# PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications

Yanhua Sun, Jonathan Lifflander, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Ave
Urbana, IL, 61801, USA
{sun51, jliffl2, kale}@illinois.edu

## ABSTRACT

Parallel programming has always been difficult due to the complexity of hardware and the diversity of applications. Although significant progress has been achieved with the remarkable efforts of researchers in academia and industry, attaining high parallel efficiency on large supercomputers with millions of cores for various applications remains challenging. Therefore, performance tuning has become even more important and challenging than ever before. In this paper, we describe the design and implementation of *PICS*: Performance-analysis-based Introspective Control System, which is used to tune parallel programs. PICS provides a generic set of abstractions to the applications to expose the application-specific knowledge to the runtime system. The abstractions are called *control points*, which are tunable parameters affecting application performance. The application behaviors are observed, measured and automatically analyzed by the PICS. Based on the analysis results and expert knowledge rules, program characteristics are extracted to assist the search for optimal configurations of the control points. We have implemented the PICS control system in Charm++, an asynchronous message-driven parallel programming model. We demonstrate the utility of PICS with several benchmarks and a real-world application and show its effectiveness.

## 1. INTRODUCTION

Modern parallel computer systems are becoming extremely complex due to complicated network topologies, hierarchical storage systems, heterogeneous processing units, etc. Physics simulation models have also been expanded to match experimental results, which greatly increases the complexity of computation. Although computer scientists focus on parallel models and implementation while physical scientists concentrate on physics models, their goal is similar in that they both seek to maximize application performance on the available resources.


*ROSS* '14, June 10, 2014, Munich, Germany


In the past, the ideal vision for parallel programming was to let the compiler automatically generate an efficient parallel program without involving effort from the developers. Over the years, this has been proven to be impractical due to the high complexity of compilers, the diversity of parallel programs, and the resulting low parallel efficiency of generated code. The most popular parallel programming model today is MPI (Message Passing Interface) [1], which automates very little. The major ongoing work tries to overcome this drawback by providing more features in programming models and underlying runtime systems to reduce the burden on programmers. Some examples include Partitioned Global Address Space (PGAS) (supported by the GASNET runtime [2]), which increases productivity by giving the programmer a global view of data and automatically optimizing communication. Cilk is designed as an efficient multithreaded runtime system by utilizing a provably-good work stealing scheduler with performance guarantees [3]. The new popular languages of Chapel [4] and X10 [5] also have powerful runtime systems.

Instead of full automation or completely manual tuning, we take the approach of almost-automation with the assistance of application developers. Many applications can be re-configured in ways that affect performance. The goal of this work is to allow the runtime system to adjust the configuration automatically based on application-specific knowledge and runtime observations. First, developers must provide the application-specific knowledge to a runtime system. The knowledge mainly includes what parameters can be tuned, what structure the application has, and how the application responds to different configurations. Second, the runtime system observes the application behavior, analyzes it, and orchestrates the reconfiguration. The knowledge of both the underlying architecture and the possible configurations of the application empowers the runtime system to automate the process of tuning applications.

Some existing parallel systems such as Charm++ [6] already observe characteristics of a parallel program's execution in order to perform dynamic load balancing [7, 8], but very few general mechanisms exist for the runtime system to control other behaviors of the application. In this paper, we have proposed the control-point centric mechanism to optimize application performance. In order to distinguish this mechanism from the previous adaptive load balancing scheme, we call it the control system. The application interacts with the control system by defining control points, first proposed in Dooley's PhD thesis [9]. Control points are

tunable parameters in an application that are made available to the control system along with information about the expected effects of changing the parameter. The supplemental information about the effects of adjusting a control point enables the control system to quickly determine which control points have the potential to fix an observed performance problem. The control system monitors the application events, collects them and performs automatic analysis to detect performance bottlenecks. Based on the performance results and a set of expert knowledge rules, the control system makes decisions about what control points to steer and how to steer them to improve the overall performance. The new configuration for each control point is then fed back into the application or the runtime system itself based on the decision made.

We have made the following contributions in this paper:

1. We propose control points as an interface for the applications to interact with the runtime system.
2. We describe how automatic performance analysis based on decision trees can be used efficiently to determine which control points should be tuned.
3. We demonstrate how PICS can be applied into both the runtime system and applications to optimize the application performance. Our results show its effectiveness for ChaNGa, a full-fledged cosmology application, on 16,384 cores.

In the rest of this paper, we first describe the performance-analysis-based introspective control system, explain the notion of control points, and discuss the analysis process used to speed up the performance tuning. The design and implementation is also discussed, followed by the experimental results to show the utility and effectiveness of PICS on both synthetic benchmarks and real applications.

