approach has its advantage and limitation.

Performance tuning has been intensively studied for communication libraries like MPI. Authors [54] have well tuned the MPI library implementation over InifiniBand network. Due to the importance of MPI collectives, more research has been done to optimize the collectives. In paper [11], Thakur and Gropp reports their work on improving the performance of collective operations in MPICH on clusters connected by switched networks. For each collective operation, multiple algorithms are discussed depending on the message size, with the goal of minimizing latency for short messages and minimizing bandwidth usage for long messages. Their results show that the new algorithms significantly outperform the old algorithms. There are other work about MPI collectives optimization [55–59].

Besides MPI tuning, in the high performance computing areas, there has been project focusing on adaptive optimizations for various applications. Three most important ones of these projects are Autopilot [60], Active Harmony [61] and MATE [62].

Autopilot is a system that gathers performance data for grid applications through sensors, either accessing program variables directly 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. In Autopilot, the sensors and actuators used by Autopilot are written specifically for each application. However, in

our control point system, the concept and APIs are general purpose for both applications and the runtime system.

Active Harmony allows parallel programs to expose a list of integer tunable parameters. The parameters can be tuned across multiple runs [61] or in an online manner [63]. The tuning algorithms used in Active Harmony include various direct search methods such as Nelder-Mead Simplex and a new algorithm called Parallel Rank Ordering [63]. Our PICS focuses on steering by analysis and the effects of control points while Active Harmony focuses on the optimization methods.

MATE tunes the parallel/distributed applications by monitoring, analysis, and tuning the environments. It does either automatic tuning for the libraries or dynamic performance tuning for applications. For application tuning, it explicitly asks users to define the performance models, which can be hard for real applications. Our PICS does not require this so as to reduce the burden of programmers.

Nowadays with power and energy becoming concern in HPC areas, researchers start to automatically control or reduce the energy cost by adaptive methods. The SEEC - a General and Extensible Framework for Self-Aware Computing project [41, 64] proposes a novel approach that is capable of addressing the power and energy constraints by adding a self-awareness dimension to computational models. Given a set of goals and actions, SEEC uses analysis and decision engines (e.g., adaptive feedback control systems and machine learning) to monitor application progress and select actions to meet goals optimally (e.g. meeting performance goals with minimal power consumption).

Chapter 11

Conclusion Remarks

This thesis aims at improving both the parallel program performance and productivity. It proposes a design of an introspective control system that supports dynamic performance steering. The application developers provide tuning knobs and related knowledge to guide the control system. The tuning knobs are represented as control points, which are the interface between the control system and the applications or runtime system. Meanwhile, the control system collects the program behaviors, analyzes the performance and dynamically steer the program execution. Expertise knowledge rules based performance analysis are applied to accelerate the process of automatic performance steering at runtime. This thesis also investigates how the parallel runtime system and applications can become more adaptive and controllable by exposing abstractions to the introspective control system. In order to make approach more feasible in practice, scalable techniques are developed so as to handle large-scale run analysis and steering.

The ideas and techniques are implemented in the PICS framework - Performance-analysis-based Introspective Control System. It is based on the Charm++ runtime system. Different control points have been added to several synthetic benchmarks and real-world applications. The relationships between these control point values and the resulting program performance and measurable effects are discussed. For all control points, it is shown that it is often possible to determine the correct direction to turn each knob to improve performance by examining various types of measurements.

There still remain more categories of control points to be examined. It is still an open problem to determine the most effective and general purpose tuning scheme for large applications with many control points. To more fully address all of these issues, more control points should be added to more applications. The costs of benefits of various tuning schemes will be further analyzed in the future.

As computers are moving towards more complex, larger parallel systems, and with Exascale computing ahead, we believe that automatic tuning of parallel programs will become necessary. Our work on control points investigates one such avenue for dynamically reconfiguring applications and the runtime system. Many open questions exist about the generality, costs, and benefits of automatic tuning, and whether automatic tuning will eliminate some of the need for human experts in developing complex applications. The answers to these questions will likely influence the designs of future parallel programming languages, runtime systems, and even the architectures of machines.

Chapter 12

References

[1] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[3] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. Mpi-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[4] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[5] Gasnet: A portable high-performance communication layer for global address-space languages, 2002.

[6] 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.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[9] 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*, pages 519–538, New York, NY, USA, 2005. ACM.

[10] Yanhua Sun, Jonathan Lifflander, and L. V. Kale. PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications. In *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*, Munich, Germany, June 2014.

