Adaptive Control Systems for Highly Effective Computing

Applicant/University: Laxmikant V. Kalé, University of Illinois at Urbana-Champaign.
Street Address/City/State/Zip: 201 North Goodwin Avenue, Urbana, IL, 61801-2302
Principal Investigator: Laxmikant V. Kalé.
Postal Address: 201 North Goodwin Avenue, Urbana, IL, 61801-2302
Telephone Number: (217) 244-0094
Email: kale@illinois.edu
topic number and title
past contractor?

Contents

1 Executive Summary

For more than 30 years since Intel's 4004 marked the inception of semiconductor microprocessors, the world has experienced a technological revolution as decreasing feature sizes have enabled ever cheaper and faster processors over the years. Starting in 2003, clock frequencies stopped increasing, due to thermal considerations. Although feature sizes will decrease over the next decade, this will come at the cost of decreased reliability, and larger issues involving power and energy. In order for applications of critical importance to the nation to continue to benefit from cutting-edge technology, it is clear that new research is needed. The proposed research aims at developing an adaptive control system to this end.

The design space in this new era is large: to address the broad issues of energy, performance, and reliability, we must deal with a higher dimensional space: the total power draw of the system and the energy cost of a computation are separate issues, as is the core temperature of each individual core in the system. There are complex interdependences between these concerns: component reliability may be tied to the voltages and to operating temperatures in each chip. The end user's objectives may also vary depending on application characteristics, machine characteristics, and the urgency of the computation: performance within power constraints for some urgent tasks, and energy minimization given a deadline for another, for example.

The proposed work is to develop an adaptive control system that can follow user's objectives and adjust system behavior to optimize the criteria of interest. By using a runtime system to monitor system conditions and dynamically adapt to them, we can pursue the user's goals for performance, reliability, and energy efficiency while minimizing the need to modify the applications directly.

We will develop and demonstrate the ability to control application energy efficiency, performance, and reliability through runtime control systems such as energy-aware load balancers and fault tolerance schemes, using the infrastructure provided by the Charm++ runtime system.

2 Introduction

For more than 30 years since Intel's 4004 marked the inception of semiconductor microprocessors, the world has experienced a technological revolution as decreasing feature sizes have enabled ever cheaper and faster processors over the years. Starting in 2003, clock frequencies stopped increasing, due to thermal considerations. Although feature sizes will decrease over the next decade, this will come at the cost of decreased reliability and larger issues involving power and energy. In order for applications of critical importance to the nation to continue to benefit from cutting-edge technology, it is clear that new research is needed.

The categories of applications one must consider for such future machines is also diverse. As discussed in section 4.4, these include science/engineering simulations, but also discrete event simulations, graph algorithms, and combinatorial search (including game tree search) applications. Each of these application categories brings its own perspective on the interplay between and the importance of various metrics of measuring system behavior.

The number of dimensions along which one can measure or adjust system behavior is also increasingly large. In addition to application execution time, we may consider individual core temperatures, cooling energy level, frequency/voltage for each processor, and so on.

The vision of the proposed work is to design a control system through which a runtime system can affect an application along these dimensions, and study some of the possible control operations in the context of future machines, and to demonstrate the use of these operations for a collection of application classes.

To implement a good control system, one needs to have an adequate set of levers — knobs that the system is allowed to vary to effect desired changes in the system behavior. We believe that programming models based on "overdecomposition" and its associated migratable-objects model is provides such a set of knobs to the runtime system. This programming model has been developed for over a decade in programming systems such as Charm++ and Adaptive MPI, and has been demonstrated in multiple production quality applications including NAMD.

In this execution model (described in detail in section 3), the computation is decomposed into a set of work and data units (objects) by the programmer, but their assignment to processors is controlled by the runtime system (RTS). Since the program is written without explicit reference to processors, the RTS is free to change this assignment as the program executes.