# 2. OVERVIEW OF PICS CONTROL SYSTEM

Figure 1 shows the infrastructure of PICS, a performance-analysis-based introspective control system. 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 expose 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 configuration. 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

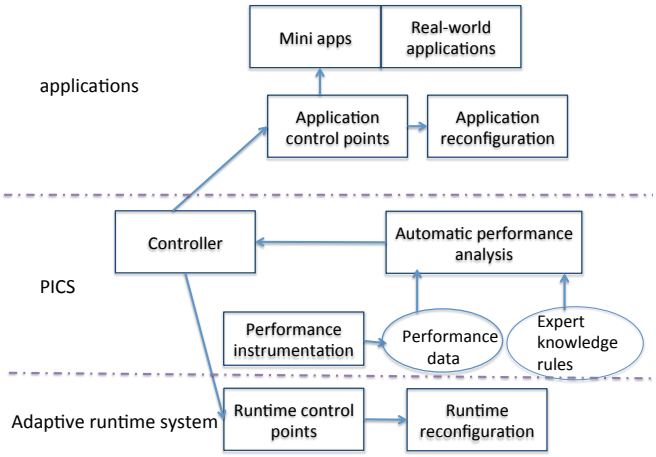



**Figure 1: Research diagram**

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. Next, we will explain the control points and describe performance-analysis-based steering.

## 2.1 Control points

*Control points* are tunable parameters that are used by the control system to interact with the application and the runtime system. Control points are registered by the applications or the runtime system with special properties. First, the values of a control point should be adjustable. The application or runtime system should be written in a way that adapts to their different values. Secondly and most importantly, the control points have some *effects* on the performance. Effects are the intermediate ramification of a control point that impacts the overall performance. Each control point is also associated with the direction of the effect, which means changing the control point will have impact on the effect either in a negative or a positive way. For example, increasing the sub-problem size in Jacobi will increase the grainsize of each task. We have found that the following categories cover most of the effects of control points.

**Degree of Parallelism** This is anything that affects the number of parallel tasks. For example, given a fixed problem size of a Jacobi program, the number of sub-problems affects the degree of parallelism. In molecular dynamics, the number of spatially decomposed chunks of particles affects the parallelism.

**Grainsize** This is the amount of computation for a parallel task. This effect is inversely proportional to the degree of parallelism.

**Priority** To maximize the performance, tasks will often be scheduled with different priorities. Ideally, tasks that are on the critical path should be scheduled earlier. This is achieved by setting their priority higher. Priority can be associated with message sending, message receiving, and task execution.

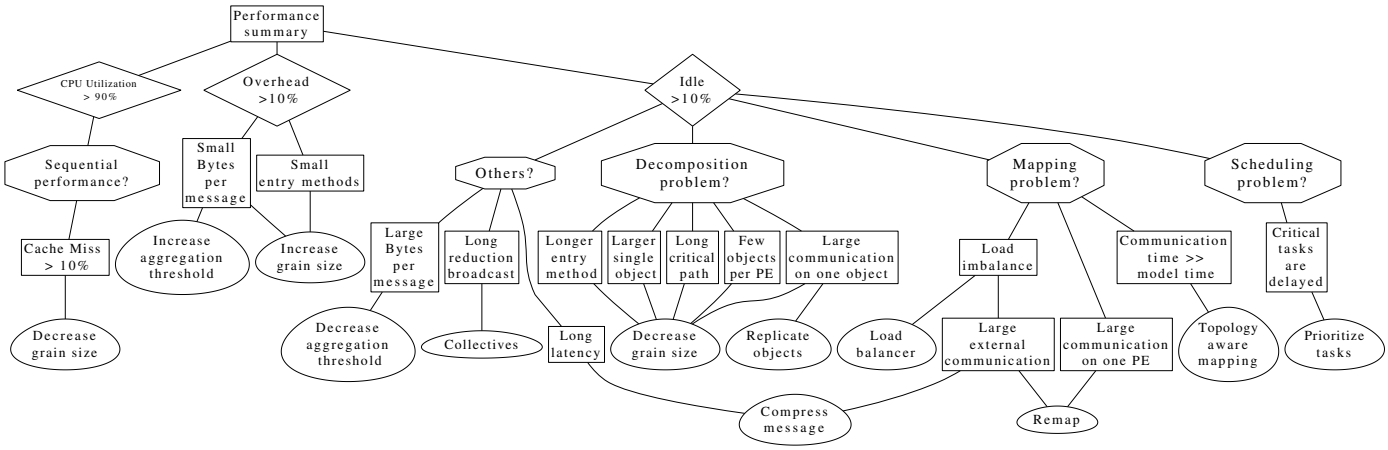



**Figure 2: Performance Analysis Decision Tree**

**Memory Consumption** Scheduling tasks in different orders can often impact the memory usage of the application and/or system, which can have performance ramifications.

