Scalable and Energy Efficient Execution Methods for Multicore Systems

Dong Li

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Kirk W. Cameron, Chair
Dimitrios S. Nikolopoulos
Bronis R. de Supinski
Wu-chun Feng
Xiaosong Ma

January 26, 2011
Blacksburg, Virginia

Keywords: Multicore Processors, Performance Modeling and Analysis, Power-Aware Computing, Concurrency Throttling, High-Performance Computing

Scalable and Energy Efficient Execution Methods for Multicore Systems

Dong Li

ABSTRACT

Multicore architectures impose great pressure on resource management. The exploration spaces available for resource management increase explosively, especially for large-scale high-end computing systems. The availability of abundant parallelism causes scalability concerns at all levels. Multicore architectures also impose pressure on power management. Growth in the number of cores causes continuous growth in power.

In this dissertation, we introduce methods and techniques to enable scalable and energy efficient execution of parallel applications on multicore architectures. We study strategies and methodologies that combine DCT and DVFS for the hybrid MPI/OpenMP programming model. Our algorithms yield substantial energy saving (8.74% on average and up to 13.8%) with either negligible performance loss or performance gain (up to 7.5%).

To save additional energy for high-end computing systems, we propose a power-aware MPI task aggregation framework. The framework predicts the performance effect of task aggregation in both computation and communication phases and its impact in terms of execution time and energy of MPI programs. Our framework provides accurate predictions that lead to substantial energy saving through aggregation (64.87% on average and up to 70.03%) with tolerable performance loss (under 5%).

As we aggregate multiple MPI tasks within the same node, we have the scalability concern of memory registration for high performance networking. We propose a new memory registration/deregistration strategy to reduce registered memory on multicore architectures with

helper threads. We investigate design polices and performance implications of the helper thread approach. Our method efficiently reduces registered memory (23.62% on average and up to 49.39%) and avoids memory registration/deregistration costs for reused communication memory. Our system enables the execution of application input sets that could not run to the completion with the memory registration limitation.

Dedicated to my amazing wife Chunyan, who is always there for me with love and support.

ACKNOWLEDGEMENTS

This dissertation was completed with a lot of support of my family, academic advisors and friends. Without them, I could not finish the thesis. First and foremost, I would like to thank my amazing wife, Chunyan. She took over most of our family responsibilities so that I could focus on my work. I cannot thank her enough for the support she gave me for such a long distance for four and a half years during my time in graduate school. I am happy that our lives will move on to the next exciting page after the end of this work. I would also like to acknowledge my parents, Zhaohui and Zhipeng. Although they are 11,000 miles away from me, their encouragement and love are always there with me.

I owe a great deal to my advisor Dr. Kirk W. Cameron for his guidance during my graduate study. He taught me how to survive an intensive Ph.D. program. I am also in appreciation of my co-advisor, Dr. Dimitrios S. Nikolopoulos. I learned a lot from him about how to do research. He told me to figure out where every CPU cycle goes in order to make my work solid. It is this kind of research spirit that generates excellent work. I also owe a lot of thanks to Dr. Bronis R. de Supinski. He shed light on the path when I was totally lost in my research. Without him, I could not have made progress as fast as I did. I am thankful to Dr. Martin Schulz, for his patience and insightful advice. I would also like to thank my other Ph.D. committee members for their discussions and suggestion: Dr. Wu-chun Feng and Dr. Xiaosong Ma.

I have been very fortunate to work with some really great friends while at Virginia Tech. Rong Ge and Xizhou Feng are like my big sister and brother. They helped me a lot when I was in Blacksburg. I have enjoyed my conversations with Hung-Ching Chang, my office buddy. He made our office time more interesting. I am also thankful to my friends, Filip Blagojevic, Matthew Curtis-Maury, Song Huang, Aleksandr Khasymski, Hari Pyla, Scott Schneider, Shuaiwen Song, Chun-Yi Su, Matthew Tolentino, Joseph Turner, Guanying

Wang and Jae-seung Yeom. I always look forward to discussing ideas and chatting casually with them.

Finally, I would like to thank the Department of Computer Science at Virginia Tech for providing such an excellent research environment for me.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

Chapter 1

Introduction

Traditional single microprocessor designs have exerted great effort to increase processor frequency and exploit instruction level parallelism (ILP) to improve performance. However, they have arrived at a bottleneck wherein doubling the number of transistors in a serial CPU results in only a modest increase in performance with a significant increase in energy. This bottleneck has motivated us into the multicore era. With multicore, we can achieve higher throughput with acceptable power [104, 110], although the core-level frequency of a multicore processor may be lower than that of a serial CPU. To continue scaling throughput performance, more and more cores are placed on a die/chip , for example, an Intel Single-chip Cloud Computer with 48 cores [128], a Tilera TILE-Gx processor with 100 cores [91] and an AMD Phenom II X6 1090T with 6 cores [90]. With this trend of an increasing number of cores on a chip, the previous performance trend—doubling performance per processing element every couple years—is ended. Instead, we observe that doubling the number of processors causes many programs to go twice as fast [93].

The switch to parallel microprocessor implies widespread in-depth changes to software design and implementation. It is predicted that multicore architectures and the supporting software technologies will reset microprocessor hardware and software roadmaps for the next

30 years [11]. The popularity of multicore architectures calls for new design considerations for high performance applications, parallel programming models, operating systems, compiler designs and so on.

In this chapter, we provide the necessary background for understanding the research performed in this dissertation. In particular, section 1.1 provides insights into the challenges and problems that we attempt to address on multicores. Section 1.3 summarizes the contributions that we make in this dissertation. Section 1.4 outlines the remainder of this dissertation.

1.1 Research Challenges on Multicore

The primary consequence of using multicores is that applications have to be concurrent in order to exploit the continuing exponential processor throughput gains. Therefore, programming languages and systems are increasingly forced to deal with concurrency. Meanwhile, efficiency and performance optimization, which were well researched on serial CPUs, become more important in multicores due to the need to leverage hardware resources fully and to boost performance. The efforts that were invested on serial CPUs must be transitioned and adapted to the new parallelism provided by multicores. Presently, we are still in the process of this adaption and transition. A simple example is the popular parallel programming model MPI [63]. While born as a standard for communication among processes mainly across nodes, many changes have been introduced to optimize MPI performance on multicore platforms, for example, optimizing intra-node communication [37, 38, 101] and optimizing collective operations [124, 180, 182]. Today, MPI is still undergoing changes to adapt to the multicore revolution. The mechanisms and policies to leverage the abundant computing elements to improve MPI performance are still under exploration. Many questions on how to implement MPI on heterogenous multicores are still open and widely debated.

In this section, we discuss two topics that are important for high performance execution on multicore platforms: scalability and power-awareness. They are two of the major challenges on multicores [50].

1.1.1 Scalable Execution on Multicore

Scalable execution requires efficient resource management. Resource management includes determining the configuration of applications and systems at a given concurrency level: for example, how many tasks should be placed within the same multicore node and how to distribute tasks scheduled into the same node between idle cores. Resource management may also indicate adjusting adaptable frequencies of processors by voltage and frequency settings [40]. Along with an increasing of number of nodes and number of processors/cores in high-end computing systems, resource management becomes more challenging, because the exploration space available for choosing configurations increase explosively.

We use the following simple example to explain this problem. Suppose we have two quad-core symmetric processors within a node and we decide to aggregate four applications or to place four threads into the same node. We can put them on the same processor (Figure 1.1(a)), or different processor with die sharing (Figure 1.1(b)), or without die sharing (Figure 1.1(c)). We can even put them in an unbalanced way (Figure 1.1(d)). Therefore, just for the case of four applications/threads per node, we have four feasible options. We further assume that there are three frequency scales available at each core and we can selectively change the frequency of each core. Then we have 12,099 different configurations from which to choose. Therefore, even with this simple example we have a large exploration space that is not seen with serial CPUs. This example does not even consider a NUMA system or an asymmetric processor like Cell broadband engine [39], which will further increase option counts. We predict that this resource management problem will be further exacerbated for

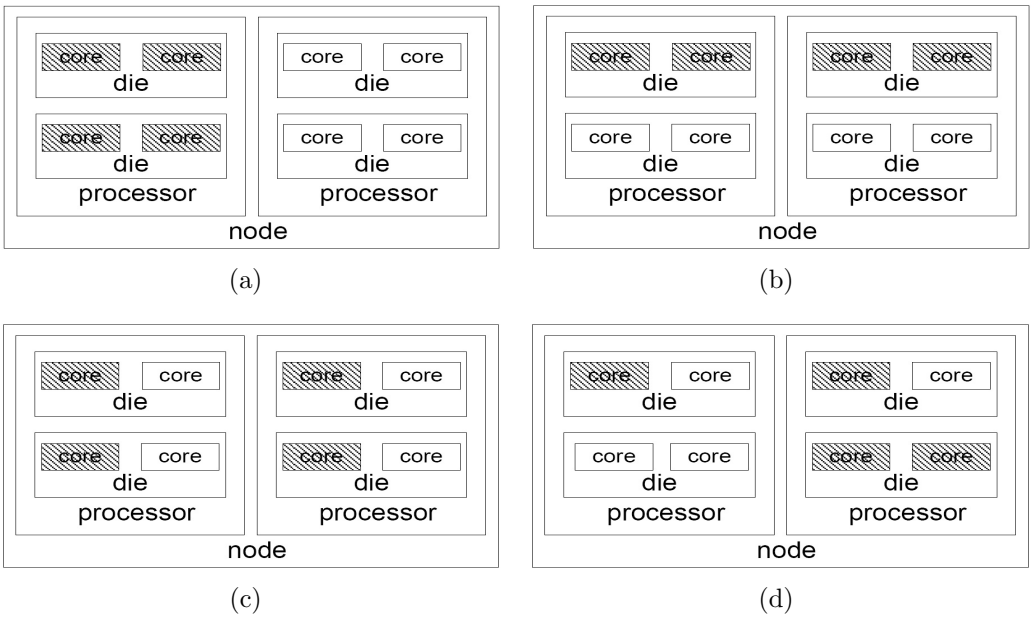


Figure 1.1: Feasible concurrency configurations with four applications/threads per node

future exascale systems equipped with many-core chips.

The explosively increased exploration space requires scalable resource management. To determine the appropriate configurations, we must pinpoint the optimal points in the exploration space without testing every possible configuration. We also must avoid introducing a negative performance impact during the process of configuration determination.

In addition to resource management, scalable execution also implies scalability concerns at other levels. For example, we need parallel programming models that support scalable programming; we must avoid performance bottlenecks caused by resource contention that occurs on the shared resources.

1.1.2 Power-Aware Execution

Power-aware execution is the other challenge posed by multicore. Growth in the number of cores of large-scale systems causes continual growth in power. Today several of the most powerful supercomputers equipped with multicore nodes on the TOP500 List [161] require

up to 10 megawatts of peak power–enough to sustain a city of 40,000 people [59]. The best supercomputer power efficiency is about 0.5-0.7 Gigaflops/Watt [167]. As far as the future exascale systems are concerned, they will require at least 40 Gigaflops/Watt to maintain the total cost of ownership, which means two orders of magnitude improvement is needed [167]. It has also been noted that the power dissipation of computational components increases by 2.7 times every two years [28]. With the present trend in power increase, the cost of supplying electrical power will quickly become as large as the initial purchase price of the computing system [15]. High power consumption leads to high utility costs and causes many facilities to reach full capacity sooner than expected. It can even limit performance due to thermal stress on hardware [59].

To alleviate the power/energy crisis faced by high performance computing communities, people have devised novel hardware and software techniques to reduce power and energy consumption. Research has focused on driving new levels of energy-efficiency through hardware optimization. Although hardware approaches have the potential to reduce power and energy significantly [28], they lack flexibility and have a long development period. Also, people cannot easily verify hardware functionality and must resort to hardware simulation. Optimized software, on the other hand, has benefits for power/energy control that cannot be achieved by hardware: (1) it can use very fine-grained system information (e.g., process-level memory usage or application workload characteristics), which is helpful for designing delicate power/energy control schemes and conducting performance analysis; (2) it provides greater flexibility in studying power efficiency for distributed and parallel systems (e.g., thermal-aware process migration [41, 130, 131, 190] or dynamically distributing users workloads for high-end servers [80, 82, 149]). In addition, software can explore and leverage hardware (e.g., dynamic voltage and frequency scaling (DVFS) and cooling fan [119]) to reduce power and heat. In this dissertation, we will resort to software control for power and energy control on multicore systems.

Using software to improve the power efficiency of HPC applications on multicore systems poses several challenges. (1) The modeling/prediction of power on multicore systems is a challenge. The modeling/prediction must be accurate enough to reflect power/energy variance when our hardware usage is changed; it must also be lightweight enough so that we can leverage it at runtime. (2) We face many unknowns with power consumption for large scale high-end computing systems, which have a more significant power problem than small scale clusters. For example, how will the power saving opportunities vary as we scale the system? Does program execution at larger scales always lead to higher energy consumption? Will power control techniques such as DVFS be effective at larger scale? Exploring power issues at a large scale is useful for detecting the tradeoff between performance and power consumption. (3) Parallel applications have unique power and thermal characteristics, which are not seen in traditional serial applications (e.g., variant power consumption across application threads and thermal interaction between cores). We need to capture, to investigate and even to leverage these characteristics.

The above concerns for scalability and power-awareness can be interleaved. For example, the performances of some applications fail to scale with the number of processing elements due to inherent program characteristics and architectural properties [50]. By appropriately selecting the number of processing elements used to execute these applications, we can have both performance improvements and energy savings.

1.2 Research Objectives

This dissertation aims to create efficient software control methodologies to improve the performance of high performance computing applications in terms of both execution time and energy. Our work is an attempt to adapt software stacks to emerging multicore systems.

The objective of this research is: (1) to design and to implement efficient resource man-

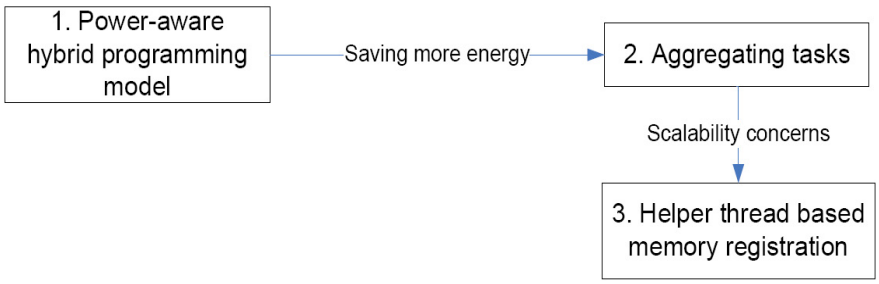


Figure 1.2: The relationship between the three thesis parts

agement tools for large-scale HPC systems based on multicore architectures; (2) to identify, to study, and to model power and energy problems that are unique in multicore architectures and manifest themselves at scale on high-end computing systems; (3) to investigate the characteristics of high performance computing applications that can be leveraged to reduce power and energy; and (4) to develop a performance-directed power/energy-aware software control framework. The framework should be scalable to systems with many nodes with many cores each and should minimize its impact on the performance of applications. The scope of this research extends across system software and hardware in several layers: the parallel programming model, communication library, operating system and processor architecture.

This dissertation consists of three parts. The relationship of the three parts is depicted in Figure 1.2. First, we explore power-aware models and algorithms for the hybrid MPI/OpenMP programming model on multicore systems. Second, we propose aggregating multiple tasks within the same node as a further step towards power/energy saving. To solve the scalability problem of memory registration for high performance networking when aggregating tasks, we then propose in the third part a novel memory registration scheme with helper threads. All three parts form the framework for scalable and energy efficient executions of parallel applications on multicore systems.

1.3 Research Contributions

In this subsection, we discuss particular research contributions made in the completion of this work. Each contribution will be presented with more detail in subsequent chapters of this dissertation.

1.3.1 Hybrid MPI/OpenMP Power-Aware Computing

With the current strong trend towards packing more cores on each processor and scaling high-end computing systems to tens of thousands of nodes, more programs are expected to be written in hybrid programming models, such as MPI/OpenMP [148], to utilize both shared memory and distributed memory for efficient communication best.

Prior research in power-aware execution focuses on programs written in a single programming model, typically MPI [63] or OpenMP [141]. However, a hybrid programming model that combines MPI and OpenMP is growing in popularity due to its suitability for large-scale systems with multicore processors. We study how to apply power saving techniques (in particular DVFS and DCT) in a hybrid MPI/OpenMP setting. This application is challenging since our problem expands from one dimension (single programming model) to two dimensions (hybrid model), which greatly increases the complexity of our exploration in the configuration space.

We study the performance implications of applying DCT in hybrid applications. Specifically, we analyze the implicit penalty of concurrent throttling on last-level cache misses and propose a DCT algorithm that aggregates OpenMP phases to overcome this problem. We also consider the effects of applying DCT locally without considering the interaction between tasks and propose a novel interaction-aware DCT coordination scheme.

We present a method that classifies hybrid applications in order to facilitate the identification of slack time and characterizes the slack available for DVFS, considering both

intra-node and inter-node interaction effects. We formalize the problem of frequency selection and propose a solution that uses dynamic programming.