To illustrate what a control system based on overdecomposition can do in a simpler context, consider the problem of controlling processor core tempera-

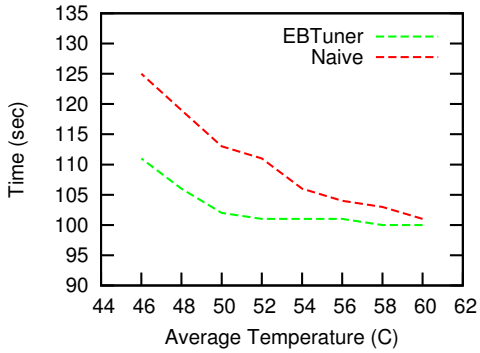


Figure 1: The effect of active runtime management on performance. The EBTuner scheme migrates work away from underclocked processors to maintain load balance while lowering ambient temperature.

ture. We constrain the temperature of the system by reducing processor frequency where the ambient temperature is too high. However, this reduction has the effect of reducing performance and increasing time to completion. As the target temperature is pushed lower and lower, the performance penalty becomes more severe. However, by active management by the runtime system can alleviate the performance penalty by migrating objects away from underclocked processors toward processors that have not been underclocked, thereby preserving load balance by taking advantage of the overdecomposition of the initial system. This strategy, described more fully in [1], delivers a significant performance improvement over the naive approach of leaving all work in its initial location, as shown in figure 1.

Thus, an adaptive runtime system with the ability to migrate objects makes it possible to reduce the total energy used by a computation or to reduce execution time, compared with the baseline strategy of reducing frequency and voltage for any core that exceeds the desired temperature. Yet, the RTS leaves to question of which parameter to optimize open. Depending on the context, energy, time, or a weighted average may be more important to optimize. A meta-control-system is needed to analyze the top level concerns from the user, understand the features of the application class, and decide what criteria to assign to the normal RTS. The RTS itself must be expanded significantly to deal with a large numbers of sensors and control knobs that future systems will have or require (as described in section 4).

It is the design of this expanded RTS and the meta-control-system that is the objective we set for ourselves in this proposed work.

3 Past Work

We now describe the execution model and the principles underlying our approach to reliability and energy efficiency. Migratable, overdecomposed units of work and data form the basis of our execution model (§3.1), which empowers the adaptive runtime system (§3.3), enabling the adaptive strategies described later (§**??**).

3.1 Execution Model and Cost Model

The computation in our execution model consists of a large number of units. Each unit may be a work unit (WU), a data unit (DU), or more typically, an amalgam of the two (WUDU). An example of a pure work unit is a pure function that is given an input and produces an output without a side effect. A pure data unit is a slice of data that can be read or written as a unit from outside. Common examples of amalgams of the two include a thread with its own stack, or an object with encapsulated data and some methods to operate on. Since both extremes can be thought of as a special case of an amalgamated work unit/data unit, we refer to them all as WUDUs. In addition to the WUDUs, our ontology includes the notion of a trigger. A simple example of a trigger is an MPI-style message; but a trigger may also be a method invocation, a continuation, or a "kick" devoid of any data that is supposed to trigger some computation when it is received at a WUDU.

Each work unit is further broken down into dependent execution blocks (DEBs), each of which depends on one or more pieces of data that is not local to the work unit concerned. In addition, optionally, it may also depend on a condition local to the work unit, typically indicating the readiness of the work unit to undertake the computation of the DEB. A DEB's dependencies are satisfied when all its dependent data is locally available, and its conditions are satisfi are satisfied is called a *ready* DEB. A DEB's execution is atomic, in the sense that it is not paused awaiting additional data (since there are no additional dependencies beyond the initial ones), nor is another DEB associated with the same WUDU allowed to execute before it completes.

Although not a part of the execution model, we also define the notion of a sequential execution block (SEB), for completeness. A SEB is a section of a DEB that is separated for analysis because of its cohesion. This could be a loop nest or a function call, or just an interval between two points in the execution of a DEB. Thus, a WUDU is a unit of migration, a DEB is a unit of scheduling, and a SEB is the smallest unit of analysis/introspection.

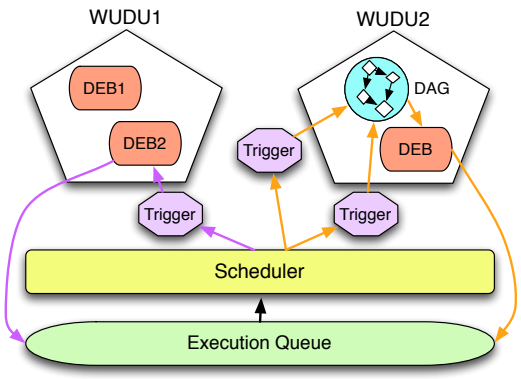


Figure 2: Execution within a processor

Execution proceeds by selecting an independent set of ready DEBs, and executing an arbitrary subset of them. Two DEBs of the same WUDU are considered dependent, and are not executed simultaneously. Execution of a DEB may lead to a change in the state of the WUDU that owns it, and to creation of new triggers, thereby adding to the set of ready DEBs. Execution continues in this fashion until one of the DEBs calls for termination of the program. The quiescent condition, when there are no ready DEBs, is not necessarily a ground for termination. It is possible to specify that a trigger be created when quiescence is attained, thereby starting the next phase of the computation.