**Cache Miss Rate** Often adjusting a knob will have some impact in the cache miss rate. For example, making the grainsize larger might increase cache misses and thereby decrease performance.

**Overhead** This is anything related to the cost of running the program, which is not a part of the computation in the application.

**Number of Messages and Message Size** Proper message size can both better utilize network bandwidth and also overlap computation with communication. For example, message aggregation by the runtime can be important effects.

## 2.2 Performance-Analysis-Guided Steering with Control Points

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) 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 tune 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.

## 2.3 Categories of Performance Problems

In order to determine application performance deficiencies and then possible solutions, we need to identify the characteristics of the program. We categorize the program characteristics and problems into three main types: decomposition, task mapping, and scheduling.

*Problem decomposition* is how a problem is decomposed into smaller problems, which can be solved in parallel with the appropriate dependencies. Problem decomposition directly determines the grain size of the computation and communication and the degree of parallelism. Effective problem decomposition is essential to achieve high performance. The specific characteristics related to the problem of decomposition are shown in the Figure 2. When these characteristics are identified, it signals a potential grain size problem.

*Task mapping* is how tasks are mapped to physical processors. Task mapping affects the communication cost. There is significant related work on how task mapping impacts overall performance, including topology-aware mapping [10]. Task mapping also affects the load balance. In addition, it may also affect memory usage and I/O usage. The characteristics related to task mapping are illustrated the Figure 2. The corresponding solutions range from performing topology-aware mapping, communication-aware load balancing, or compressing messages.

*Scheduling* is about the order in which the runtime executes available tasks on processors. The main ramification of deficient scheduling is that critical tasks may be delayed, causing processors that depend on the critical tasks to become idle. The other potential problem caused by scheduling is running out of memory. If only the tasks that consume memory are scheduled while the tasks that free the memory are not scheduled, the program may cause an out of memory error.

We represent the program characteristics and corresponding solutions in the complete decision tree shown in Figure 2. In this figure, starting from the performance summary data, the decisions are made based on the performance characteristics and the specific performance data collected from an execution. The three diamonds represent the course-grained

performance metrics. Under each diamond, we check if any corresponding performance characteristic exists. The performance characteristic is shown in boxes. When a characteristic is matched, the corresponding performance solution is proposed at the leaves of the tree, shown by oval shapes. The corresponding performance solution has two implications. The first is the aspect of the applications that requires steering. The other is the direction of the steering required to fix the deficiency. For example, if the solution is to decrease the grain size, we must adjust control points whose effect is related to the grain size.

Therefore, the process of performance analysis is performed by traversing a decision tree. Whenever a leaf node is found, it is saved for tuning. As a result, we have a list of performance solutions after a full traversal. We feed these effects into the control points database to determine what control points to tune and in which direction. Based on current values, the tuning directions and movement unit to adjust, the new values for control points are determined.

# 3. CONTROL SYSTEM IMPLEMENTATION IN CHARM++

To investigate the appropriate mechanisms required to add control points to parallel applications, an initial control system framework has been created 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.

## 3.1 PICS Framework in CHARM++

The idea of our work can be applied to most parallel runtime systems. In this paper, our design and techniques are based on the CHARM++ runtime system. 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 overdecompose 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.

## 3.2 Control Point API

In our framework, a control point has the structure as shown in Listing 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 2. 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.

```
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 1: Struct of Control Point**

Besides registering control points, application users also need to tell the control system about the pattern of the application. In this work, we have focused on scientific simulation applications. These applications are generally composed 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 this purpose is `startStep` and `endStep`. For applications that contain multiple phases for one step, `startPhase` and `endPhase` are provided to mark these phases. The API is shown in Listing 2.