Based on studies of DCT and DVFS, we model hybrid OpenMP/MPI applications when their shared-memory components are executed under a varying degree of concurrency and their waiting phases are executed under varying frequency, using DCT and DVFS respectively. The derived models formalize execution behaviors and energy saving opportunities. We further investigate three strategies of applying DCT and DVFS and identify the best strategy that minimizes energy consumption.

To apply the power-aware hybrid MPI/OpenMP programming model, we must estimate power consumption and execution time of OpenMP phases with variant concurrency and frequency configurations. Therefore, we present a system power estimation method and an execution time prediction method that uses the sample configurations to learn execution properties and to make predictions.

Based on our study, we develop a runtime library that performs online adaptation of DVFS and DCT for hybrid MPI/OpenMP applications. We study energy saving opportunities in both strong scaling and weak scaling using realistic parallel applications on system scales of up to 1024 cores.

1.3.2 Power-Aware MPI Task Aggregation Prediction

Emerging large-scale systems have many nodes with several processors per node and multiple cores per processor. These systems require effective task distribution between cores, processors and nodes to achieve high levels of performance and utilization. Current scheduling strategies distribute tasks between cores according to a count of available cores, but ignore the execution time and energy implications of task aggregation (i.e., grouping multiple tasks within the same node or the same multicore processor). Task aggregation can save signifi-

cant energy while sustaining or even improving performance. However, choosing an effective task aggregation becomes more difficult as the core count and the options available for task placement increase.

We divide the task aggregation problem into three sub-problems: predicting the impact of task count per node on computation; predicting the communication cost of all aggregation patterns and; combining the computation and communication predictions to choose an aggregation pattern with minimal energy consumption.

To predict computation performance, we set up a piecewise linear model to predict the *IPC* of computation phases with different aggregation patterns. To predict communication performance, we first analyze how communication performance is sensitive to task placement within the node and across the nodes. Then we formalize the problem of deciding how tasks should be aggregated given an aggregation pattern in order to optimize communication performance. We map the problem into a graph partitioning problem and solve with a heuristic algorithm. Then we propose a method of predicting an upper bound of communication time across different aggregation levels.

We implement a tool suite to facilitate task aggregation. We evaluate our method on system scales of up to 1024 cores. We also study the effects of task aggregation with strong scaling tests.

1.3.3 Scalable Network Memory Registration Using Helper Threads

Many high performance networks leverage remote direct memory access (RDMA) to achieve high bandwidth transfers. RDMA operations require memory registration, which explicitly identifies the memory used in these transfers. Further, every page of the communication buffer must be pinned to prevent swapping by the OS. Pinning memory limits adaptability of the memory system and reduces the amount of memory that user processes can allocate.

Pinned memory is also a precious resource. CMPs increase the challenge of the memory registration problem since the demand for registered memory grows linearly with the number of processes on the node.

We propose a novel memory registration strategy that reduces the size of registered memory in pinned-based high performance networks. Our approach hides the cost of dynamic memory management by offloading all dynamic memory registration/deregistration requests to a dedicated memory management helper thread, while sustaining performance. Our solution makes more memory available to the application.

The helper thread hides operation latency through proactive memory registration and asynchronous deregistration. In particular, the helper thread predicts when it should register which memory locations using a communication context-aware predictor. The predictor leverages the predictability of HPC applications to achieve high prediction accuracy with a short learning process. The helper thread uses time-oriented techniques that avoid increasing critical path latencies. To leverage idle cores efficiently, we also investigate two design policies (i.e., aggressive and conservative policies) to distribute helper threads.

We implement our helper thread mechanism within the Open MPI communication library. We apply it to HPC applications, including communication intensive applications, and perform a detailed study of the variance of registered memory in both strong scaling and weak scaling tests. We investigate how our helper thread affects the registered memory size when aggregating multiple tasks per node.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows:

In Chapter 2, we review related work and present the context for the research performed in this dissertation. In particular, we provide background and literature survey for eight

different areas: the hybrid MPI/OpenMP programming model; power-aware MPI programming models; performance issues of concurrency throttling; power and energy estimation for multicores; communication performance models; multicore effects on communication; efficient memory registration for high performance networking; and exploring the predictability of HPC applications.

In Chapter 3, we describe our research of power-aware modeling for hybrid MPI/OpenMP applications. We present how we apply DCT and DVFS to save energy as well as the optimal strategies to apply the two techniques. We present novel models to predict performance and power consumption with different configurations.

In Chapter 4, we investigate, evaluate, and model the performance impact coming from task aggregation on multicore platforms. We show the significant energy benefits brought by task aggregation.

In Chapter 5, we present a novel scalable memory registration scheme. We describe our design and implementation details. We present a study of how to leverage idle cores efficiently to distribute helper threads. Our experiments explore the performance implications of using helper threads.

In Chapter 6, we explain future work based on our existing research. We discuss the effects of NUMA systems on DCT and propose potential work. We discuss our ongoing simulator development to improve prediction accuracy for communication performance. We discuss exploration of medium-grain concurrency throttling to improve DCT performance. We also propose aggregating hybrid MPI/OpenMP tasks to improve energy efficiency. Then we discuss the effects of task aggregation on networking. We present our initial plan of extending current work to heterogeneous computing environments.

Finally in Chapter 7,we present a brief summary of the research done in this dissertation.

This page intentionally left blank.

Chapter 2

Background and Literature Survey

This dissertation focuses on scalable and energy efficient execution methods on multicores. It involves power-aware parallel programming models, performance models, power estimation method, and network performance estimation and optimization. In this chapter, we present background and work that is most closely related to our own, with each of the eight categories of work given its own section.

2.1 Hybrid MPI/OpenMP Terminology and Related Work

Hybrid MPI/OpenMP programming models exploit coarse-grain parallelism at the task level and medium-grain parallelism at the loop level. Its hierarchical decomposition closely matches most large-scale HPC systems, which are composed of clustered nodes, each of which has multiple cores, often distributed across multiple processors. In this dissertation, we consider programs that use the common THREAD_MASTERONLY model [148]. The model uses a single master thread to invoke all MPI communication functions outside of parallel regions, although thread-safe MPI implementations may also allow the use of

MPI calls inside parallel regions. Almost all MPI programming environments support the THREAD_MASTERONLY model. Applying this model to existing MPI applications requires little programmer effort since the program is explicitly structured as MPI tasks at a high level.

OpenMP directives parallelize the sequential code of the MPI tasks. The OpenMP programming model exploits fast intra-task data communication through shared memory via loop-level parallelism. While other mechanisms (e.g., POSIX threads) could add multithreading to MPI tasks, OpenMP can incrementally parallelize the tasks and facilitates DCT [48]. Thus it is widely adopted by hybrid applications.

Iterative parallel computations dominate the execution time of scientific applications. Hybrid programming models exploit these iterations. Figure 2.1 depicts a typical iterative hybrid MPI/OpenMP computation, which partitions the computational space into subdomains, with each subdomain handled by one MPI task. The communication phase (MPI operations) exchanges subdomain boundary data or computation results between tasks. Computation phases that are parallelized with OpenMP constructs follow the communication phase. We use the term *OpenMP phases* for the computation phases delineated by OpenMP parallelization constructs.

Collections of OpenMP phases delineated by MPI operations form *OpenMP phase groups*, as shown in Figure 2.1. Typically, MPI collective operations (e.g., MPI_Allreduce and MPI_Barrier) or grouped point-to-point completions (e.g., MPI_Waitall) delineate OpenMP phase groups. No MPI primitives occur within an OpenMP phase group. MPI operations may include slack since the wait times of different tasks can vary due to load imbalance. Based on notions derived from critical path analysis, the *critical task* is the task upon which all other tasks wait.

The *configurations* of OpenMP phases of hybrid MPI/OpenMP applications can be adjusted dynamically. A *configuration* includes CPU frequency settings and concurrency con-

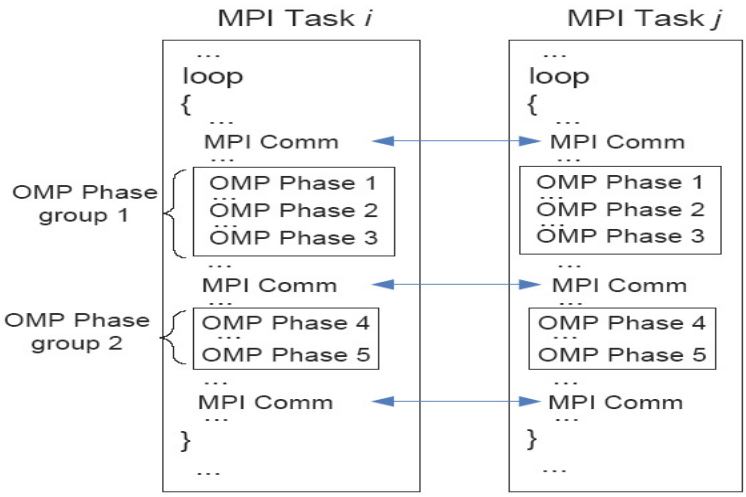


Figure 2.1: Typical MPI/OpenMP scheme in scientific applications

figurations. The *concurrency configuration* specifies how many OpenMP threads to use for a given OpenMP phase and how to map these threads to processors and cores. We can choose configurations through OpenMP mechanisms for controlling the number of threads and by setting the CPU affinity of threads using system calls. We use DCT and DVFS to adjust configurations so as to avoid performance loss while saving as much energy as possible. Also, configuration selection should have negligible overhead.

Significant effort has been invested in the study of the performance of the hybrid MPI/OpenMP programming model. Chow and Hysom attempt to understand the performance of hybrid MPI/OpenMP programs by describing performance using four parameters: multithreading efficiency, relative cache efficiency, network interface efficiency and message passing scaled efficiency. They use the parameters to abstract the characteristics of hybrid applications and their computing environments and presents results that are useful for the performance tuning of hybrid MPI/OpenMP. Some work [162] reveals that the hybrid MPI/OpenMP programming model can provide performance benefits for applications that have poor scaling with MPI processes due to load imbalance, a too fine grain problem size, memory limitations due to the use of a replicated data strategy, or a restriction on the number of MPI processes.

Ever since the birth of the hybrid MPI/OpenMP programming model, researchers have been studying the relative benefits of this programming model over traditional ones. Rane and Stanzione [150] compare the performance of pure MPI program and hybrid MPI/OpenMP programs. It was observed that beyond a certain number of cores the time spent in communication increased significantly in the pure-MPI case whereas it continued to decrease in the hybrid case. The hybrid program benefits from sending messages in an aggregated fashion and contributes lesser to message transfer overhead. Cappello and Etiemble [33] found that the superiority of one model (either pure MPI model or hybrid MPI/OpenMP) depends on the level of shared memory model parallelization, the communication patterns and the memory access patterns. The relative speeds of the main architecture components (CPU, memory and network) are of tremendous importance for selecting one model. The hybrid approach becomes better only when fast processors make the communication performance significant and the level of parallelization is sufficient. Drosinos and Koziris [54] tested the performance of nested loop algorithms with flow data dependencies with MPI and hybrid programming models. They conclude that hybrid programming models can match the architecture characteristics of an SMP cluster better under an appropriate processor mapping that would replace message passing communication with synchronized thread-level memory access.

Researchers have also developed tools and models to study the hybrid MPI/OpenMP programming model. Wolf and Mohr [187] provide a tool to present a multidimensional hierarchical decomposition of the search space for performance problems. With this tool, the performance analysis can be carried out along three interconnected dimensions: class of performance behavior, call tree and thread of execution. Their tool provides a complete and extensible solution for hybrid applications. Adhianto and Chapman [5] proposes a framework to predict the behavior of a hybrid application on a given target system for different problem sizes. They perform static analysis to retrieve an application signature

(such as computation loops and cache access pattern). They rely on runtime benchmarks to collect system profiles (such as communication latency and machine information). Then, they combine the application signature and system profile to predict performance.

The previous work provides important insights into the performance of the hybrid programming model. However, they do not consider power consumption into consideration. With the need to save power for high-end computing systems, it is important to consider both power and performance. In addition, the previous work does not adapt computation phases to multicore architectures. However it is possible to improve computation performance by appropriately throttling concurrency on multicores. Our work distinguishes from the previous work by leveraging DCT and DVFS to adapt program behaviors with both performance improvement and energy savings in mind.

2.2 Power-Aware MPI Programming Model

Power-aware MPI programming has attracted considerable research efforts. By predicting or modeling performance under different configurations (e.g., number of nodes and CPU frequency), we can find opportunities (e.g., slack time appearing in MPI communication or slow intra-node memory accesses) to save energy while avoiding performance penalty.

Exploring Energy-Time Tradeoff: Springer et al. [164] addressed the problem of finding a schedule that minimizes execution time on a power-scalable cluster with a maximum energy budget. They found a schedule through a combination of performance modeling, performance prediction and program execution. They later investigated the tradeoff between energy and performance in the NAS parallel benchmark suite [66]. They compared energy consumption and execution time across different numbers of nodes and studied via direct measurement and simulation both intra-node and inter-node effects of memory and communication bottlenecks. They found that energy saving is possible because of delays in

the processor, where executing at a high frequency and voltage does not make the program execute faster but instead wastes energy. The delays come from the processor waiting for memory accesses or blocking awaiting a message from a remote processor.

Optimizing Frequency Setting: Freeh et al. designed a framework [65] for executing a single application in several frequency-voltage settings. They divided programs into phases and then executed a series of experiments, with each phase assigned a prescribed frequency. During each experiment, they measured energy consumption and time and then used a heuristic to choose the assignment of frequency to phase for the next experiment. Rountree et al. [153] developed a system that determines a lower bound on the energy consumption of an application. They used mixed integer linear programming with the application communication trace and the cluster power characteristics as inputs and then output a schedule. The schedule listed how long each task should run in each frequency to realize the energy bound. This method, however, is near-optimal, because there is often no available single frequency that is low enough to remove all slack but not so low that the slowed computation impinges on the critical path. Also, the linear programming solver they used to generate the schedule was too costly to be used at runtime. They later proposed a runtime DVFS algorithm [152] that uses the proper MPI semantic level of granularity and split frequencies to minimize energy consumption. In order to estimate the performance of computation under different processor frequencies, they assume execution slowdown proportional to that of the change in frequency. For communication and memory-boundedness of a task, they record the observed slowdown and then refine the estimation. This performance prediction method may lose performance during the process of refining performance estimation. On the contrary, our work selects the frequencies based on an accurate prediction model that greatly reduces the potential of performance loss. In addition, the previous work implicitly assumes that putting the computation into lower frequencies always leads to energy saving. However this is not always true. Although we can avoid increasing total execution time of an

application by carefully slowing down frequencies in the computation portions of non-critical tasks, the execution times of those computation portions are extended. Since the energy is the product of time and power, an overall energy saving is only possible when the execution times of those computation portions do not increase too much such that the decrease in power consumption is counteracted. Therefore, we need to predict both execution time and power consumption to guarantee energy saving.

Leveraging load imbalance: Kappiah et al. [105] explored the potential for applying DVFS in the presence of load imbalance in parallel programs executed on power-scalable clusters. They monitored synchronization points in MPI (i.e., MPI communication library calls) to determine which node is the bottleneck at each communication point across iterations of the application. They then set non-bottleneck nodes to a lower DVFS level so that these nodes reached the synchronization point "just-in-time" and thereby reduced energy consumption.

Power-Aware Analytical Model: Ge et al. [72] introduced a power-aware speedup model to capture the interacting effects of parallelism and processor frequency analytically. They classified execution times according to CPU frequency scaling and parallelism effects. They then derived a new metric (power-aware speedup) to explain their effects. Ge et al. [73] further extended the power-aware speedup in the context of multicore based clusters. They included the number of allocated cores per computing node into the model and used a similar methodology to derive a new metric that captures the effects of both parallelism and processor frequency. Ge et al. [74] set up a model to divide a timestep (i.e. a program phase) into many small timeslices, the size of which depends on the current frequency. They then gathered performance counters for each timeslice and used past workload history to select a single frequency for the next timeslice. They used two algorithms to leverage past history for prediction. The first predicted that the workload at the next timeslice is the same as the one in the last timeslice; the second predicted the workload using both history values and

run-time profiling.

Most previous work usually focused on single programming models and does not consider how to save power or energy for the combination of multiple programming models. In addition, they did not consider potential energy-saving opportunities of aggregating multiple tasks in a node which are commonly met scenarios in modern high-end computing systems.

2.3 Performance Issues of Concurrency Throttling

With the proliferation of Simultaneous multithreading (SMT) and Chip-level multiprocessing (CMP), many researchers have applied concurrency throttling to optimize multithreaded codes on shared memory multiprocessors. Adaptive serialization [185] is one of the earliest attempts to study parallelization on shared memory multiprocessors. They considered the fork/join overhead of a parallel loop. They predicted sequential performance based on observed scalability properties of the machine and recorded loop execution time off-line. Then they compared the measured parallel loop time to the predicted serial time and chose the parallelization only when it was profitable. This work, however, did not consider the optimal number of threads and only considered using either one thread or the maximum concurrency level.