An efficient way of ensuring *independence* and *atomicity* properties mentioned above is to anchor each WUDU to a single core, at a time. This is the default strategy we adopt. But we can relax it when needed, as long as these properties are upheld.

The Cost Model for the application developer, or for the higher level languages that translate to the level of WUDUs is as follows. Creation of each trigger has a cost depending on the size of the payload data on it, via a standard $\alpha + n\beta$ form, where α and β are expressed in (say) cycles. Accessing data within a WUDU is relatively cheap, whereas accessing data via triggers is expensive. Accessing data in other WUDUs directly is not allowed.

This execution model is similar to that used in Charm++ for many years, as well as in models such as HPX. Its utility, which was relatively small at terascale for most applications, is much higher at exascale, because of its flexibility, and the increased need for an adaptive runtime system.

3.2 Message-driven Execution

The execution model is *message-driven*: since there are multiple WUDUs on each core, a user-level scheduler must schedule DEBs based on availability of a trigger, and optionally, DAG dependencies within the WUDU (see Figure 2). This confers performance and modularity benefits to the application: if one DEB is waiting for some remote data, another ready DEB is scheduled for execution. This generates an adaptive and *automatic* overlap of communication and computation. Further, it spreads communication over the iteration, unlike the current MPI compute-communicate paradigm, which leads to the network being used for only a small fraction of the time. As a result, one can avoid the need for aggressively engineered (and power-inefficient) networks.

Compositionality: Since executions of distinct modules can interleave in a message-driven manner on a single core, parallel composition of parallel modules is well-supported by this model [2].

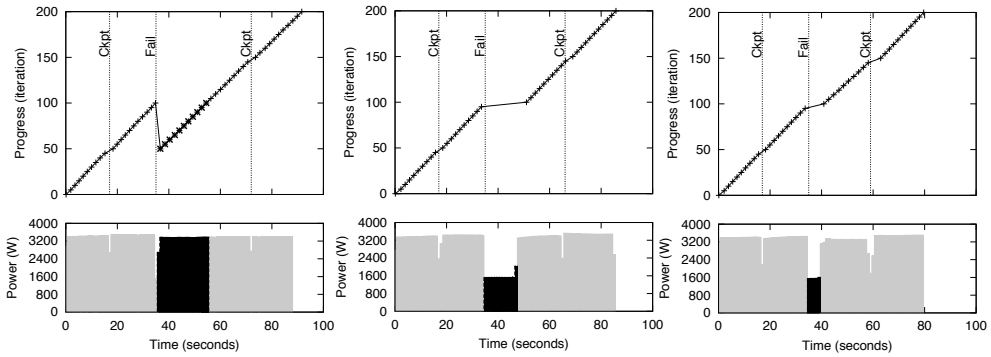
3.3 WUDUs empower Runtime Systems

In our scheme, the programmer (at least at the base level) is still responsible for the grainsize decisions; but the grainsize decision can be made without reference to the total number of processors. For good performance the computation of a DEB should be large enough to amortize the cost of its trigger. Another consideration in deciding the grainsize is that of the memory overhead. However, none of these quantities depend on the number of processors. Further, application developers (or compilers) may produce parameterized code which makes it easy to change the grainsize (See §**??**).

As a consequence, this methodology produces an abundance of parallelism: the number of WUDUs (and dynamically, the number of ready DEBs) can be much larger than the number of processors. This *over-decomposition* fundamentally empowers the runtime system to optimize the execution in multiple ways (§**??**), without intervention from the application developer.

3.4 Impact of Fault Tolerance in Energy Consumption

Power management and fault tolerance are considered two of the most critical challenges to deliver an exascale machine [3]. In a recent study [**?**], we analyzed the energy profile of different fault tolerance strategies implemented in the Charm++ runtime system. Our goal in that study was to understand how much



(a) Checkpoint/Restart (b) Message-Logging (c) Parallel Recovery

Figure 3: Comparison of three fault tolerance strategies according to their performance and energy consumption.

benefit it is possible to get by having an intelligent runtime system able to migrate tasks in the system. We compared the in-memory checkpoint/restart, a particular flavor of message-logging and an extension of message-logging with parallel recovery (a technique unique to our migratable-object technology).