```
void startStep();
void endStep();

void startPhase(int phaseId);
void endPhase();
```

**Listing 2: APIs for steps and phases**

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:

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

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.

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

# 4. RESULTS OF CONTROL POINTS IN APPLICATIONS

To illustrate the utility of PICS and how PICS can be used to intelligently optimize parallel applications, this section presents the results of two synthetic benchmarks, a stencil kernel code and a real application, ChaNGa. We have abstracted and added multiple control points. Some are added in the applications while others are in the runtime system. Applications with their specific control points are affected by both the runtime control points and their own application control points. Applications without their own control points are only affected by the runtime control points. In this paper, we only present a few control points, but our full-fledged control system framework will have much more control points.

All the experiments were performed on two systems: IBM Blue Gene/Q 'Vesta' at Argonne National Laboratory and the Cray XE6 'JYC' at National Center for Supercomputing Applications. Vesta has 2048 nodes, each of which consists of one 1.6 GHz PowerPC A2 processor with 16 cores supporting 4-way simultaneous multithreading and 16 GB DDR3 memory. JYC has 64 nodes of thirty-two 2.2GHz AMD 'Bulldozer' processor and 64GB DDR3 memory. The PAMI (Parallel Active Messaging Interface) machine layer in CHARM++ [11] was used on Vesta and uGNI (user Generic Network Interface) [12, 13] was used on JYC.

## 4.1 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

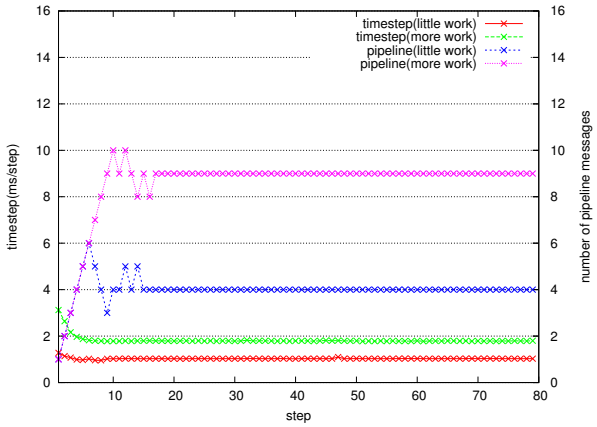


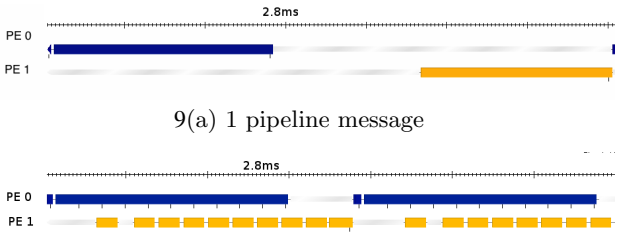

**Figure 3: Tuning the number of pipeline messages to optimize performance**

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.

Figure 3 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 4 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 the run.

## 4.2 Message Compression

Compressing communication data reduces the network load and accelerates the communication, possibly improving the performance. However, whether it really benefits the per-




9(a) 1 pipeline message

9(b) 10 pipeline messages

**Figure 4: Timeline of pipeline transferring using 1 message and 10 messages**

formance 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.

Figure 5 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.

## 4.3 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,

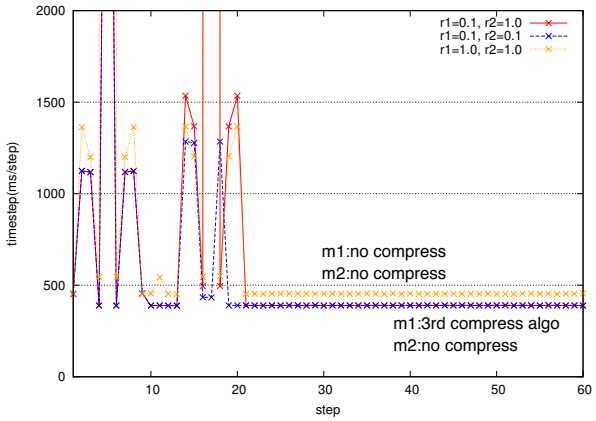



**Figure 5: Steering the compression algorithm for all-to-all benchmark**

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 6 illustrates how PICS steers Jacobi3D in determining the best sub-block sizes. The test is running on 64 cores on 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.

## 4.4 Communication Bottleneck in ChaNGa

In some applications, due to the science requirement or due to task mapping, some processors get much more communication requests than the others. We call this a 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 "mirror" idea to solve this problem. Besides the original tasks and data, we keep several copies of tasks and data, which are called a "mirror". Data requests can go to either the original copy or the mirrors. These mirrors are distributed on other processors to avoid the heavy communication on the proces-

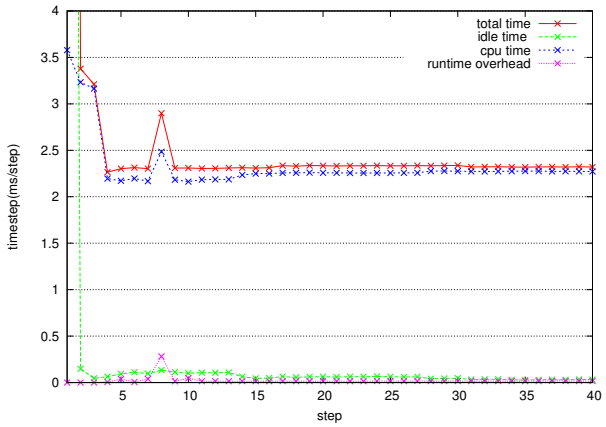



**Figure 6: Jacobi3d performance steering on 64 cores for problem of 1024*1024*1024**

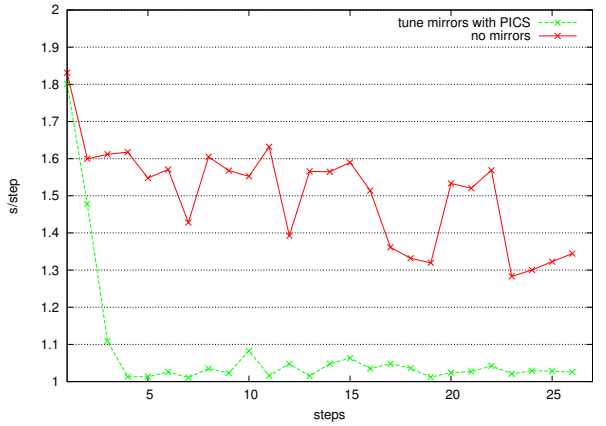