Zhang et al. [192] proposed a self-tuning OpenMP loop scheduler to react to behavior caused by inter-thread data locality, instruction mix, and SMT-related load imbalance. Their scheduler used sample executions of each possibility. Zhang et. al later improved their work by using a hardware-counter directed scheduler which samples the performance of parallel loops and quickly decided on a good scheduler using a decision tree that is created off-line [194]. They predicted an optimal number of threads for a given phase instead of predicting performance.

Suleman et al. [170] proposed two analytical models to control the number of threads at

runtime based on application behavior. One model captured the impact of data-synchronization on execution time. The other estimated the minimum number of threads required to saturate the bus. Their work took into consideration only two factors (data synchronization and memory bus contention) that impacted performance and ignored other important factors, such as cache contention.

Li et al. [121] applied concurrency throttling and DVFS on single chip multiprocessors to maintain performance while reducing power consumption. They conducted empirical searches of the configuration space to reduce the number of test executions necessary to perform adaptation. Their experiments were deployed based on simulation and ignored possible overheads when changing the configurations. On the contrary, our approach is implemented on a real system and considers all overheads.

A compiler-based approach [78, 103] controlled execution using a simple threshold-based strategy. The parallel code region is either sequentialized or run with a programmer-specified fixed number of threads. However, their approach could not provide the functionality to decide configurations using runtime information automatically. Our work provides autonomous concurrency throttling based on performance prediction of each configuration.

Most existing work leverages concurrency throttling based on performance estimations that may differ significantly from real execution time. In order to apply our DCT control algorithm, we sometimes need explicit execution time information to make scheduling decisions or estimate energy consumption. We extend previous work on DCT to obtain and to leverage accurate execution time information.

2.4 Power and Energy Estimation for Multicore Systems

There are four dominant approaches to estimate power and energy consumption: architecture-level simulation; hardware counter-based approaches; system information-based approaches; and direct measurement.

Simulation-based approaches [30, 61] usually perform power analysis by tracking, on a per-cycle basis, the usage and data activity of micro-architectural structures. Since they require detailed knowledge of the architecture (e.g., floorplan layout of the processor) and tend to be computation intensive, they are not good solutions to be applied at runtime.

Hardware counter-based approaches [96, 123, 186] leverage performance counters that can be configured to gather specific micro-architectural events, and these events closely approximate the number of accesses to each functional unit. They generally model the power consumption as a function of access rates of related functional units. Hardware counter-based approaches have less overhead than simulation-based approaches when applied at runtime. However, they require training with microbenchmarks to obtain model parameters. Due to differences between processor architectures, the model parameters and related function units for power estimation are different from one machine to another, which makes the model less flexible.

System information-based approaches collect information provided by the operating system to estimate system power and energy. Fan et al. [56] used CPU utilization to estimate the dynamic power usage. Economou et al. [55] leverage component utilization reported by the OS and CPU performance counters to model the power consumption of a blade. Heath et al. [81] developed linear models that include CPU, memory, and disk utilizations to estimate power. Horvath et al. [86] developed a power model that captures the effects of both CPU utilization and frequency setting. They claim that average system power and CPU

frequency exhibit near-linear relationships, when CPU utilization is within a limited range. They base this conclusion on empirical observations.

Direct measurement approaches [60, 75, 96] target direct, automatic profiling of power consumption at the component level of a server (e.g., processor, memory and disk). The authors proposed a methodology to separate component power after conversion from AC to DC current in the power supply of a typical server [60, 75, 96]. They automated data profiling, measurement and analysis by creating a tool suite called PowerPack [60, 75]. They mapped the power/energy consumption to application segments and exploited parallel performance inefficiencies characteristic of non-interactive distributed applications. This method is straightforward for estimating power consumption and provides power profiles for components for which power cannot be estimated with simulation-based and hardware counter-based approaches. However, this method must add extra circuits to measure DC current and must set up the mapping between current variance and component utilization. In addition, this method resorts to professional software (e.g., National Instrument Labview [94]) to collect power data from data acquisition systems (e.g., Analog Input Module NI 9205). Therefore, it is not convenient to feed power data to other runtime systems to control power consumption.

Most power and energy estimation methods have two drawbacks. First, they have too much overhead to be applied as an online method. Hence, they are not practical solutions for a real runtime system. Second, they usually implicitly assume that the configurations on a multicore, such as processor frequencies, are fixed. Hence, they cannot be applied for power-aware hybrid MPI/OpenMP models where we need to estimate power/energy consumption for various configurations. Our power estimation method is hardware-counter based and light weight. It can provide accurate power estimation without assuming fixed multicore system configurations.

2.5 Communication Performance Models

Communication performance models have been widely used to improve network topology, optimize algorithms, and evaluate system performance. They have been researched for many years and there are many successful models, such as PRAM [62], BSP [181], and LogP [45]. These models leverage software and hardware parameters to analyze and to estimate communication performance.

The Parallel Random Access Machine (PRAM) model consists of a set of processors that communicate through a global random access memory. It assumes that communication can be finished in unit time. This model does not capture the costs of contention, latency, and synchronization. Although it is used for parallel algorithm design [8, 9, 76], it is not practical for performance prediction. Several PRAM variants have been proposed [7, 42, 147].

The Bulk-Synchronous Parallel (BSP) model consists of a set of processors with local memory, routers that deliver point-to-point messages, and facilities for barrier synchronization. The BSP model defines a sequence of supersteps for the execution of a BSP algorithm. The superstep consists of local computation, communication, and is concluded by a barrier synchronization. The BSP model has a set of hardware parameters: the number of available processors p, the bandwidth inefficiency or gap g, and the time for barrier synchronization L. Therefore, the BSP model serves as a bridge between hardware and programming for parallel computation. Several BSP variants have been proposed, such as D-BSP [17] and BSP* [176].

The LogP model is another model to characterize communication performance. It defines four important parameters. L is the end-to-end latency from process to process, combining all contributing factors, such as copying data to and from network interfaces and the transfer over the physical network. o is the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message and cannot perform

other operations. g, the gap, is the minimum time interval between consecutive message transmissions or receptions. P is the number of processor/memory modules. This model is sufficiently detailed to reflect the major practical issues in parallel algorithm design, yet simple enough to support detailed algorithmic analysis. At the same time, the model avoids specifying the programming style or the communication protocol, being equally applicable to shared-memory, message passing, and data parallel paradigms.

LogP model has many variants that address its original limitations. LogGP [10] extends LogP to support sending of long messages by adding a new parameter G, defined as time per byte for a long message. The reciprocal of G characterizes the available per processor communication bandwidth for long messages. LoGPC [135] and LoPC [64] model network and resource contention by introducing a contention parameter C. The LogfP model [85] adds the new parameter f to stand for the number of consecutive small messages that can be sent for "free." The model targets hardware parallelism in current high performance networks like Infiniband. LognP and Log3P [31, 32] consider the impact of middleware on distributed communication. mlognP and 2log2,3P [179] abstract the memory hierarchy to consider both intra-node communication and inter-node communication.

The LogP model has been widely used to optimize algorithms [22, 44, 84, 21]. To use LogP, the parameters must be measured accurately. Many methods have been proposed to give fast and efficient measurements [20, 46, 89, 107, 125]. They usually leverage different communication patterns (e.g., ping-pong) to derive parameters.

The above communication performance models have not considered concurrent communication operations occurring on multicore systems. They do not model communication contention when aggregating multiple tasks per node. LoGPC and LoPC only model network contentions and do not consider resource contention in the memory hierarchy. In order to predict communication performance when communication contention exists, we must devise a new model or introduce new parameters.

2.6 Multicore Effects on Communication

One topic of this dissertation is to investigate effective task aggregation on multicore systems in terms of both power and execution time. One concern for task aggregation is how task aggregation on multicore systems affects communication performance. This section will review prior work on the subject. Most high-end computing systems use high performance networking, such as Infiniband (IB) [1, 12], Myrinet [27, 137] and the Quadrics network [4, 18]. High performance networking pays a lot of attention to design systems to accommodate multiple tasks per node. We will review related work on how high performance networking (in particular Infiniband) handles performance challenges caused by task aggregation.

Optimizing Intra-node Communication for Task Aggregation: Chai et al. [36] observed that cache and memory contention may be a potential bottleneck in multicore clusters. Communication middleware and applications should be multicore aware to alleviate this problem. They also found that on average about 50% of the messages are transferred through intra-node communication on multicore machines, and they indicated that optimizing intra-node communication is important. To optimize intra-node communication, they then allowed a kernel module to copy messages directly from one process's address space to another [101]. They further proposed to distinguish small and large messages and to handle them differently to minimize data transfer overhead for small messages and the memory space consumed by large messages [37]. They later generalized the implementation of intra-node communication to the two approaches [38]: user-level shared memory and kernel-assisted direct copy. They analyzed the advantages and limitations of the two approaches and proposed a hybrid approach that is aware of processor topology and process skew effects.

The above work provides important insights into intra-node communication on multicore clusters. However, only considering intra-node communication performance is not enough for evaluating the performance of task aggregation. Concurrent inter-node communication

can impact performance. We must take these effects into consideration when aggregating tasks.

Communication-Aware Task Mapping: As we distribute tasks across a variable number of nodes using different degrees of task aggregation per node, one of the problems we must address is how to map the tasks into nodes with communication performance in mind. Some research has been conducted on communication-aware task mapping.

Leng et al. [118] used the High Performance Linpack benchmark to demonstrate how task remapping can improve overall performance. However, they manually arranged the process mapping based on communication path characteristics and MPI message information, which is not a feasible solution for complex systems.

Orduna et al. [142, 143, 144, 145] explored communication-aware task mapping strategies that account for the communication requirements of parallel tasks and bandwidth in different parts of the network. They modeled the network resources as a table of costs for each pair of communicating processors. Their heuristic random search method attempted to identify the best mapping for a given network topology and communication pattern. Their work targeted heterogeneous inter-node networks and did not consider intra-node communication. Also, their evaluation only mapped one process to each processor. Thus, they did not account for potential interference between tasks executing communication operations on the same node.

Several research efforts have focused on communication sharing effects. Kim et al. [109] predicted communication delay with a linear model, taking into account the path sharing over a Myrinet network of workstations. They modeled communications through a piece-wise linear equation, and in the case of path sharing, this equation is multiplied by the maximum number of communications within the sharing conflict. However, their study was based on the GM and BIP network protocols, unlike ours, which focuses on MPI. Also, they did not consider intra-node communication. Martinasso et al. [100, 126, 127] introduced a notion of resource sharing within communication patterns. They decomposed a chain of

sources of communication interference into several elementary sources of interference and then predicted communication time by the flow cut of each source. They evaluated their model on SMP clusters with dual processors. Nodes with more processors and complex core layouts introduce complications in resource sharing that have not been considered in their work. Also, their model required careful analysis to decompose communication into sources of interference manually, which makes it infeasible for complex communication patterns.

Scogland et al. [155] and Narayanaswamy et al. [138] demonstrated that the interactions between multicore architectures and communication stacks can result in asymmetry in the effective capabilities of the different cores. They presented a software stack to monitor such interactions and to manage the mapping of processes onto processor cores dynamically to improve performance. Jang et al. [99] proposed a networking process scheduling system for multicore systems with multiple network interfaces. It found an optimal processor affinity based on the processor cache layout, communication intensiveness and processor loads. Trahay et al. [177] exploited idle cores to speed up communications in two ways: moving CPU intensive operations out of the critical path, and letting rendezvous transfers progress asynchronously.

Bhatele et al. [23, 24, 25, 26] conducted a series of studies on network resource contention. They demonstrated that network topology is still important in large supercomputers, although virtual cut-through, wormhole routing and faster interconnects are deployed. They studied message latencies with typical communication patterns on torus and mesh networks. They further proposed topology-aware task mapping to avoid link occupancy or contention. Their work does not consider intra-node communication and does not consider resource contention on machine nodes.

Most previous work has two major drawbacks: they usually did not have a comprehensive analysis on both intra-node and inter-node communications on large scales; and their evaluations were usually deployed on machines with a small number of cores. For machines

with more cores, their work is usually too complex to be applied.

High Performance Networking with Multicore Systems: The high performance networking community has done a lot of work to handle multicore challenges (e.g., more severe resource contention and more concerns about scalability in MPI implementation).

IB network contention on multicore machines has been a research topic for years. Koop et al. [111] profiled IB messages at different time slices and viewed how different applications that reside in the same node interact. They stated that since many applications use synchronous in communication, all processes within a job will typically require simultaneous access to the network and thus create a significant bottleneck. Therefore, they proposed mixing different application workloads to break synchronization between processes on the same node and thus to reduce network contention. Subramoni et. al [169] explored QoS provisioning for IB. They used inter-job and intra-job bandwidth partitioning by use of multiple service levels and effective use of NIC level buffers by utilizing multiple virtual lanes for data transmission. Their experiments showed significant performance improvement in applications with small messages.

Open MPI [140] and MVAPICH [88] are the most widely used IB MPI implementations. Both attempt to make the implementations scalable to future HPC systems of thousands or millions of nodes, as more nodes with more processors per node and more cores per processor are deployed in modern high-end computing systems. To improve the scalability of their IB implementation, Open MPI can use a connection-less unreliable datagram transport [70] that allows near-constant resource usage, unlike per-process resource usage that grows linearly with the number of processes in a connection-oriented transport. To avoid receive buffer depletion, which negatively impacts performance, and improve receive buffer utilization, Open MPI uses buckets of receive buffers of different sizes, each bucket using a single shared receive queue (called B-SRQ) [156]. This method matches receive buffer sizes with the size of received data better than the previous shared received queues (SRQ). Shipman et al. [157]

further proposed X-SRQ to improve scalability and performance of IB on multicore machines. X-SRQ is based on the recently developed IB techniques: a single queue pair (QP) can be associated with multiple SRQs by specifying the SRQ for delivery on the send side. The X-SRQ allows each process to maintain a single send QP to each host rather than to each remote process. A receive QP is established per remote send QP. These receive QPs can be shared among all the processes on the host. This mechanism significantly reduces memory footprint and improves performance due to better cache context resource data and thereby avoids a lookup on host memory.

The research on high performance networking mentioned above typically leverages existing network protocols and hardware features (e.g., QoS and extended reliable connection for X-SRQ), or application characteristics (e.g., mixing workload to break synchronization between processes with the assumption that application characteristics are known) to reduce contention when tasks are aggregated. They cannot dynamically adjust system resources based on application characteristics. However, it is possible to manage system resources at runtime to reduce resource contention, given the prediction of application characteristics.

2.7 Memory Registration Optimization

Many methods to optimize memory registration and deregistration in communication protocols have been proposed. Woodall et al. [188] described a pipeline protocol to overlap memory registration with RDMA operations. Their work changes the RDMA communication protocol, while our work is independent of the RDMA communication implementation. Zhou et al. [195] combined memory pinning and allocation to eliminate pinning and unpinning from the registration and deregistration paths. They also batch deregistrations to reduce the average cost. Tezuka et al. [173] and Farreras et al. [57] proposed a pin-down cache for Myrinet. They delay deregistration and cache registration information for future accesses to

the same memory region. Bell and Bonachea [19] proposed an RDMA algorithm for a shared memory system that determines the largest amount of memory that remote machines can share and then registers all shared memory regions and links them to a firehose interface, to which remote machines can write and read at any time. Unlike our scheme, the above work, when applied to reduce registered memory sizes, cannot completely remove registration and deregistration overhead from the communication critical path.

Shipman et al. [156] investigated network buffer utilization and introduced a new protocol to increase the efficiency of receiver buffer utilization for Infiniband. The protocol uses buckets of receive buffers of different sizes, instead of consuming buffers regardless of the actual incoming message size. They target optimization of bounce buffers, while our work aims to reduce registered user buffer sizes. We register memory based on the real size of user buffers instead of a matched bucket size of network buffers.

2.8 Exploring the Predictability of HPC Applications

Many prior efforts have studied common communication patterns in HPC applications [29, 108, 184, 191]. HPC applications often exhibit spatial locality [108] (i.e., a small number of communication partners) and temporal locality [69] (i.e., iterative patterns). Researchers have proposed prediction techniques to leverage these characteristics to optimize communication performance. Afsahi et al. [6] propose several heuristics to predict MPI activity. They use these heuristics to cache incoming messages before posting the corresponding receive calls. Freitag et al. [68, 69] propose a periodicity-based predictor based on the computation of distances between patterns in data streams. They apply the predictor to prepare the receiver memory to guarantee enough resources for message reception. Other work leverages time-delay neural networks and Markov models [154] to detect memory access patterns. These methods tend to require long training times. Ratn et al. [151] enable timed replay

of communication events by distinguishing histograms with call stack and path sequence. Their approach does not consider the runtime overhead due to the approach's nature of performance analysis, while the overhead can be comparable to communication time.

Our work has two primary differences from prior work. First, we use a a time-based predictor. Time information is important for prediction-based memory management in order to meet registration deadlines while minimizing registered memory sizes. Prior work focuses on iterative memory access patterns while ignoring time information, thus missing opportunities to reduce registered memory sizes. Second, our predictor is lightweight and has a short training process, which makes it appropriate for online optimization.