Figure 3 shows an experiment of that study. For a 7-point stencil code, we ran the three different fault tolerance strategies through a faulty execution. The top figure is called a *progress diagram* and presents the number of iterations completed as a function of time. A higher slope in the curve stands for a faster algorithm. A failure in the execution requires the progress to return to the previous saved checkpoint. However, in the case of message-logging and parallel recovery, only one node in rolled back. During recovery, the three strategies differ dramatically. Whereas checkpoint/restart requires all nodes to roll back and re-execute the code after a crash, message-logging techniques only have one node executing the lost work. This makes the rest of the system to stay idle and return to the base power. Furthermore, parallel recovery shrinks the recovering time, adding more energy savings.

We developed an analytical model to understand the impact of these techniques at exascale, when failure rates are expected to be high. Our model predicts parallel recovery to reduce 13% energy consumption and 17% time execution when compared with checkpoint/restart.

4 Proposed Work

The Charm++ runtime system gives us the opportunity to dynamically collect a wide variety of data about system performance, and to react to that data dynamically. However, the complexity of the relationships between power, energy efficiency, performance, and reliability means that there is no simple relationship or set of rules that we can reliably follow to achieve the application user's objectives. Rather, we must monitor a variety of input parameters and adjust the runtime's behavior to meet system constraints and manage the trade-offs between our various objectives.

Foremost among those constraints is total power consumption. This is typically an unchanging "maximum system power" that depends on the physical infrastructure of the machine in question, but may also include the possibility of short duration spikes past the nominal peak. Further, there may also be a minimum power consumption so as to not cause sudden load changes to the power grid [true??].

While today's systems are typically designed to be unable to exceed their maximum power budget even if all nodes are drawing as much power as possible, potentially much better results could be obtained by overprovisioning hardware, so that more energy efficient applications can make use of a greater number of cores without exceeding the total power budget. In this scenario, active application management to avoid exceeding the total power available is extremely important.

In addition to meeting hard constraints like total power consumption, the runtime system must also make trade-offs between different objectives based on measurements. For example, we can monitor the temperature of each core in the system, and dynamically adjust the threshold beyond which we attempt to reduce the temperature. By setting a higher threshold, you can save cooling energy, but you also risk higher fault rates.

Our approach for addressing these complex issues has three parts. First, we will develop a model which expresses the relationships between the various system measurements we can make, the constraints we must work within, and the actions that the runtime system can make. Second, we will investigate techniques for dynamically monitoring those quantities that can be used to guide runtime decisions and providing that data to the runtime system. Finally, we will investigate the range of control systems that can be used by the runtime to change the balance of performance, reliability, and energy efficiency in an application dynamically.

4.1 Modeling

It is simple enough to speak qualitatively about the trade-offs between energy efficiency, time to completion, and reliability, but the quantitative relationship between these quantities in the context of supercomputing applications can quickly become fiendishly complicated. In order to successfully pursue a user's objectives with regard to these trade-offs, we must construct a model with which we can predict the ways in which concrete actions taken by the runtime system can be expected to affect performance, reliability, and energy efficiency.

The runtime system can control the scheduling of SEBs onto physical resources, the power draw of individual chips, the checkpoint interval, and so on. It can measure core temperatures and power draw and the progress rates for individual SEBs. We will construct a model that, given these inputs and an objective function, will determine what actions the runtime system should take in pursuit of the objective. This objective may have hard constraints based on the host machine (such as maximum total power draw), based on application logic (such as maximum aggregate error rate), or based on external factors (such as a hard deadline for completion). The goal is to optimize other factors under these constraints (for example, attaining maximum energy efficiency while completing execution by some deadline).

Much of the work of constructing this model will consist of characterizing the response of existing hardware to changes in scheduling policy, fault tolerance policy, power draw, and so on.

4.2 Monitoring

In order for the runtime system to make dynamic decisions in response to environmental conditions, it must have accurate data about the state of those conditions and the way that they have responded to past actions by the runtime. In order to facilitate the collection and use of this data, we will investigate the use of a continuous introspection framework, which will efficiently maintain a database of recent behavior of the WUDUs and their individual code-blocks, using hardware counters, sensors and timers. Modern microprocessors provide extremely cheap (fast) access to such counters, which we plan to leverage; e.g., the cost of a timer call on the Sandy Bridge machine is 25 nanoseconds. It is feasible to maintain such a database effectively because all the events required to monitor the activity are accessible to the runtime system. The placement of WUDUS on processors is done by the runtime, communication between WUDUs is mediated by the run-

time and execution of individual DEBs on physical resources is also scheduled by the runtime. The WUDUs are relatively coarse-grained (as a guide to intuition, readers may assume tens of the WUDUs on a core, although in some applications one may use thousands of them per core), so that the memory overhead of this database is modest on each core.