**Figure 7: Time cost of calculating gravity with various mirrors and no mirror on 16k cores on Blue Gene/Q**

sor where the original data resides. By carefully selecting the task processor mapping and the number of mirrors each task has, we can minimize the communication deviation on various processors. Depending on the specific application, the number of mirrors each original task should have is the control point we added in the runtime system. One particular real application that shows this communication bottleneck problem is ChaNGa [14], 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 7 compares the time cost of calculating gravity without using mirror and with using various number of mirrors. The red curve on the top 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. Here, we have generalized the idea and make the number of mirrors a runtime control point.

## 5. RELATED WORK

Autonomic computing and adaptive systems have been proposed as one method to deal with the rising complexity of computer systems [15, **?**, **?**]. Adaptive techniques have been built to provide performance in web servers [16]. In the high performance computing areas, the three most important projects are Autopilot [17], Active Harmony [18] and MATE [19]. 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 [18] or in an online manner [20]. 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 [20]. 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.

## 6. CONCLUSION AND FUTURE WORK

This paper has proposed a novel control system methodology in which programs express control points by providing tunable parameters along with information about their effects. Instead of focusing on the optimization techniques, one novelty of this work is that it is built on automatic performance analysis, which detects the performance problem and correlate the problems with corresponding control points. This dramatically reduces the search space of the configurations. A set of rules of program characteristics and their solutions are summarized in this paper. We also described that our control system is not only able to steer the applications for reconfiguration but also the runtime system itself. A set of control points are abstracted to guide the users to optimize their applications.

We have implemented the system within the Charm++ runtime system. Different control points have been added to three parallel programs. The relationships between these control point values, the resulting program performance and measurable effects were discussed. For all the control points, it was 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 fully address all of these issues, we will continue adding control points to more applications. We will fully analyze the various costs and benefits of various tuning schemes as we implement them.

As computers are moving towards more complex, larger parallel systems, and with exascale computing ahead, we be-

lieve 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.

## Acknowledgment

This work was supported in part by NIH Grant 9P41GM104601, Center for Macromolecular Modeling and Bioinformatics. It was also supported in part by DOE DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory.

## 7. REFERENCES

[1] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

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

[3] 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.

[4] 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.

[5] 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.

[6] Laxmikant Kale, Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Lifflander, Harshitha Menon, Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman, and Lukasz Wesolowski. Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge. Technical Report 12-47, Parallel Programming Laboratory, November 2012.

[7] L. V. Kalé, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 251–261, 1998.

[8] Harshitha Menon, Nikhil Jain, Gengbin Zheng, and Laxmikant V. Kalé. Automated load balancing invocation based on application characteristics. In *IEEE Cluster 12*, Beijing, China, September 2012.

[9] 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.

[10] 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.

[11] Yanhua Sun Sameer Kumar 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.

[12] Howard Pritchard, Igor Gorodetsky, and Darius Buntinas. A ugni-based mpich2 nemesis network module for the cray xe. 6960:110–119, 2011.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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