[11] Rajeev Thakur and William D. Gropp. Improving the Performance of Collective Operations in MPICH. *Lecture Notes in Computer Science*, 2840:257–267, October 2003.

[12] Osman Sarood, Phil Miller, Ehsan Totoni, and L. V. Kale. 'Cool' Load Balancing for High Performance Computing Data Centers. In *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[13] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[14] Isaac Dooley. *Intelligent Runtime Tuning of Parallel Applications With Control Points*. PhD thesis, Dept. of Computer Science, University of Illinois, 2010. http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

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

[16] Cliff Young, Joseph A. Bank, Ron O. Dror, J. P. Grossman, John K. Salmon, and David E. Shaw. A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[17] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 1986.

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

[19] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010. IEEE Computer Society.

[20] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato. Adaptive Techniques for Clustered N-Body Cosmological Simulations. *ArXiv e-prints*, September 2014.

[21] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Lecture Notes in Computer Sciences*, volume 4382, pages 235–250. Springer-Verlag, 2007.

[22] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10. ACM, 2011.

[23] Sameer Kumar, Yanhua Sun, and L. V. Kale. Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q. In *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, May 2013.

[24] Yanhua Sun, Gengbin Zheng, L. V. Kale, Terry R. Jones, and Ryan Olson. A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect. In *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[25] Yanhua Sun, Gengbin Zheng, Chao Mei Eric J. Bohm, Terry Jones, Laxmikant V. Kalé, and James C.Phillips. Optimizing fine-grained communication in a biomolecular simulation application on cray xk6. In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, Salt Lake City, Utah, November 2012.

[26] Wikipedia. Control theory. `http://en.wikipedia.org/wiki/Controltheory`, 2013.

[27] David B Skalak. Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Proceedings of the eleventh international conference on machine learning*, pages 293–301, 1994.

[28] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[29] Abhinav Bhatele and Laxmikant V. Kale. Application-specific Topology-aware Mapping for Three Dimensional Topologies. In *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS '08)*, April 2008.

[30] Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé. A Case Study of Communication Optimizations on 3D Mesh Interconnects. In *Euro-Par 2009, LNCS 5704*, pages 1015–1028, 2009.

[31] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *23rd ACM International Conference on Supercomputing*, 2009.

[32] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[33] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kale. An Adaptive Framework for Large-scale State Space Search. In *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[34] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kalé. ParSSSE: An Adaptive Parallel State Space Search Engine. *Parallel Processing Letters*, 21(3):319–338, September 2011.

[35] Lukasz Wesolowski, Ramprasad Venkataraman, Abhishek Gupta, Jae-Seung Yeom, Keith Bisset, Yanhua Sun, Pritish Jetley, Thomas R. Quinn, and Laxmikant V. Kale. TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages. In *Proceedings of the International Conference on Parallel Processing*, ICPP '14, Minneapolis, MN, September 2014.

[36] Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, 1999.

[37] Chao Mei, Yanhua Sun, Gengbin Zheng, Eric J. Bohm, Laxmikant V. Kalé, James C.Phillips, and Chris Harrison. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[38] James C. Phillips, Yanhua Sun, Nikhil Jain, Eric J. Bohm, and Laximant V. Kale. Mapping to irregular torus topologies and other techniques for petascale biomolecular simulation. In *Proceedings of ACM/IEEE SC 2014*, New Orleans, Louisiana, November 2014.

[39] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[40] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM.

[41] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[42] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. *SIGOPS Oper. Syst. Rev.*, 41(3):289–302, March 2007.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[51] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[52] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164. ACM, 2012.

[53] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science-Research and Development*, 26(3-4):205–210, 2011.

[54] 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.

[55] Jelena Pjesivac-Grbovic, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack Dongarra. Decision trees and mpi collective algorithm selection problem. In *Euro-Par*, pages 107–117, 2007.

[56] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, and John Gunnels. Mpi collective communications on the blue gene/p supercomputer: Algorithms and optimizations. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pages 63–72. IEEE, 2009.

[57] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377–384. IEEE, 2000.

[58] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[59] Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of mpi collectives on clusters of large-scale smp's. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

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

[61] 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.

[62] 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.

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

[64] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *SIGPLAN Not.*, 46(3):199–212, March 2011.