This page intentionally left blank.

Chapter 3

Model-Based Hybrid MPI/OpenMP Power-Aware Computing

As the number of cores on processors increases exponentially, high-end computing systems built with multicore processors begin to provide an unprecedented degree of thread-level parallelism to applications. To exploit many cores available on a single chip or a single node with shared memory, programmers tend to use programming models with efficient implementations on cache-coherent shared-memory architectures, such as OpenMP. At the same time, programmers tend to use programming models based on message passing, such as MPI, to execute parallel applications efficiently on clusters of compute nodes with disjoint memories. We expect that hybrid programming models, such as MPI/OpenMP, will become more popular, because the current high-end computing systems are scaled to a large amount of nodes, each of which is equipped with processors packing more cores. The hybrid programming models can utilize both shared memory and distributed memory of high-end computing systems.

Hybrid programming models are implemented by multiple runtime systems, each implementing one aspect of parallel execution. The lack of coordination between the runtime

systems and the contention for common hardware resources are known sources of inefficiency in hybrid programming models. In this chapter, we are concerned in particular with the problem of energy inefficiency in runtime systems that implement hybrid programming models. While prior research has proposed several methods to reduce dynamic power consumption while sustaining performance in a single runtime system implementing a single programming model [49, 105, 153, 164], no methods have been proposed to improve coordinated power management methods across the runtime systems that implement a hybrid programming model. We propose methods to make hybrid parallel programming models power-aware.

Recent solutions to software-controlled power-aware execution of HPC applications exploit different characterizations of slack. Slack can arise from an imbalanced workload between tasks, differences in the scalability of parallel workloads on a given architecture, different node capabilities, communication latency or any type of non overlapped latency. Software-controlled power management algorithms leverage slack by dilating computation into slack intervals. Many state-of-the-art algorithms dilate computation into slack time that occurs between MPI communication events, using dynamic voltage and frequency scaling (DVFS) [87, 105, 133]. Similar algorithms have been proposed for exploiting slack in OpenMP, at barriers or other global synchronization operations [53].

Dynamic Concurrency Throttling (DCT) [48, 170] is an alternative software method for power-aware execution of HPC applications, best suited for shared-memory architectures. DCT is applied in shared-memory programming models like OpenMP, by controlling the number of active threads executing regions of parallel code. Concurrency control can tune the power and performance of a parallel code region simultaneously [47]. Interestingly, since the scalability of different code regions may vary significantly due to lack of concurrency or system bottlenecks (e.g., memory bandwidth), reducing concurrency of a region of parallel code can often reduce execution time while simultaneously reducing power consumption [47]. On the downside, DCT is difficult on distributed-memory execution environments, as tuning

concurrency requires on-the-fly reshaping of applications and complex data redistribution and process migration operations.

In this chapter, we integrate DCT and DVFS in the runtime systems of the hybrid MPI/OpenMP programming mode, to expose and to leverage energy saving opportunities hidden in parallel applications. Such an attempt faces many challenges. Since hybrid programming models involve interactions between tasks, imposing an energy-saving strategy on one task may have a negative impact on other tasks. We investigate this impact and propose solutions to avoid any potential performance loss. Furthermore, the parameter space available for energy savings in hybrid programming models is large, including one dimensional (only DCT, or only DVFS) and two dimensional (DCT+DVFS) solutions. These choices complicate the design of energy saving algorithms, especially with respect to its scalability.

In section 2.1, we review the terminology of the hybrid MPI/OpenMP programming model. In this chapter, we will present a novel power-aware hybrid MPI/OpenMP runtime system. The runtime system adapts the execution of hybrid parallel applications using DCT and DVFS, to leverage energy saving opportunities while minimizing the impact of energy saving actions on performance.

The rest of this chapter is organized as follows. We discuss applying DCT and DVFS in sections 3.1 and 3.2. We present and compare three strategies of applying DCT and DVFS in section 3.3. These strategies require execution time and power predictions. We present methods to predict execution time and power in sections 3.4 and 3.5 respectively. We provide implementation details in section 3.6 and present performance results in section 3.7. We conclude in section 3.8.

3.1 Performance Implications of Concurrency Throttling

DCT can reduce dynamic power consumption by putting cores in a low-power state, when they are not needed or when they do not benefit application execution time. In the latter case, DCT may also save execution time by alleviating contention for shared resources. The use of uncoordinated DCT in the OpenMP phases of multiple MPI tasks may introduce load imbalance between tasks, thereby creating more slack than the slack that already exists in the application due to load imbalance at communication waiting points. In this section, we study how to apply DCT effectively in hybrid MPI/OpenMP applications.

To guide the discussion, we run the AMG benchmark from the ASC Sequoia Benchmark suite [113] on two nodes, each of which has four AMD Opteron 8350 quad-core processors. AMG has four OpenMP phases in the computation loop of its solve phase. Phases 1 and 2 are in phase group 1, phases 3 and 4 are in phase group 2, and the phase groups are separated by an `MPI_Waitall` primitive. We describe AMG in detail in Section 3.7. We run the benchmark with input parameters P=[2 1 1], n=[512 512 512] with two tasks. We place OpenMP phases under different concurrency configurations and profile them. We plot the results in Figure 3.1, which indicates the execution time of the fastest configuration for each task and OpenMP phase with stripes. The results display varied scalability across phases and even within the same phase when executed in different MPI tasks, due to differences in workload and data sets.

We further run the benchmark under a fixed configuration throughout the execution of all OpenMP phases in all tasks for the entire duration of the run. We then manually select the best concurrency configuration based on these static observations. The configuration using 4 processors and 2 threads per processor on each node, shown as the first bar in each group of bars in Figure 3.2, has the lowest total time in the solve phase and, thus, is the

Figure 3.1: AMG phase performance

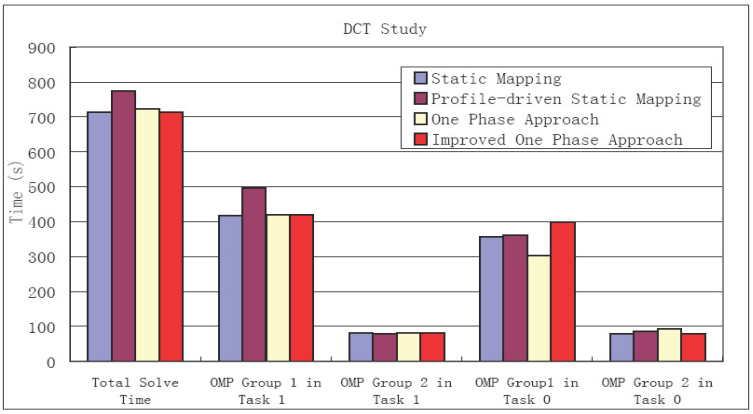


Figure 3.2: Impact of different DCT policies on AMG

best static mapping that we use as our baseline in the following discussion.

3.1.1 Profile-Driven Static Mapping

Under the best static mapping (i.e., a single concurrency configuration for all OpenMP phases in the program), each individual OpenMP phase may not use its best concurrency configuration. Intuitively, using the best concurrency configuration for each OpenMP phase should minimize the computation time of each MPI task. We call this DCT strategy the profile-driven static mapping. We apply this strategy based on the results of Figure 3.1 and rerun the benchmark. The second bar in each group of bars in Figure 3.2 shows the execution time of the profile-driven static mapping, which is worse than the performance of the best static mapping. To determine the source of performance loss under this strategy, we profile each OpenMP phase with the profile-driven static mapping and the best static mapping. Figure 3.3 shows the results. Three of the four OpenMP phases lose performance with the profile-driven static mapping, even though each uses the best concurrency configuration based on the fixed configuration runs. This loss arises from frequent configuration changes from one OpenMP phase under the profile-driven static mapping. An OpenMP phase may incur additional cache misses if the previous OpenMP phase uses a different configuration.

The experimental results shown in Figure 3.3 confirm that cache interference is the problem. The charts show the last-level cache misses that each OpenMP phase incurs, normalized to the number of last-level cache misses with the static mapping. Combined with frequent configuration changes, we observe a significant increase in cache misses.

Previous work [47, 48] showed that the profile-driven static mapping can outperform the best static mapping. These results combine with ours to demonstrate that the profile-driven static mapping has no performance guarantees: it benefits from improved concurrency configurations while often suffering additional cache misses. We propose a scheme to solve this problem.

3.1.2 One Phase Approach

To avoid cache misses caused by changing configurations, a simple solution is to combine all OpenMP phases in a task and regard them as one combined phase. We can then select the configuration that minimizes the time of the combined phase in future iterations. Under this DCT scheme, which we call the *one phase approach*, all OpenMP phases run with the same concurrency configuration. We apply this strategy to AMG and, as depicted in Figure 3.2, it greatly reduces the performance loss compared to the profile-driven static mapping. Figure 3.3 shows that cache misses are also reduced significantly. However, we still incur significant performance loss compared to the best static mapping. Further performance analysis reveals that the one phase approach leads to a sub-optimal configuration in OpenMP phase groups (collections of phases delineated by MPI operations), dispite minimizing the time across all OpenMP phases.

We use our AMG results to explain this observation further. In the case of AMG under the best static mapping (configuration (4,2)), the time in OpenMP phase group 2 of task 0 is shorter than the corresponding one in task 1, as shown in Figure 3.2. Task 1 therefore

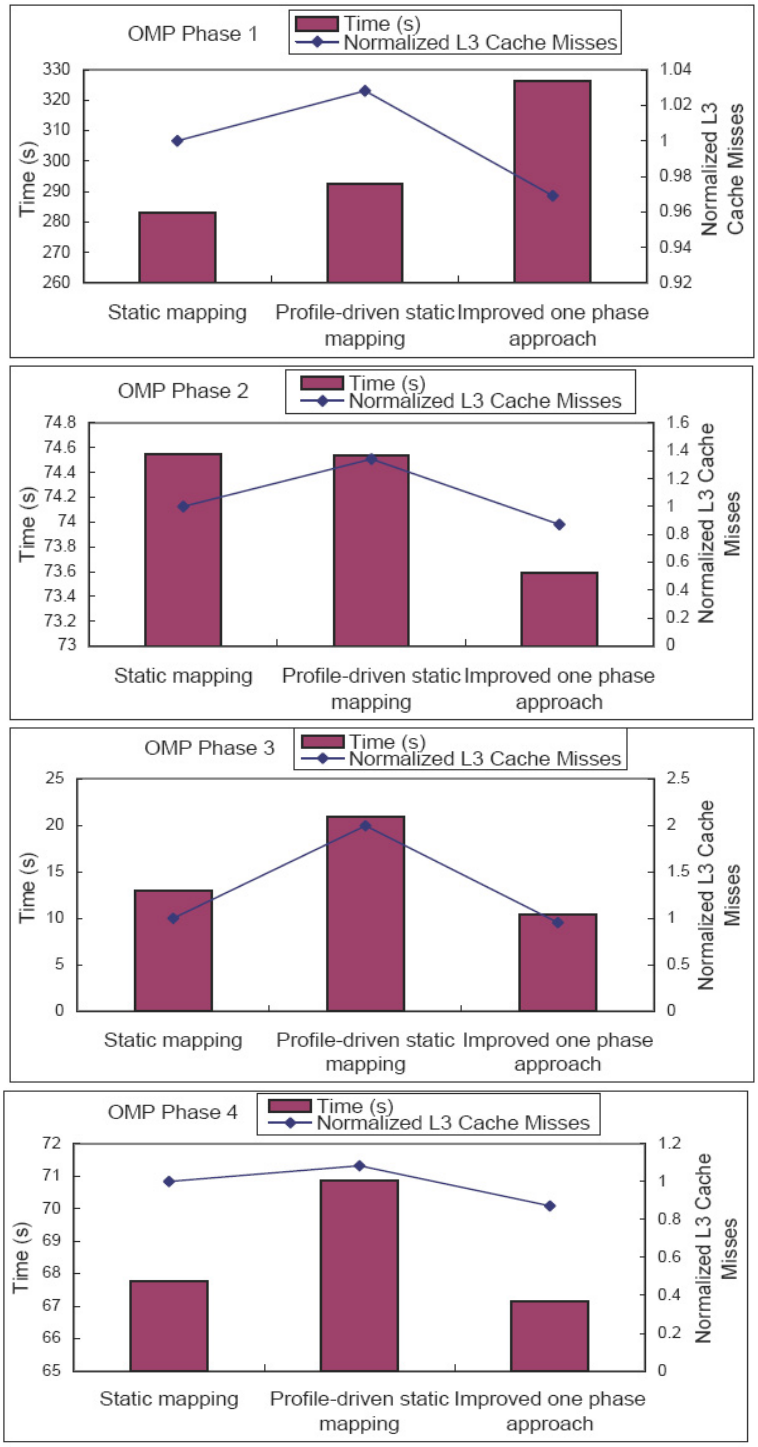


Figure 3.3: Phase profiles of task 0 under DCT policies

becomes the critical task in OpenMP phase group 2. Applying the one phase approach, task 1 uses configuration (4,2) and achieves the same performance as with the best static mapping, while task 0 uses the configuration (3,4) where the time in group 2 is longer than that in task 1. Consequently, task 0 now becomes the critical task in group 2, which leads to the increase of total computation loop time.

This problem arises because configurations are selected without coordination between tasks. Instead, each task greedily chooses the best configuration for each combined phase regardless of the global impact. Under our *improved one phase approach*, each task considers the time at the critical task when making its DCT decisions. Each task selects a configuration that does not make its OpenMP phase groups longer than the corresponding ones in the critical task. Although this strategy may result in a configuration in which performance for a task is worse than the one achieved with the best static mapping, it maintains performance as long as the OpenMP phase group time is shorter than the corresponding one in the critical task. Unlike the profile-driven static mapping, this strategy has a performance guarantee: it selects configurations that yield overall performance no worse than the best static mapping, as Figure 3.2 shows.

To summarize, the profile-driven static mapping adjusts configurations at a fine granularity and suffers from the performance impact of cache interference between adjacent OpenMP phases. The one phase approach, on the other hand, does concurrency throttling at the coarsest granularity, thus ignoring that particular OpenMP phases may miss opportunities to execute with better configurations. The improved one phase approach strives for a balance between the two approaches by introducing task coordination and considering performance at a medium granularity (OpenMP phase groups).

3.2 Formalization of the Frequency Scaling Problem

The workloads across tasks may not finish within the same execution time, because (1) the input problem may divide unevenly by all tasks; (2) scalability differences of workloads across tasks; and (3) a heterogeneous computing environment. Using DVFS, we disperse the slack time to OpenMP phases by extending the execution times of computation phases of non-critical tasks, to reduce the slack time and energy consumption. DVFS methods disperse slack time that appears in non-critical tasks during parallel execution due to load imbalance, to reduce power consumption. We use DVFS to disperse slack time to OpenMP phases within MPI tasks, by selecting appropriate core frequencies for the cores that execute each OpenMP phase. We assume for simplicity and to avoid introducing load imbalance that all cores that execute an OpenMP phase are set to run at the same frequency during the phase.

We formulate this problem as a knapsack problem. We define the time of each OpenMP phase under a particular core frequency f_k as an *item*. Each item is associated with a weight, w, which is the time change under frequency f_k compared to running at peak frequency. Each item is also associated with a value, p, which is the energy consumption under frequency f_k. The optimization objective is to keep the total weight of all phases under a given limit, which corresponds to the slack time Δt^{slack}, and minimize the total value of all phases. This is a 0-1 knapsack problem [159], which is NP-complete. In our case, we further require that some items cannot be selected at the same time. This requirement arises since we assume that we cannot select more than one frequency for each OpenMP phase. Thus, we must solve a variant of the 0-1 knapsack problem.

Dynamic programming can solve the knapsack problem in pseudo-polynomial time. If each item has a distinct value per unit of weight ($v = p/w$), the empirical complexity of this kind of problem is $O((log(n))^2)$, where n is the number of items. In our case, we design a

unique dynamic programming solution to our problem. For convenience in its description, we replace p with $-p$ to solve the problem of maximizing the total value. Let L be the number of available CPU frequency levels, $w_{(i-1)\cdot L+1}$, $w_{(i-1)\cdot L+2}$, $\cdots$, $w_{L\cdot i}$ be the available weights of OpenMP phase i, and $p_{(i-1)\cdot L+1}$, $p_{(i-1)\cdot L+2}$, $\cdots$, $p_{i\cdot L}$ be the available values of OpenMP phase i. The total weight limitation is W (i.e., the available slack time). We denote the maximum attainable value with weight less than or equal to Y using items up to j as $A(j, Y)$.
$A(j, Y)$ is defined recursively as:

$$A(0, Y) = -\infty, \qquad A(j, 0) = -\infty \tag{3.1}$$

if all w_{j-L+1}, $\ldots$, w_{j-1} are greater than Y, then

$$A(j, Y) = A(j - L, Y) + p_j; \tag{3.2}$$

else, for any $i \in [j - L + 1, j - 1]$, and $w_i \leq Y$

$$A(j, Y) = \max\left(A(j - L, Y) + p_j, \max_i(p_i + A(j - L, Y - w_i))\right) \tag{3.3}$$

We solve the problem of choosing frequencies by calculating $A(n, \Delta t_i^{slack})$ for task i, where n is the number of items. For a given total weight limitation W, the empirical complexity is still $O((log(n))^2)$.