4.3 Control Systems

Our proposed work is premised on the idea that a runtime system has a variety of possible actions available to it, through which it can influence an application's energy efficiency, performance, and reliability. A key part of our research agenda will be to develop these control systems by which the runtime can steer an application through the parameter space described by these variables.

The fundamental task of the runtime system in our execution model is to schedule WUDUs onto physical resources. This mapping of tasks and data to hardware can in itself significantly affect the energy efficiency and performance characteristics of an application. However, by combining this mapping with explicit control of power consumption for individual chips, selection of fault tolerance protocols and checkpoint intervals in accordance with the model described in section 4.1, we will be able to change the performance, energy, and reliability characteristics of an application according to the user's goals.

- Power clamping: by adjusting the power consumption of individual hardware elements, the runtime system can react to local load imbalances or data center hot-spots.

- Mapping of tasks and data to processors: load imbalances can also be addressed by changing the mapping of tasks to processors. In addition, not all SEBs are equally sensitive to processor frequency, so energy efficiency can potentially be gained by running less sensitive SEBs at lower frequencies.

- Fault tolerance protocol: by comparing reliability estimates based on current machine conditions with the user's tolerance for failures, the runtime can adjust the checkpoint interval and use either more or less conservative fault tolerance protocols. In cases where extra reliability is needed, parts of the application may be replicated and their results tested for agreement, allowing for triple modular redundancy for components where high reliability is paramount.

- Dynamic job shrinkage and expansion: depending on available hardware resources, the set of processors running the job can be expanded or shrunk, leaving the remainder in a low-energy mode or available for other computations. The runtime system remaps application WUDUs to either take advantage of newly available hardware resources or to consolidate tasks on a smaller set of processors.

Of course, in order to effectively pursue a user's desired priorities, the runtime system must be aware of what those priorities are. As part of our research, we will investigate ways to communicate information about the user's intent with regard to energy, performance, and reliability trade-offs, whether through APIs, metadata associated with the application, or direct human intervention through real-time interactive tools. As part of our proposed research, we will investigate a variety of control systems relevant to energy efficiency and reliability.

4.4 Application Scenarios

Depending on application characteristics and the needs of the user, different trade-offs between energy efficiency, time to completion, and reliability will be appropriate. Consequently, the policies pursued by the runtime system must depend on the context in which the application is running. To illustrate the way that these concerns may play out in practice, consider the following different application scenarios. Each requires a different way of navigating the parameter space.

- A physical simulation, such as fluid dynamics or weather modeling. A possible user objective in this scenario may be to minimize execution time, while keeping energy low to the extent possible, and to run with relatively low core temperatures such that component reliability is high in order to minimize the time lost to checkpointing and recovery protocols. Alternatively, if the simulation doesn't have a strong deadline, and the supercomputer facility is not too busy with other jobs, one may want to minimize the energy taken by the overall computation instead.

- A distributed real-time simulation with humans-in-the-loop (e.g. a war-game) which involves one or more components running on parallel computers: Here, the parallel components must keep pace with the real-time simulation. Checkpoint/restart or even message-logging may not be enough to avoid hiccups in the simulation in case of failures. A triple-modular-redundant (TMR) simulation may be appropriate here. And since we are

using TMR, it may be worthwhile to keep energy low by running at near-threshold voltage, which reduces reliability but decreases energy substantially.

- A graph algorithm that has a non-iterative structure (unlike physical simulations). The floating point unit is not used much by this application, and performance is dominated by memory access and communication. One can afford to run processors at low frequencies, just high enough to match the memory bandwidth.

- Combinatorial state-space search. Most combinatorial search problems are "needle-in-haystack" problems, with a tree-structured computation. Most branches of the tree represent "wasted" computation because they don't contain the solution. But, in such a situation, if the number of feasible solutions is large, one can run the computation in a relatively low reliability mode. Even if some branches go wrong (either missing a solution, or mispredicting a solution, which can be verified as infeasible very efficiently), it's very likely that one of the solutions will be found, and with a low-energy cost.

5 Schedule

6 Milestones

7 Deliverables

8 Management Approach

References

[1] Osman Sarood and Laxmikant V. Kalé. A 'cool' load balancer for parallel applications. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[2] Attila Gursoy and L.V. Kalé. Performance and Modularity Benefits of Message-Driven Execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.

[3] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.