Appendix A

Rules for Decision Trees

Listing A.1: Certainty Tree

```
-1 Root
0 CPU_Util AVG UtilizationPercentage SUB -1 0.94 GT 1 Root
0 High_Overhead AVG OverheadPercentage SUB -1 0.08 GT 1 Root
0 High_Idle AVG IdlePercentage SUB -1 0.08 GT 1 Root
0 Small_Message AVG BytesPerMsg SUB -1 300 LT 1 High_Overhead
0 High_CacheMissRate AVG CacheMissRate SUB -1 0.1 GT 1 CPU_Util
0 LOW_CacheMissRate AVG CacheMissRate SUB -1 0.1 LT 1 CPU_Util
0 Small_Entry AVG EntryMethodDuration SUB -1 0.00006 LT 1 High_Overhead
0 Long_Entry MAX EntryMethodDuration DIV 0 AVG LoadPerPE GT 1.2 1
    High_Idle
0 Long_Object MAX LoadPerObject DIV 0 AVG LoadPerPE GT 1.2 1 High_Idle
0 Load_Imbalance MAX LoadPerPE DIV 0 AVG LoadPerPE GT 1.1 1 High_Idle
0 Long_Critical_Path MAX CriticalPathLength DIV 0 AVG LoadPerPE GT 1.2 1
    High_Idle
0 FEW_INVOC AVG NumInvocations SUB -1 1 LT 1 High_Idle
0 FEW_ENTRIES AVG EntryMethodDuration SUB 0 AVG LoadPerPE LT 0 1 High_Idle
1 Leaf_Up_Grainsize UP PICS_EFF_GRAINSIZE 1 Small_Entry
1 Leaf_Down_Grainsize DOWN PICS_EFF_GRAINSIZE 5 Long_Entry Long_Object
      Long_Critical_Path FEW_ENTRIES FEW_INVOC
1 Leaf_Up_Aggregation UP PICS_EFF_AGGREGATION 1 Small_Message
1 Leaf_Up_LDB_Frequency UP PICS_EFF_LDBFREQUENCY 1 Load_Imbalance
1 Leaf_Up_Msgsize UP PICS_EFF_MESSAGESIZE 1 Small_Message
```

Listing A.2: Fuzzy Tree

```
#inter-leaf key avg-min-max key_in_int OP flag value/index symbol parent
-1 Root
0 CPU_Util AVG UtilizationPercentage SUB -1 0.94 GT 1 Root
0 High_Overhead AVG OverheadPercentage SUB -1 0.08 GT 1 Root
0 High_Idle AVG IdlePercentage SUB -1 0.08 GT 1 Root
0 Small_Message AVG BytesPerMsg SUB -1 300 LT 1 High_Overhead
0 High_CacheMissRate AVG CacheMissRate SUB -1 0.1 GT 1 CPU_Util
0 LOW_CacheMissRate AVG CacheMissRate SUB -1 0.1 LT 1 CPU_Util
0 Few_Obj_Per_PE AVG NumObjectsPerPE SUB -1 3 LT 1 High_Idle
0 Large_Bytes_Per_Obj MAX BytesPerObject DIV 0 AVG NumMsgRecv GT 1.2 1
    High_Idle
0 Large_Bytes_PerMsg AVG BytesPerMsg SUB -1 100000 GT 1 High_Idle
0 Load_Imbalance MAX LoadPerPE DIV 0 AVG LoadPerPE GT 1.1 1 High_Idle
0 Comm_Imbalance MAX NumMsgRecv DIV 0 AVG NumMsgRecv GT 1.5 1 High_Idle
0 Much_External_Comm AVG ExternalBytePerPE DIV 0 AVG BytesMsgRecv GT 0.8 2
     High_Idle High_Overhead
1 Leaf_Up_Grainsize UP PICS_EFF_GRAINSIZE 2 Small_Message High_Overhead
1 Leaf_Down_Grainsize DOWN PICS_EFF_GRAINSIZE 3 Few_Obj_Per_PE
      High_CacheMissRate Load_Imbalance
1 Leaf_Down_LDB_Frequency DOWN PICS_EFF_LDBFREQUENCY 1 High_Overhead
1 Leaf_Up_Nodesize UP PICS_EFF_NODESIZE 1 Much_External_Comm
1 Leaf_Down_Aggregation DOWN PICS_EFF_AGGREGATION 1 Large_Bytes_PerMsg
1 Leaf_Down_Msgsize DOWN PICS_EFF_MESSAGESIZE 1 Large_Bytes_PerMsg
1 Leaf_Up_Compression UP PICS_EFF_COMPRESSION 1 Large_Bytes_PerMsg
1 Leaf_Perf_Good UP PICS_EFF_PERFGOOD 1 LOW_CacheMissRate
1 Leaf_Up_Replica UP PICS_EFF_REPLICA 1 Large_Bytes_Per_Obj
```