3.3 Adaptation Strategies for Concurrency Throttling and Frequency Scaling

We can apply DCT and DVFS in order or simultaneously to obtain energy savings from both knobs. In this section, we discuss the alternatives for combining DCT with DVFS in more detail and formalize the execution behaviors and energy saving opportunities. Table 3.1 summarizes the notation used in our formalization.

M	Number of OpenMP phases in a phase group
N	Number of MPI tasks
t_i	Time for OpenMP phases in task i
C	Set of possible concurrency configurations
t_{ijc_i}	Time for OpenMP phase j of task i under concurrency configuration c_i
t_c	Critical time
t_{dvfs}	DVFS overhead
$t_i^{comm\_send}$	Time to send data from task i
t_{ijf_j}	Time for OpenMP phase j of task i under frequency setting f_j
P_{ijf_j}	Average system power for OpenMP phase j of task i under frequency setting f_j
f^{max}, f^{min}	Maximum and minimum frequency settings
c^{max}	Maximal concurrency configuration
P_{ijc_i}	Average system power for OpenMP phase j of task i under concurrency configuration c_i
$t_{ijc_if_j}$	Time for OpenMP phase j of task i under concurrency configuration c_i and frequency settng f_j
$P_{ijc_if_j}$	Average system power for OpenMP phase j of task i under concurrency configuration c_i and frequency setting f_j

Table 3.1: Power-aware MPI/OpenMP model notation

3.3.1 DCT First

With this strategy, MPI tasks first apply DCT, using our improved one phase approach to reduce cache pollution. The computation time of OpenMP phases in MPI task i should also be minimized. We model this DCT approach as:

$$t_i = \min_{c_i \in C} \sum_{j=1}^{M} t_{ijc_i} \tag{3.4}$$

where c_i is the concurrency configuration for task i and C is the set of all possible configurations on one node.

The critical task has the longest execution time, the *critical time* in the OpenMP phase group. We model it as:

$$t_c = \max_{1 \le i \le N} (\min_{c_i \in C} \sum_{j=1}^{M} t_{ijc_i}) \tag{3.5}$$

We compute the slack for task i (Δt_i^{slack}) with Equation 3.6. We reduce the available slack by the DVFS overhead and the time to send data from task i in order to avoid reducing the frequency too much. Figure 3.4 illustrates these constraints.

$$\Delta t_i^{slack} = t_c - t_i - t_i^{comm\_send} - t_{dvfs} \tag{3.6}$$

After applying DCT to determine the concurrency configurations, we use communication to provide each non-critical task with the available slack. We then apply DVFS in these tasks so as to ensure that the selected frequency satisfies our time constraint (Equation 3.7). Δt_{ijf_j} is change in time to execute phase j on task i at frequency f_j, compared to the time to execute it at the maximum frequency setting f_{max}, which we use as the default during the application of DCT. Our time constraint ensures that the total time change on task i does

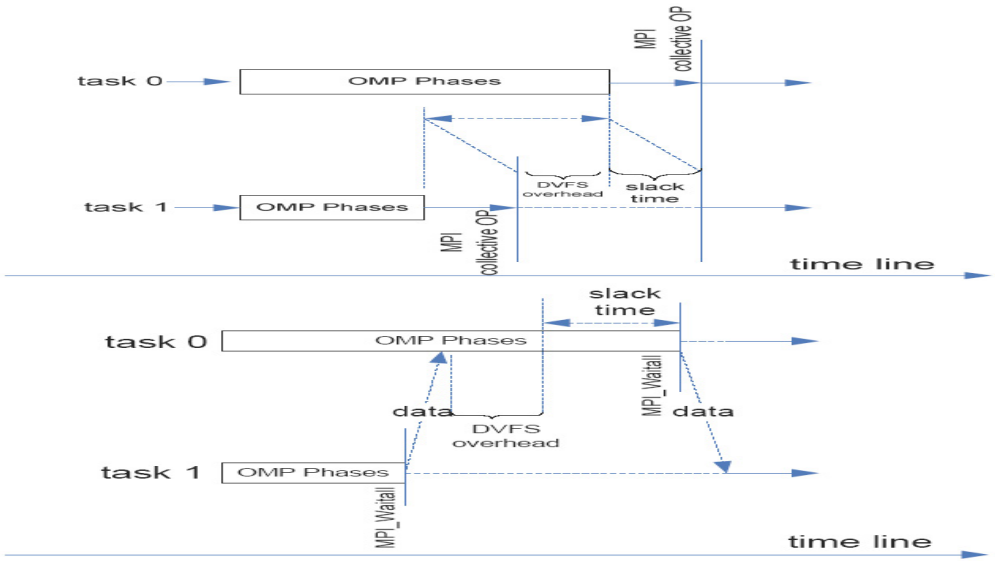


Figure 3.4: Leveraging slack to save energy with DVFS

not exceed the available slack.

$$\sum_{1 \leq j \leq M} \Delta t_{ijf_j} \leq \Delta t_i^{slack} \tag{3.7}$$

We also have an energy constraint (Equation 3.8) for DVFS that limits the energy consumption with the selected frequencies by the energy consumption at the maximum frequency.

$$\sum_{1 \leq j \leq M} P_{ijf_j} t_{ijf_j} \leq \sum_{1 \leq j \leq M} P_{ijf^{max}} t_{ijf^{max}} \tag{3.8}$$

Under our time and energy constraints, we minimize the total energy consumption (E_i) with DVFS:

$$E_i = \min_{f^{min} \leq f_j \leq f^{max}} \sum_{1 \leq j \leq M} P_{ijf_j} t_{ijf_j} \tag{3.9}$$

3.3.2 DVFS First

With this strategy, we use communication to provide each non-critical task with the available slack before we apply DCT. Following the requirements of Equations 3.7– 3.9, we then apply DVFS in these tasks. At the same time, the critical task performs DCT and communicates the new slack time to the non-critical tasks. The non-critical tasks then perform DCT using the new slack, following the constraints of Equations 3.10– 3.12. Equations 3.10– 3.12, in which c^{max} is the maximal concurrency configuration (i.e., a thread running on each core of each processor of the node) are similar to Equations 3.7– 3.9, except that they consider concurrency configurations (c_i) instead of frequency settings (f_j).

$$\sum_{1 \leq j \leq M} \Delta t_{ijc_i} \leq \Delta t_i^{slack} \tag{3.10}$$

$$\sum_{1 \leq j \leq M} P_{ijc_i} t_{ijc_i} \leq \sum_{1 \leq j \leq M} P_{ijc^{max}} t_{ijc^{max}} \tag{3.11}$$

$$E_i = \min_{c_i \in C} \sum_{1 \leq j \leq M} P_{ijc_i} t_{ijc_i} \tag{3.12}$$

3.3.3 Simultaneous DCT and DVFS

With this strategy, we again initially use communication to provide non-critical tasks with the available slack. We then simultaneously apply DCT and DVFS in each task. We then use additional communication to provide the non-critical tasks with the new slack. We then simultaneously readjust the DCT and DVFS settings of these tasks.

The configurations of the critical task follow Equation 3.13, which minimizes execution time. The configurations of the non-critical tasks follow Equations 3.14– 3.16, which minimize energy consumption (Equation 3.16) subject to our new time (Equation 3.14) and energy (Equation 3.15) constraints. We compute the slack time from Equation 3.6.

$$t_i = \min_{c_i \in C, f^{min} \leq f_j \leq f^{max}} \sum_{j=1}^{M} t_{ijc_if_j} \tag{3.13}$$

$$\sum_{1 \leq j \leq M} \Delta t_{ijc_if_j} \leq \Delta t_i^{slack} \tag{3.14}$$

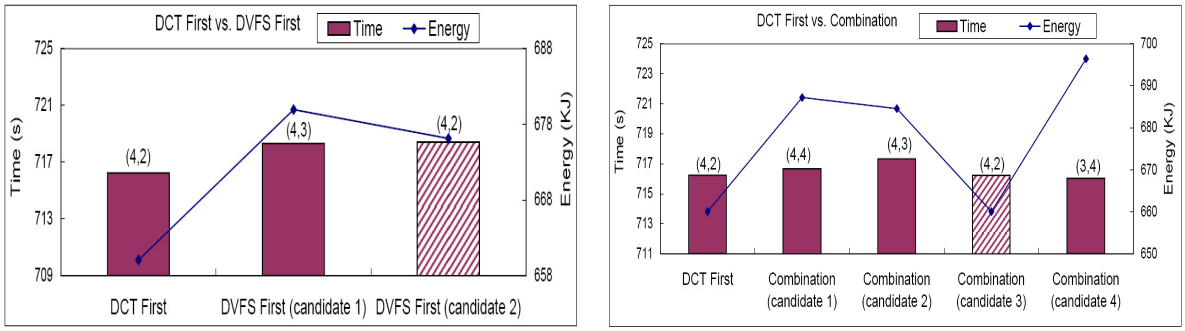
$$\sum_{1 \leq j \leq M} P_{ijc_if_j} t_{ijc_if_j} \leq \sum_{1 \leq j \leq M} P_{ijc^{max}f^{max}} t_{ijc^{max}f^{max}} \tag{3.15}$$

$$E_i = \min_{c_i \in C, f^{min} \leq f_j \leq f^{max}} \sum_{1 \leq j \leq M} P_{ijc_if_j} t_{ijc_if_j} \tag{3.16}$$

3.3.4 Performance

We perform a set of tests to investigate how the three strategies affect energy saving and performance. All tests are executed on two nodes, each of which has four quad-core processors (AMD Opteron Processor 8350) with five feasible frequency settings. We measure the energy consumption of the two nodes with a Watts Up Pro ES power meter. We apply the three strategies for combining DCT with DVFS to the AMG benchmark. To study how effectively DCT and DVFS contribute to energy savings, we display feasible configurations that satisfy the energy and time constraints but do not necessarily satisfy Equations 3.9, 3.12 and 3.16 (i.e., minimizing energy consumption). We name these configurations *candidates*. The best candidate (i.e., the one with the minimum energy consumption) is marked with stripes.

Figure 3.5(a) compares the results of using DCT first or DVFS first. The numbers above the bars in the figure refer to the DCT configuration. For example, "(4,2)" refers to running on four processors and two cores per processor. The results show that the two DVFS candidates consume more energy than the execution with DCT first. Candidate 2, despite using the same DCT configuration as DCT first, consumes more energy, because it has a



(a) (b)

Figure 3.5: Comparison of three adaptation strategies

higher frequency setting. The strategy of DVFS first chooses the frequencies according to the slack time before using DCT. In our case, this slack time is shorter than slack time after concurrency throttling in the DCT first strategy, which leads to higher frequency settings. We also derive that candidate 2 does not fully leverage the available slack. Unless we re-adjust the frequency, the slack time cannot be fully leveraged to save energy by DVFS first. Candidate 1 has a DCT configuration having a slower execution than that of candidate 2. Since candidate 1 also occupies more processor cores than the candidate 2, it consumes more energy than candidate 2.

Figure 3.5(b) compares the results of using DCT first and the combined strategy. We notice that DCT first and the combined strategy choose the same best configuration. Candidate 4 is an interesting case. The DCT configuration of candidate 4 is (3,4), which leads to slower execution than that of candidate 3 with configuration (4,2), while the frequency setting of candidate 4 is higher than that of candidate 3. In other words, to extend the execution time of the non-critical task to cover a fixed slack time, candidate 4 uses both DCT and DVFS, while candidate 3 uses only DVFS (DCT does not extend execution time). Since candidate 4 consumes more energy than candidate 3, we conclude that DVFS is more efficient in energy saving than DCT for this workload.

3.3.5 Discussion

Section 3.3.4 shows that a combined strategy for applying DCT and DVFS simultaneously saves the most energy, which is however as good as DCT first. The results in Section 3.7 also show that the combined strategy never performs worse than the other two strategies. If we regard the configuration space as a 2D plane with DCT configuration as one dimension and DVFS configuration as the other, DCT first (DVFS first) chooses a DCT (DVFS) configuration (i.e., chooses a line in the 2D plane) and then chooses the frequency setting (concurrency setting) along this line. The best configuration may be a blind spot that cannot be achieved by either DCT first or DVFS first. The combination strategy has the freedom of choosing any point in the plane, thus avoiding the blind spot.

In fact, neither DCT first nor DVFS first can guarantee the optimal configuration. After DCT, slack time can either be extended or reduced. If slack time is extended, then DCT first could save more energy than DVFS first, because more slack time can be leveraged by setting the frequency after DCT. If slack time is shrunk by Δt, the energy saving potential for DVFS is reduced. If frequency setting is more effective in reducing energy than concurrency throttling, then DVFS first can save more energy than DCT first.

The modeling described thus far is based on the assumption that the runtime system knows the execution times and power consumption of OpenMP phases with different configurations. In the following two sections, we will describe how we can predict execution time and power consumption in OpenMP phases.

3.4 Execution Time Modeling for OpenMP Phases

We presented models that need execution time to compute slack time (Equation 3.6) and to avoid performance loss (Equation 3.14); execution time is also needed to compute energy consumption and to determine if energy constraints are met (Equation 3.15) and energy

consumption is minimized (Equation 3.16).

We present an execution time predictor that extends previous work on IPC (Instructions Per Cycle) prediction for DCT [47, 48, 49]. This prior work has shown that to apply DCT effectively, it is sufficient to predict a correct ranking of DCT configurations for parallel execution phases, even if the absolute IPC predictions for each phase are inaccurate. Mere ranking is insufficient in our case, as we need to estimate slack time in order to apply our DCT and DVFS policies. We use execution samples collected at runtime on specific configurations to predict the time on other, untested configurations. From these samples, our predictor learns about each OpenMP phase's execution properties that impact the time under alternative configurations.

The input from the sample configurations consists of elapsed CPU clock cycles and a set of n hardware event rates ($e_{(1\cdots n,s)}$) observed for the particular phase on the sample configuration s, where the event rate $e_{(i,s)}$ is the number of occurrences of event i divided by the number of elapsed cycles during the execution of configuration s. The event rates capture the utilization of particular hardware resources that represent scalability bottlenecks, thus providing insight into the likely impact of hardware utilization and contention on scalability. We choose which events to use with our model based on correlation analysis. In particular, we determine which event rates on the sample configurations are most strongly correlated with execution time. Then we select the top n events from the sorted list.

The model predicts time on a given target configuration t, which we call $Time_t$. This time includes the time spent within OpenMP phases plus their parallelization overhead.

For an arbitrary collection of samples, S, of size $|S|$, we model $Time_t$ as a linear function:

$$Time_t = \sum_{i=1}^{|S|} (Time_i \cdot \alpha_{(t,i)}(e_{(1\cdots n,i)})) + \lambda_t(e_{(1\cdots n,S)}) + \sigma_t \tag{3.17}$$

The term λ_t is defined as:

$$\lambda_t(e_{(1\cdots n,S)}) = \sum_{i=1}^{n}(\sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,i,j,k)} \cdot e_{(i,j)} \cdot e_{(i,k)})))+$$

$$\sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,j,k,time)} \cdot Time_j \cdot Time_k)) + l_t \tag{3.18}$$

Equation (3.17) illustrates the dependency of terms $\alpha_{(t,i)}$, λ_t and σ_t on the target configuration. We model each target configuration t through coefficients that capture the varying effects of hardware utilization at different degrees of concurrency, different mappings of threads to cores and different frequency levels. The term $\alpha_{(t,i)}$ scales the observed $Time_i$ on the sample configurations up or down based on the observed values of the event rates in that configuration. The constant term σ_t is an event rate-independent term. It includes the overhead time for parallelization or synchronization. The term λ_t combines the products of each event across configurations and of $Time_{j/k}$ to model interaction effects. Finally, μ is the target configuration-specific coefficient for each event pair and l is the event rate-independent term in the model.

We use multivariate linear regression (MLR) to obtain the model coefficients (α, μ and constant terms) from a set of training benchmarks. We select the training benchmarks empirically to vary properties such as scalability and memory boundedness. The training benchmarks we used are MM5 and UA from the NAS parallel benchmarks [139] OpenMP version with 119 phases in total. The observed time $Time_i$, the product of the observed time $Time_i$ and each event rate, and the interaction terms on the sample configurations are independent variables for the regression while $Time_t$ on each target configuration is the dependent variable. We derive sets of coefficients and model each target configuration separately.

We use three sample configurations: one uses the maximum concurrency and frequency,

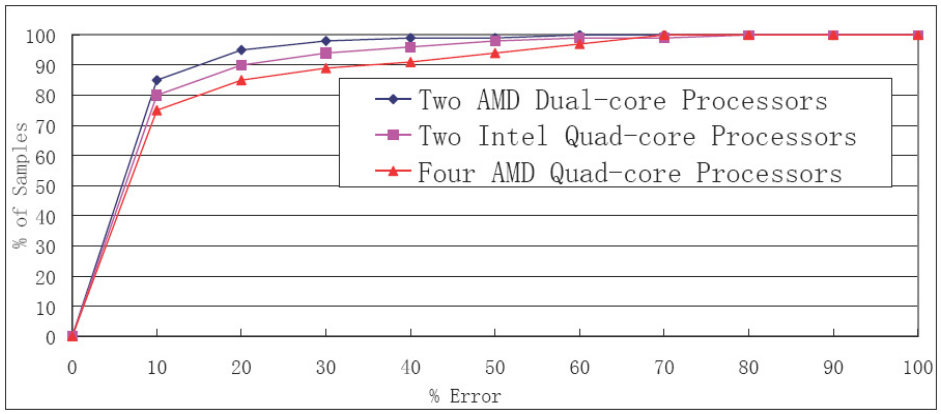


Figure 3.6: Cumulative distribution of execution time prediction accuracy

while the other two use configurations with half the concurrency—with different mappings of threads to cores—and the second highest frequency. Thus, we gain insight into utilization of shared caches and memory bandwidth while limiting the number of samples.

We verify the accuracy of our models on systems with three different node architectures. One has four AMD Opteron 8350 quad-core processors. The second has two AMD Opteron 265 dual-core processors. The third has two Intel Xeon E5462 quad-core processors. We present experiments with seven OpenMP benchmarks from the NAS Parallel Benchmarks suite (v3.1) with CLASS B input. We collect event rates from three sample configurations and make time predictions for OpenMP phase samples in the benchmarks. We then compare the measured time for the OpenMP phases to our predictions. Figure 3.6 shows the cumulative distribution of our prediction accuracy, i.e., the total percentage of OpenMP phases with error under the threshold indicated on the x-axis. The results demonstrate the high accuracy of the model in all cases: more than 75% of the samples have less than 10% error.

3.5 System Power Estimation Method

Our models need to estimate power to determine whether a configuration meets our energy constraints (Equation 3.15). The models also need power measurements to choose the configuration with the minimum energy consumption (Equation 3.16). We use the term power to refer to average system power.

Previous work [86] found that the average system power is approximately linear in processor frequency and processor utilization for any fixed utilization and frequency respectively. HPC applications running under different frequency settings keep CPU utilization high and therefore tend a near linear relationship of power to frequency. Therefore, we could estimate power based on frequency. Unfortunately, this method only works for adaptation methods that use DCT first, where all candidates have the same concurrency configuration. For DVFS first and the combined strategies, the candidates may have different concurrency configurations, which means that their CPU utilization may vary significantly. Therefore, we need a new method to estimate power for these cases.

Previous work [123] reveals strong correlation between power consumption and IPC. However, this correlation exists assuming that the processor frequency is fixed. When the frequency varies, the impact of processor stalls on IPC varies as well. On lower frequencies, processor stalls are smaller components of CPI than on higher frequencies, which in turn tends to increase IPC on lower frequencies. Therefore, IPC cannot reflect power consumption when frequency changes. We propose to use instructions per second (IPS) to estimate power. In particular, we use a linear model (Equation 3.19) to capture the relationship between IPS and power. The model parameters (k_1, k_0) are obtained by offline training. Given the IPS of any configuration, our model estimates the power at runtime as:

$$P = k_1 \times IPS + k_0 \tag{3.19}$$

To calculate IPS, we need the instruction count and execution time. We predict execution time using the method presented in Section 3.4. The instruction count is independent of processor frequency, but dependent on the number of threads executing in OpenMP phases, due to the varying parallelization and synchronization overhead. In principle, using more threads tends to increase the instruction count due to additional overhead. To estimate the instruction count, we record instruction counts from sample configurations and use linear interpolation to make estimations for the other untested configurations. Note that interpolation is different from the linear regression used in Section 3.4. Interpolation constructs new data points *within* a range of known data points. Therefore, the predicted value is bound by known values and guarantees prediction accuracy. We choose interpolation instead of linear regression since the instruction counts should be within the range of values collected from samples with minimum concurrency and samples with maximum concurrency. We cannot use interpolation for time prediction, because we cannot determine an upper bound of execution time in advance. Furthermore, we choose *linear* interpolation based on the intuition that the instruction count variance across different DCT configurations mainly comes from data accesses and thread synchronization and that these are approximately *linear* with the number of threads.

We train our model with three benchmarks (bt.B, cg.C and ft.B) from the OpenMP version of the NAS parallel benchmark suite with a total of 239 configurations and predict the power consumption of the other four NAS benchmarks (260 configurations) on a machine with four AMD Opteron 8350 quad-core processors. We also measure power with a Watts Up Pro ES power meter and compare with our prediction. The cumulative prediction accuracy is shown in Figure 3.7. We notice that prediction with estimated instruction counts is very close to prediction with measured instruction counts. This demonstrates the effectiveness of our estimate of instruction count. In addition, about 80% of the predictions has error less than 18% and more than 90% has error less than 25%, which proves that our prediction

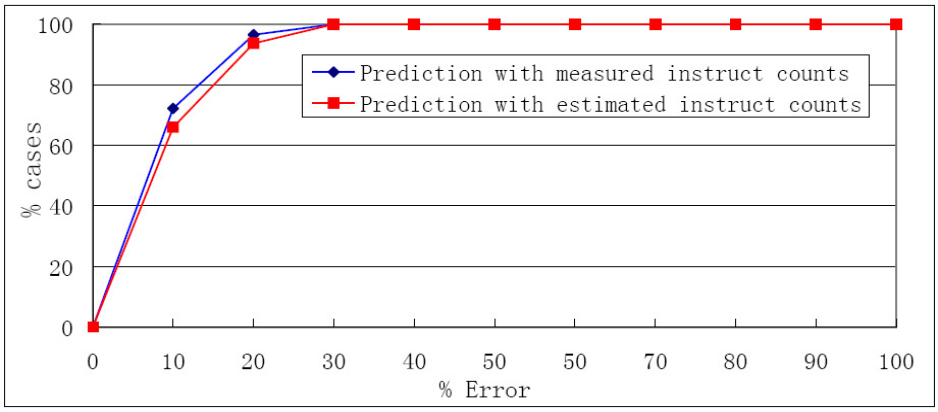


Figure 3.7: Cumulative distribution of power prediction accuracy

scheme is usually accurate.

3.6 Implementation

We have implemented our power-aware MPI/OpenMP model in a runtime system that performs online adaptation of DVFS and DCT. To use the runtime system, applications are instrumented with function calls around OpenMP phases and selected MPI operations (collectives and MPI_Waitall). This instrumentation is mechanical and could be automated with a source code instrumentation tool, like OPARI [134], in combination with a PMPI wrapper library.

We cannot apply DCT to those OpenMP phases in which the code in each thread depends on the thread identifier, since this would violate correct execution. In addition, we cannot make accurate time and power prediction for very short OpenMP phases due to the overhead of performing adaptation as well as accuracy limitations in performance counter measurements. We empirically identify a threshold of one million cycles as the minimum DCT granularity for an OpenMP phase. For each phase below this threshold, we simply use the active configuration of the preceding phase.

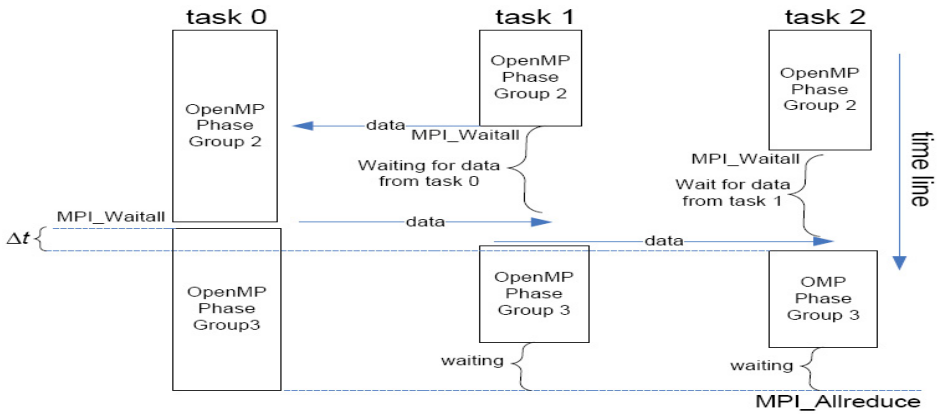


Figure 3.8: Impact of communication on slack with IRS

Our model computes slack time from Equation 3.6. The communication time used to compute slack ($t_i^{comm\_send}$) is estimated a priori, using MPPtest [77]. In practice, however, several factors can cause inaccuracies in the computed slack estimates. First, if the workload of OpenMP phases varies between outer iterations of the computation, the sampled iterations may have a different workload than non-sampled iterations. Second, our execution time predictions for OpenMP phases have error, as shown in Section 3.4, and therefore affect the computed slack. Third, communication phases preceding computation phases that incur slack, may reduce slack time compared to the time computed by our model. Figure 3.8 illustrates this possibility for a scenario observed in the IRS benchmark from the ASC Sequoia Benchmark Suite. More details about the IRS benchmark are given in Section 3.7. In this case, three tasks execute OpenMP phase group 2, non-blocking MPI communication, MPI_Waitall, OMP phase group 3, and MPI_Allreduce. Due to load imbalance, the tasks arrive at MPI_Allreduce at different times. Task 0, the most heavily loaded, arrives late. Task 2 arrives earlier and can disperse the slack that our model computes. However, point-to-point communication between group 2 and group 3, reduces the slack by Δt. In particular, task 2 starts executing OpenMP phase group 3 later than task 0, which reduces the available slack by Δt.

To solve the first two problems, we introduce an error tolerance, ϵ, that adjusts our computed slack to compensate for those inaccuracies, thus preventing reductions of the frequency beyond the actual slack, which is the maximum time we can disperse to the OpenMP phases by DVFS without incurring performance loss. We modify our slack model in Equation 3.6 to:

$$\Delta t_i^{slack} = (t_c - t_i - t_i^{comm\_send} - t_{dvfs})/(1+\epsilon) \tag{3.20}$$

The selection of an appropriate ϵ depends on prediction accuracy and other factors as discussed above. According to our results (shown in Figure 3.6) and practical experiences, a value between 0.1 and 0.2 effectively compensates for errors while allowing energy-saving in most cases. We use $\epsilon \leq 0.2$ for our evaluation, which results in negligible performance loss. This value corresponds to prediction errors of 20% or less, which captures most of our results.

To solve the third problem, we record the wait time in MPI operations (particularly MPI_Allreduce in Figure 3.8) and use it as an upper-bound of the slack that we can disperse. The rationale is that any gap (Δt) will be reflected as a shortened wait time and our slack should never be longer than the wait time. If the computed slack is longer than the wait time, we simply disperse the wait time minus DVFS overhead. This heuristic enhances the accuracy of our prediction and decreases the effective ϵ value from 1.6 to 0.2 for IRS.

Simply using the wait time to estimate the slack is insufficient. The wait time, including communication time, has higher variance across iterations due to minor network perturbations. Thus, recorded wait time can lead to too high an estimate of slack. The slack computed from Equation (3.20), on the other hand, reflects the slack that is actually available in the tasks and therefore is a more reliable value that is often less than the actual wait time.

Figure 3.9 displays how our runtime system works. We collect hardware events from performance counters to learn the application's execution properties. We use four sample configurations: one uses the maximum concurrency and frequency and one uses the minimum concurrency and frequency to bound instruction counts for linear interpolation (Section 3.5), while the other two use configurations with half the concurrency–with different mappings of threads to cores–and the second highest frequency. Using these samples, we gain insight into utilization of shared caches and memory bandwidth while keeping the number of samples low. Based on sample events, we are then able to predict execution time of other untested configurations with the coefficients obtained by offline training and our time prediction model (Section 3.4). After collecting hardware events, tasks coordinate with each other to determine critical tasks and available slack time. Based on slack time and the adaption strategy (Section 3.3), we choose the configuration candidate that satisfies the time constraint. Then we choose the best configuration based on the power model (Section 3.5). The best configuration should minimize energy consumption while satisfying the energy constraint (Section 3.3). We enforce the configuration decisions with the Linux processor affinity system call, $sched\_setaffinity()$, threading library-specific calls for changing concurrency levels ($omp\_set\_num\_threads()$ in the case of OpenMP), and a set of $cpufreq$ pseudofiles in the $/sys$ directory for changing processor frequencies. The rest of the execution is performed under the predicted optimal configurations.

3.7 Performance Evaluation

We evaluate our model with the Multi-Zone versions of the NPB benchmarks (NPB-MZ) [139] and two realistic applications (AMG and IRS). The NPB-MZ [102] suite has three benchmarks (LU-MZ, SP-MZ and BT-MZ). Each has the program flow shown in Figure 3.10. The benchmark loop has one procedure to exchange boundary values using point-to-point MPI

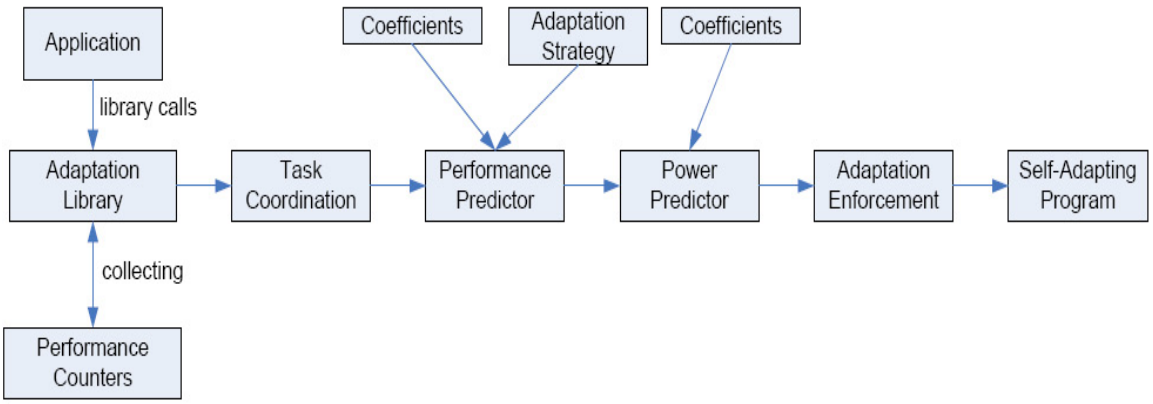


Figure 3.9: Online adaptation of DVFS and DCT for hybrid MPI/OpenMP applications

communication. Therefore, the entire benchmark loop has only one OpenMP phase group. A bin-packing algorithm balances the workload of the OpenMP phases between all tasks. Under this algorithm, LU-MZ and SP-MZ allocate the same number of zones for each task and each zone has the same size. For BT-MZ, zones have different sizes and each task owns a different number of zones, however each task has almost the same total zone size.

IRS uses a preconditioned conjugate gradient method for inverting a matrix equation. Figure 3.11 shows its simplified computational kernel. We group the OpenMP phases into four groups. Some OpenMP phase groups include serial code. We treat serial code as a special OpenMP phase with the number of threads fixed to 1. Although DCT is not applicable to serial code, it could be imbalanced between MPI tasks and hence provide opportunities for saving energy through DVFS. We use input parameters NDOMS=8 and NZONES_PER_DOM_SIDE=90. The IRS benchmark has load imbalance between the OpenMP phase groups of different tasks.

AMG [83] is a parallel algebraic multigrid solver for linear systems on unstructured grids. Its driver builds linear systems for various 3-dimensional problems; we choose a Laplace type problem (problem parameter set to 2). The driver generates a problem that is well balanced between tasks.

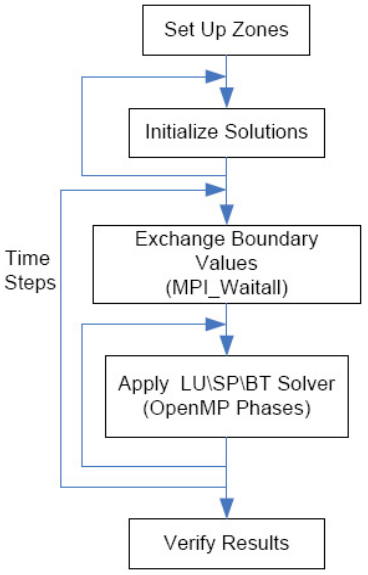


Figure 3.10: NPB-MZ flow graph

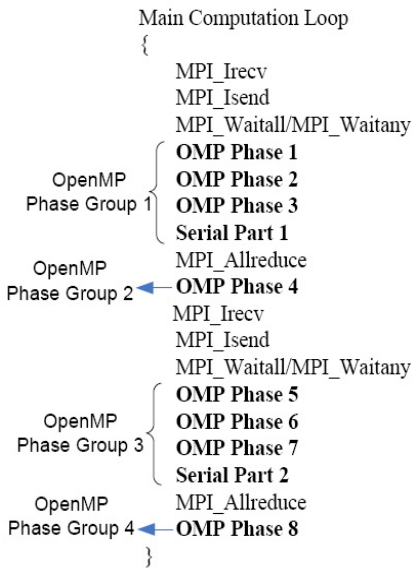


Figure 3.11: Simplified IRS flow graph

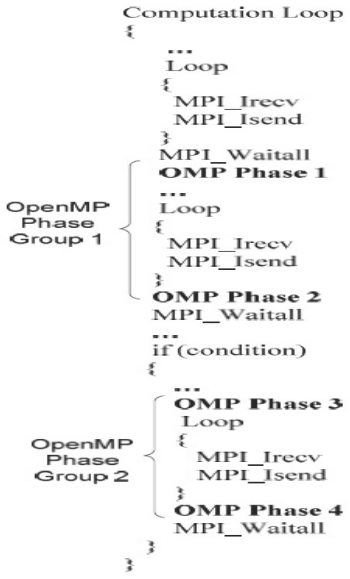


Figure 3.12: Simplified AMG flow graph

For our experiments, we introduce artificial load imbalance into BT-MZ and AMG to investigate how energy savings vary with explicit load imbalance under different strategies. However our system does not necessarily need imbalanced load to save energy. BT-MZ and AMG are chosen, because their problem partitions across tasks can be easily changed without violating the correctness of the applications. In particular, we slightly modified BT-MZ so that each task owns the same number of zones, but each task has a different total zone size. We also generated a new problem with imbalance load for AMG. The load distribution ratio between pairs of MPI tasks in this new version is 0.45:0.55.

We categorize hybrid MPI/OpenMP applications based on the workload characteristics of their OpenMP phases: (1) imbalanced and constant workload per iteration (e.g., modified BT-MZ) or nearly constant workload per iteration (e.g., IRS); (2) imbalanced and non-constant workload per iteration (e.g., modified AMG); (3) balanced workload (e.g., SP-MZ, LU-MZ, BT-MZ and AMG).

We first run all benchmarks on two homogeneous nodes, each with four AMD Opteron

8350 quad-core processors (a total of 16 cores per node). The baseline is the execution under the configuration using 4 processors and 4 cores per processor, all running at the highest processor frequency. DVFS on the AMD Opteron 8350 has five frequency settings and we apply DVFS to the whole processor (all cores on a single socket). We display the results with three adaptation strategies (Section 3.3). The results with pure DCT are displayed as well, so that we can know whether DCT or DVFS lead to the energy savings.

Figure 3.13 shows the results for the NPB benchmarks. The pure DCT scheme selects the same concurrency configuration as the performance baseline for BT-MZ, which leads to no performance or energy gains. Due to good scalability of the OpenMP phases, DCT maintains maximum concurrency and cannot save energy. However with DCT first, we achieve energy savings (10.21%) with almost no performance loss. With DVFS first, we choose the same DCT/DVFS configurations as DCT first and thus the energy consumption is almost the same as DCT first. With the combined strategy, we achieve the largest energy saving (10.85%), because the combined strategy is able to choose a configuration that cannot be selected by the other strategies.

The OpenMP phases in SP-MZ do not scale well, so we can save energy (5.72%) by applying DCT alone. Due to the balanced load in SP-MZ, our DVFS algorithm cannot save energy, as shown by pure DCT and the three strategies having the same energy consumption. The LU-MZ benchmark has scalable OpenMP phases and balanced load, so our runtime system does not have any opportunity to save energy. However, this test case shows that our system has negligible overhead (0.736%).

The AMG problem that we study has non-constant workload per iteration, which makes our predicted configurations based on sampled iterations incorrect in later iterations. After profiling its OpenMP phases, we find that AMG has a periodic workload. OpenMP phase group 1 has a period of 14 iterations and OpenMP phase group 2 has a period of 7 iterations. Threfefore, we can still apply our control schemes, but with application-specific sampling.

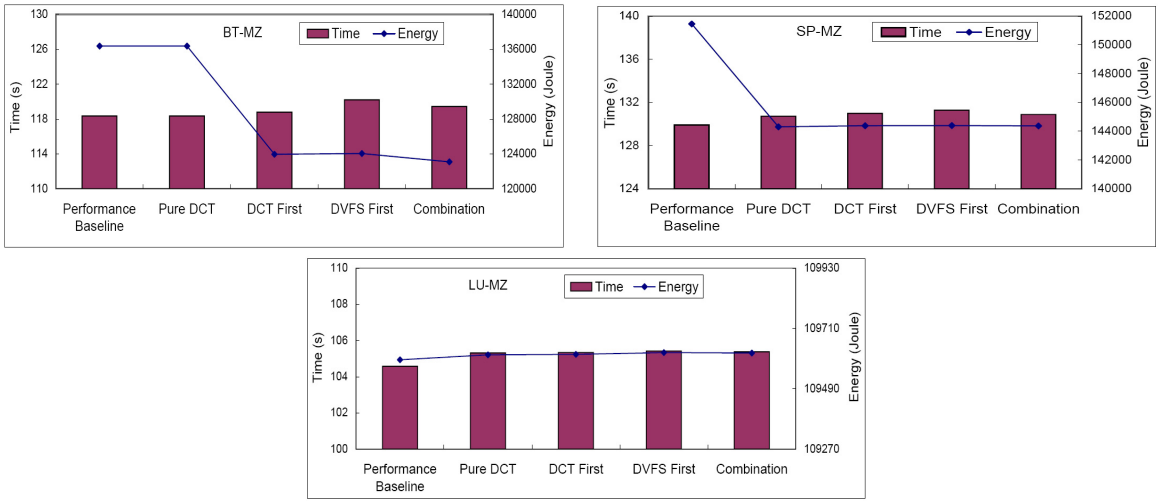


Figure 3.13: Execution time and energy consumption of NPB-MZ under adaptive DCT/DVFS control

Since the workload within a period varies from one iteration to another, we select configurations for every iteration within a period. We use more sample iterations during at least one period and change configurations for each iteration within a period.

The results show that pure DCT achieves 8.38% energy saving and 7.39% performance gain. The best energy saving (13.80%) is achieved by applying either DCT first or the combined strategy. DVFS first, however, achieves less energy saving, due to the inability to fully leverage slack time (discussed in Section 3.3). In IRS, we observe performance improvement of 7.5% and energy savings of 12.25% by applying only DCT. By applying DVFS with three strategies, we can further reduce energy however with a slight performance loss, compared to the performance of pure DCT because the workload in OpenMP phases varies slightly and irregularly. The selection of our DVFS scheme based on sample iterations may hurt performance in the rest of the run. The best energy savings (13.31%) are achieved with DCT first and the combined strategy, with overhead of only 1.89% and 2.07% respectively, compared to pure DCT.

To summarize, our hybrid MPI/OpenMP applications present different energy-saving op-

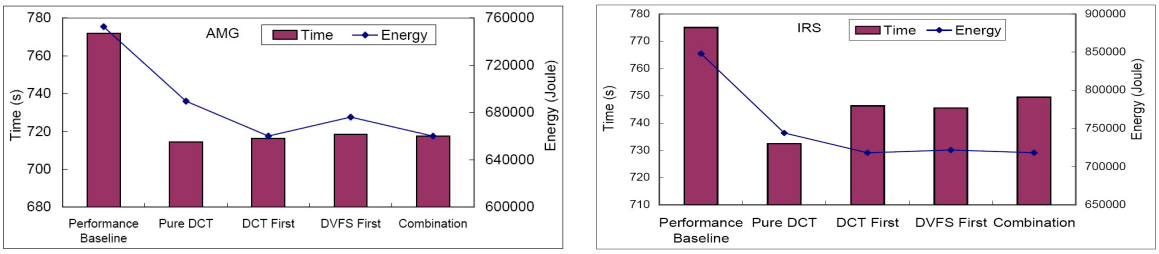


Figure 3.14: Execution time and energy consumption of AMG and IRS under adaptive DCT/DVFS control

portunities and the energy-saving potential depends on workload characteristics. Our model can detect and leverage this potential. In particular, for balanced workloads, if OpenMP phases are non-scalable, we can save energy with DCT; if OpenMP phases are scalable, our algorithm does not save energy, but also does not hurt performance. For imbalanced and constant (or close to constant) per iteration workloads, our algorithm is effective, saving energy while maintaining performance. For imbalanced and non-constant per iteration workload, if the workload is periodic, we can still apply our algorithm after detecting the periodicity of the workload; if the workload is totally irregular, our algorithm can lead to increased runtimes and energy consumption. We could detect the period of a workload by testing applications with small problem sets. Alternatively, many scientific applications use a recursive computation kernel, thus creating a periodic workload that we could track based on the stack trace depth. We also notice that the combination strategy consistently achieves the best energy saving, while the other two strategies cannot guarantee the selection of the most energy efficient configurations.

We extend our analysis to systems of larger order to investigate how our model reacts as the number of nodes changes. The following experiments consider scalability of energy saving techniques. We present results from experiments on the System G supercomputer at Virginia Tech. System G is a research platform for Green HPC, composed of 320 nodes powered by Mac Pro computers, each with 2 quad-core Xeon processors, and thousands of

power and thermal sensors. Each processor has two frequency settings for DVFS. The nodes are connected by Infiniband (40Gb/s). We vary the number of nodes and study how our power-aware model performs under strong and weak scaling. We use the execution under the configuration using 2 processors and 4 cores per processor running at the highest processor frequency, which we refer to as (2,4), as the baseline by which we normalize reported times and energy.

Figure 3.15 displays the results of AMG and IRS under strong scaling (i.e., maintaining the same total problem size across all scales). Actual execution time is shown above normalized execution time bars, to illustrate how the benchmark scales with the number of nodes. On our cluster, the OpenMP phases in AMG scale well, therefore DCT does not find energy-saving opportunities in almost all cases although DCT leads to concurrency throttling on some nodes with 64 nodes or more. However due to the small length of OpenMP phases at this scale, DCT does not lead to significant energy savings. When the number of nodes reaches 128, the per node workload in OpenMP phases is further reduced to a point where some phases become shorter than our DCT minimum phase granularity threshold and DCT simply ignores them.

On the other hand, our three strategies using DVFS save a significant amount of energy in most cases. However, as the number of nodes increases, the ratio of energy-saving decreases from 3.72% (4 nodes) to 0.121% (64 nodes) because the load difference between tasks becomes smaller as the number of nodes increases. With 128 nodes, load imbalance is actually less than DVFS overhead, so DVFS becomes ineffective. We also notice the slightly increased overhead when the number of nodes is increased due to the task coordination cost, which becomes larger as the node count increases. In the AMG benchmark, which has a workload period of 14, 14 coordinations (DCT first) or 28 coordinations (DVFS first and combination) are needed to select the appropriate configurations. However, the overhead is less than 4% in total and less than 2% in most cases. In IRS, our strategies with DCT lead to measurable

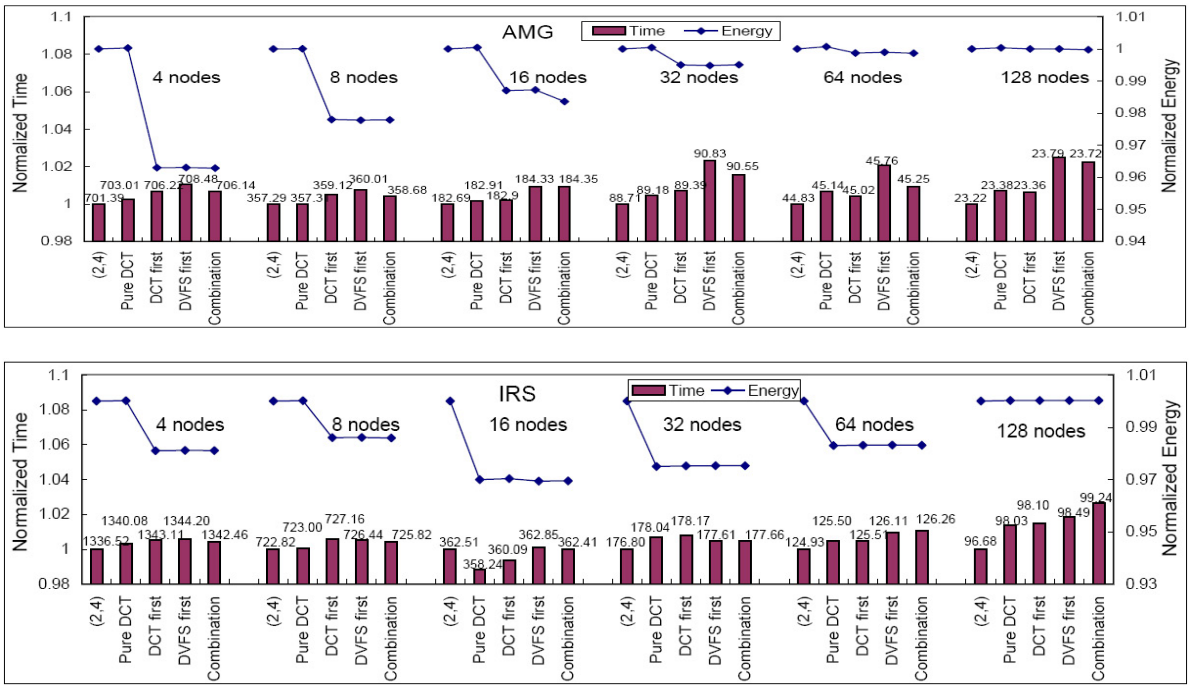


Figure 3.15: Strong scaling results on System G

energy savings when the number of nodes is more than 8 (up to 3.06%). We even observe performance gains with DCT when the number of nodes reaches 16. However, DCT does not lead to energy savings in the case of 128 nodes for similar reasons to AMG. DVFS leads to energy savings with less than 16 nodes but does not provide benefits as the number of nodes becomes large and the imbalance becomes small.

Figures 3.16 displays weak scaling results. We adjust the input parameters as we vary the number of nodes so that the problem size per node remains constant (or close to it). For IRS, the energy savings ratio grows slightly as we increase the number of nodes (from 1.9% to 2.5%). Slightly increased imbalance, as we increase the problem size, allows additional energy savings. For AMG, we observe that the ratio of energy savings stays almost constant (2.17%∼2.22%), which is consistent with AMG having good weak scaling. Since the workload per node is stable, energy saving opportunities are also stable as we vary the number of nodes.

In general, energy-saving opportunities vary with workload characteristics. They become

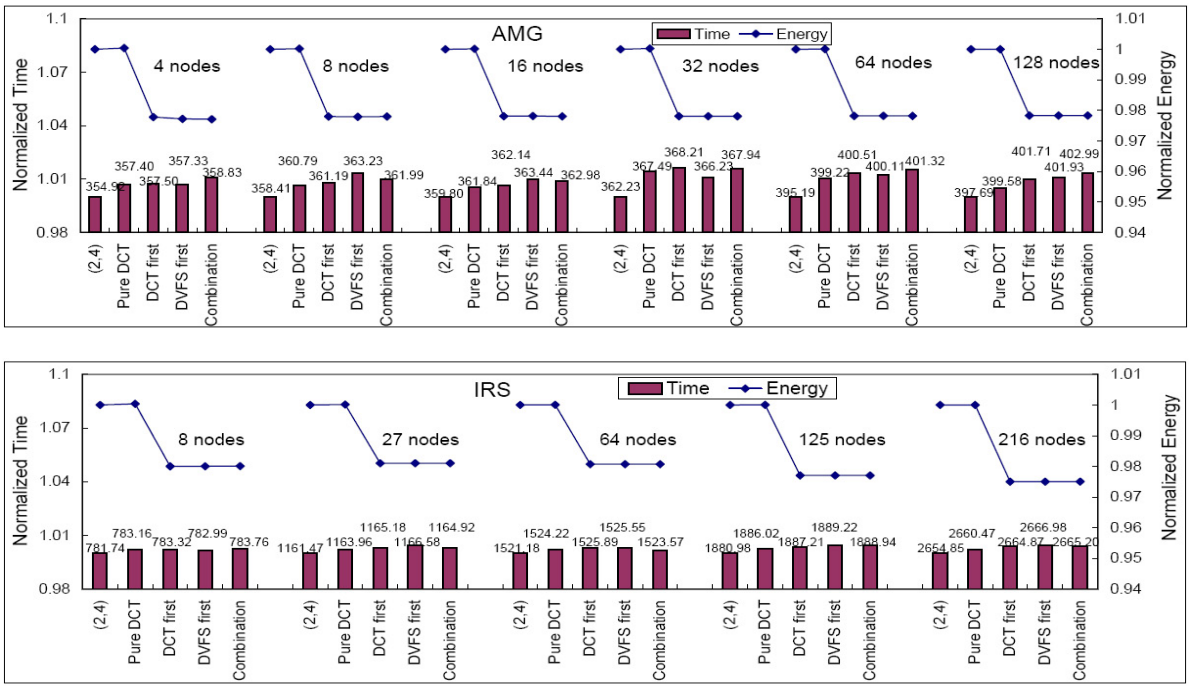


Figure 3.16: Weak scaling results on System G

smaller as the number of nodes increases under a fixed total problem size because the subdomain allocated to a single node becomes so small that the energy-saving potential that DVFS or DCT can leverage falls below the threshold that we can exploit. An interesting observation is that, when the number of nodes is below the threshold, some benchmarks (e.g., IRS with less than 32 nodes) present good scalability of energy-saving opportunities for DCT because of the changes in their workload characteristics (e.g., scalability and working data sets) as the allocated sub-domain changes. With weak scaling, energy-saving opportunities are usually stable or increasing and actual energy savings from our model tends to be higher than with strong scaling. Most importantly, under any case our model can leverage any energy saving opportunity without significant performance loss as the number of nodes changes.

3.8 Chapter Summary

In this chapter, we presented models and algorithms for energy-efficient execution of hybrid MPI/OpenMP applications. We characterized energy-saving opportunities in these applications, based on the interaction between communication and computation. We used this characterization, to propose algorithms using two energy-saving tools, DCT and DVFS, to leverage energy-saving opportunities without performance loss. We study three strategies to apply DCT and DVFS and investigate their impact on energy savings. We propose a power estimation model that can estimate power for variant concurrency and frequency configurations.

Our work improves existing DCT techniques by characterizing the potential performance loss due to concurrency adjustment. We use this insight to provide performance guarantees in our new "one phase approach", which balances between DCT performance penalties and energy savings. We also present a more accurate model for measuring slack time for DVFS control and solve the problem of frequency selection using dynamic programming. We apply our model and algorithm to realistic MPI/OpenMP benchmarks at larger scales than any previously published study. Overall, our new algorithms yield substantial energy savings (8.74% on average and up to 13.8%) with either negligible performance loss or performance gain (up to 7.5%). Further, our results are the first to characterize how energy-saving opportunities vary under strong and weak scaling, on systems with large node and core counts.

This page intentionally left blank.

Chapter 4

Power-Aware MPI Task Aggregation Prediction

Modern high-end computing systems have many nodes with several processors per node and multiple cores per processor. The distribution of tasks across the cores of multiple nodes impacts both execution time and energy. Current job management systems, which typically rely on a count of available cores for assigning jobs to cores, simply treat parallel job submissions as a 2D chart with time along one axis and number of cores along the other [58, 165]. They regard each job as a rectangle with width equal to the number of cores requested by the job and height equal to the estimated job execution time. Most scheduling strategies are based on this model, which has been extensively studied [16, 115, 178]. Some job scheduling systems also consider communication locality factors such as the network topology [163]. Unfortunately, job schedulers ignore the power-performance implications of the layouts of cores available in compute nodes to execute tasks from parallel jobs.

In the previous chapter, we assume one MPI task per node without consideration of task distribution. In this chapter, we remove this assumption and consider the effects of task distribution. In particular, we propose power-aware task aggregation.

Task aggregation refers to aggregating multiple tasks within a node with shared memory. A fixed number of tasks can be distributed across a variable number of nodes, using different degrees of task aggregation per node. Aggregated tasks share system resources, such as the memory hierarchy and network interface, which has an impact on performance. This impact may be destructive, because of contention for resources. However, it may also be constructive. For example, an application can benefit from the low latency and high bandwidth of intra-node communication through shared memory. Although earlier work has studied the performance implications of communication through shared-memory in MPI programs [36, 37, 118], the problem of selecting the best distribution and aggregation of a fixed number of tasks has been left largely to ad hoc solutions.

Task aggregation significantly impacts energy consumption. A job uses fewer nodes with a higher degree of task aggregation. Unused nodes can be set to a deep low-power state while idling. At the same time, aggregating more tasks per node implies that more cores will be active running tasks on the node, while memory occupancy and link traffic will also increase. Therefore, aggregation tends to increase the power consumption of active nodes. In summary, task aggregation has complex implications on both performance and energy. Job schedulers should consider these implications in order to optimize energy-related metrics while meeting performance constraints.

We target the problem of how to distribute MPI tasks between and within nodes in order to minimize execution time, or minimize energy, under a given performance constraint. The solution must make two decisions: how many tasks to aggregate per node; and how to assign the tasks scheduled on the same node to cores, which determines how these tasks will share hardware components such as caches, network resources, and memory bandwidth. In all cases, we select a task aggregation pattern based on performance predictions.

We assume the following:

1. The number of MPI tasks is given and fixed throughout the execution of the application;
2. The number of nodes used to execute the application and the number of tasks per node is decided at job submission time and this decision depends on a prediction of the impact of different aggregation patterns on performance and energy;
3. Any aggregation must assign the same number of tasks to each node;
4. Jobs are SPMD (Single Program Multiple Data) programs;
5. MPI communication patterns—including message size and communication target—can vary across tasks;
6. Aggregation patterns must not result in total DRAM requirements that exceed a node's physical memory.

Allowing aggregation patterns that place more tasks on some nodes than others may be more efficient in imbalanced applications, however, the resulting load imbalance would hurt performance and waste energy in well balanced SPMD applications. In these cases, the system could leverage slack to save energy. We studied energy-saving opportunities due to slack in the previous chapter.

A series of tests with the NAS 3.2 MPI parallel benchmarks (problem size D) on 16 nodes of a cluster under various aggregation patterns demonstrates the effect of aggregation on execution time and energy. Each node of this cluster has two Xeon E5462 quad-core processors, each of which has two dies shared by two cores and 8 GB of physical memory. Figure 4.1 shows the results for the eight possible aggregation patterns on this platform. Energy and energy-delay product data are normalized to the results with aggregation pattern 1. Figure 4.2 depicts the aggregation patterns with the cores assigned to tasks indicated with stripes. Pattern 8 is not available for some benchmarks as each task requires more than an

eighth of the memory on a node. Our results for FT use more grid points and iterations than in the standard problem size D to achieve longer execution times. As seen in the execution time and energy consumption of the benchmarks across all feasible aggregation patterns, no single pattern always provides the best results. For example, using one core of each die (pattern 5) for CG uses the least energy but results in sub-optimal execution time, while using one core of each processor (pattern 2) for FT provides the lowest execution time but does not minimize energy consumption.

We decompose the aggregation problem into three subproblems:

1. Predicting the impact of task count per node on computation;

2. Predicting the communication cost of all aggregation patterns;

3. Combining the computation and communication predictions.

We study the impact of aggregation on computation and communication separately, since the same aggregation pattern can impact computation and communication differently.

In this chapter, we present a prediction-based framework to solve the aggregation problem. The framework exploits the iterative structure of parallel phases that dominate execution time in scientific applications. We collect samples of hardware event counters and communication pattern information at runtime. From this data, we predict the performance under all feasible aggregation patterns and then rank the patterns to identify a preferred aggregation pattern.

The rest of this chapter is organized as follows. Section 4.1 presents our method for predicting the performance of computation phases after task aggregation. Section 4.2 presents our graph partitioning algorithm for task grouping. Section 4.3 presents our communication performance prediction method. Section 4.4 discusses our method for ranking aggregation patterns. Section 4.5 presents our experimental analysis. Section 4.6 concludes the chapter.

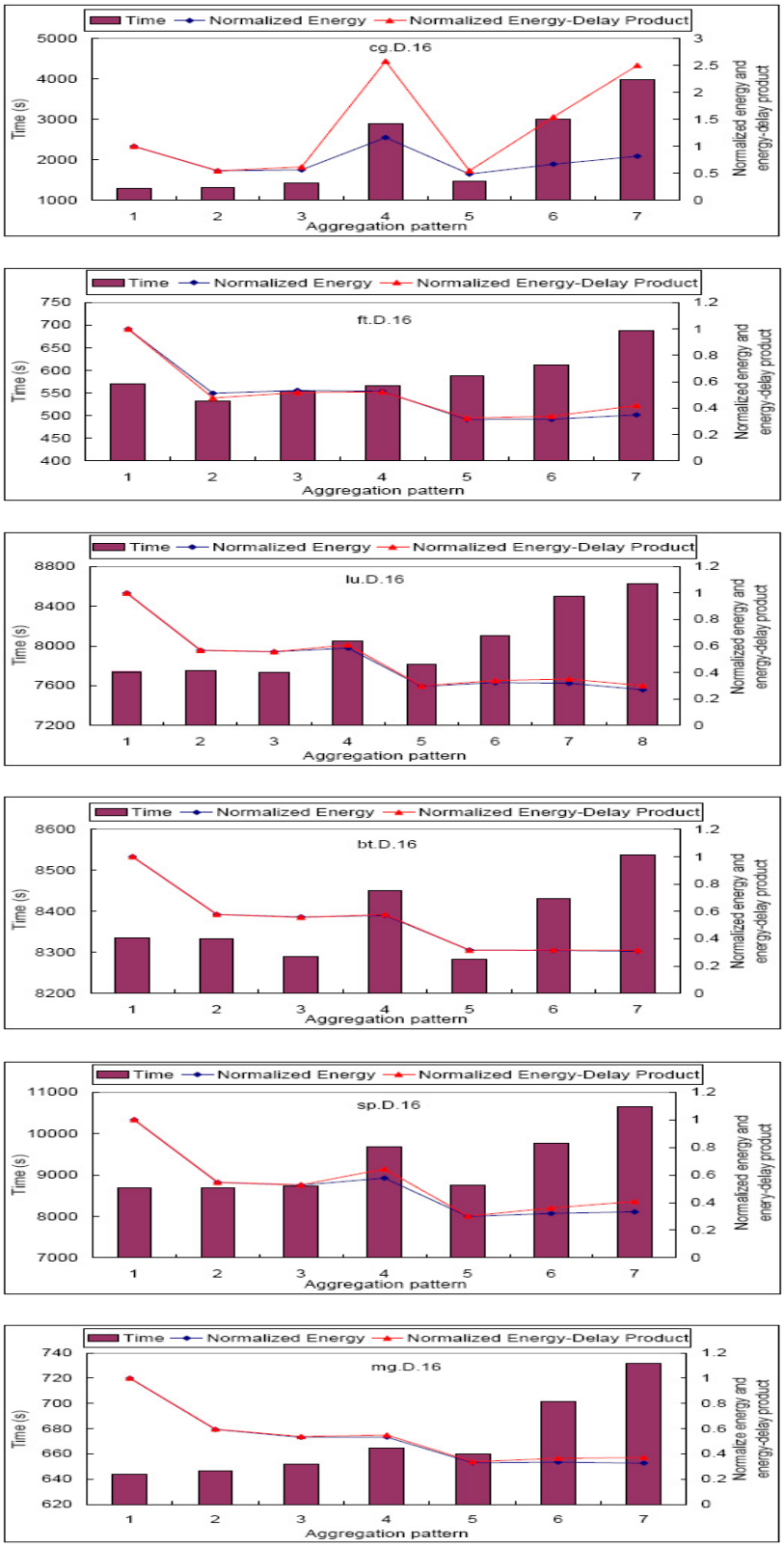


Figure 4.1: Impact of task aggregation on the NAS PB suite

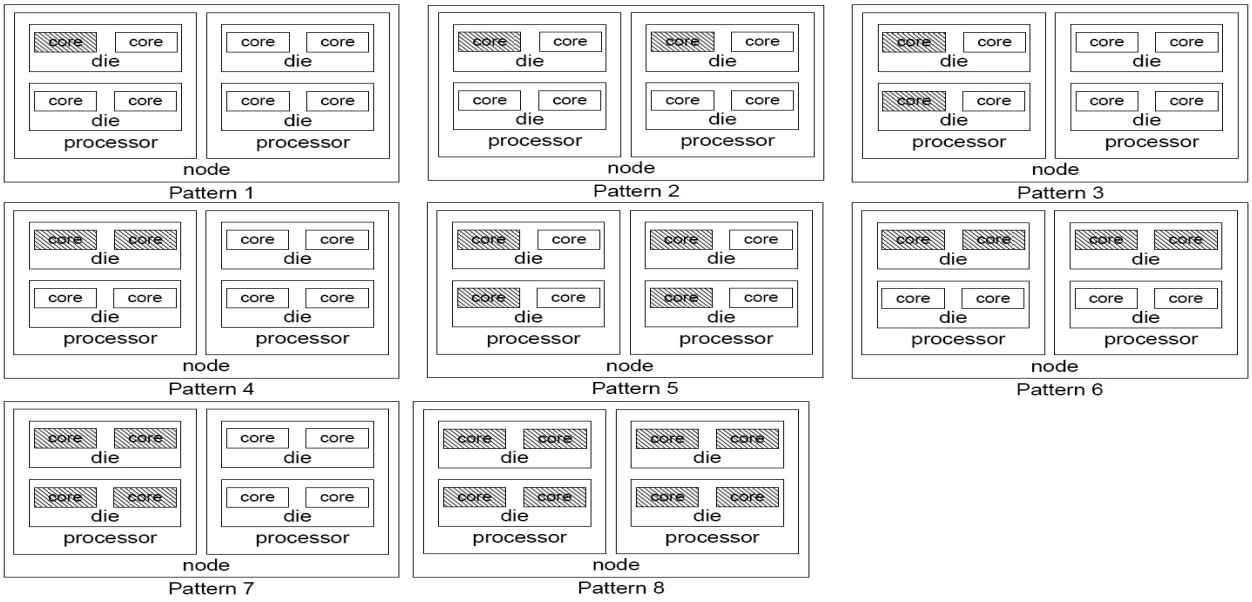


Figure 4.2: Aggregation patterns on our test platform

4.1 Predicting Computation Performance

We predict performance during computation phases by predicting IPC. We derive an empirical model based on previous work [47, 48, 49], which predicts IPC of computation phases in OpenMP applications. We use iteration samples collected at runtime on specific aggregation patterns to predict the IPC for each task on other untested aggregation patterns. The IPC for each aggregation pattern is the average value of the IPC of all tasks.

The execution samples provide statistical indicators about the execution properties of computation phases that impact IPC. Different aggregation patterns imply different patterns in resource sharing and contention, which in turn influence IPC. If we can accurately predict the impact of sharing and contention on IPC then we can also identify the aggregations that improve energy efficiency.

Based on this discussion, we build an empirical model derivation shown in equation (4.1):

$$IPC_t = \sum_{i=1}^{|S|} (IPC_i \cdot \alpha_{(t,i)}(e_{(1\cdots n,i)})) + \lambda_t(e_{(1\cdots n,S)}) + \sigma_t \tag{4.1}$$

The model predicts the IPC of a specific aggregation pattern t based on information collected in S samples. We collect n hardware event rates $e_{(1\cdots n,i)}$ and IPC_i in each sample i. The function $\alpha_{(t,i)}()$ scales the observed IPC_i in sample i up or down based on the observed values of event rates while λ_t is a function that accounts for the interaction between events and σ_t is a constant term for the aggregation pattern t. For a specific sample s, we define α_t as:

$$\alpha_t(e_{(1\cdots n,s)}) = \sum_{j=1}^{n}(x_{(t,j)} \cdot e_{(j,s)} + y_{(t,j)}) + z_t \tag{4.2}$$

where $e_{(j,s)}$ is a hardware event in sample s, and $x_{(t,j)}$, $y_{(t,j)}$ and z_t are coefficients.

λ_t is defined as:

$$\begin{aligned}\lambda_t(e_{(1\cdots n,S)}) = \sum_{i=1}^{n}(\sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,i,j,k)} \cdot e_{(i,j)} \cdot e_{(i,k)})))+ \\ \sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,j,k,IPC)} \cdot IPC_j \cdot IPC_k)) + l_t\end{aligned} \tag{4.3}$$

where $e_{(i,j)}$ is the i_{th} event of the j_{th} sample. $\mu_{(t,i,j,k)}$, $\mu_{(t,j,k,IPC)}$ and l_t are coefficients.

We approximate the coefficients in our model with multivariate linear regression. IPC, the product of IPC and each event rate, and the interaction terms in the sample aggregation patterns serve as independent variables, while the IPC on each target aggregation pattern serves as the dependent variable. We record IPC and a predefined collection of event rates while executing the computation phases of each training benchmark with all aggregation patterns. We use the hardware event rates that most strongly correlate with the target IPC in the sample aggregation patterns. We develop a model separately for each aggregation pattern and derive sets of coefficients independently. We use the twelve SPEC MPI 2007 benchmarks [166] under different problem sets as training benchmarks. These benchmarks demonstrate wide variation in execution properties such as scalability

	Measured IPC	Predicted IPC	Error rate
lu.D.16, pattern1	1.926	1.944	0.9510%
lu.D.16, pattern2	1.921	1.937	0.8234%
lu.D.16, pattern3	1.924	1.942	0.9109%
lu.D.16, pattern6	1.846	1.834	0.6689%
lu.D.16, pattern7	1.763	1.743	1.137%
lu.D.16, pattern8	1.699	1.669	1.741%
bt.D.16, pattern1	2.033	2.028	0.2475%
bt.D.16, pattern2	2.048	2.032	0.7474%
bt.D.16, pattern3	2.051	2.041	0.4738%
bt.D.16, pattern6	2.014	2.008	0.3041%
bt.D.16, pattern7	1.989	1.995	0.3033%
ft.D.16, pattern1	1.441	1.469	1.934%
ft.D.16, pattern2	1.568	1.601	2.109%
ft.D.16, pattern3	1.512	1.543	2.032%
ft.D.16, pattern6	1.451	1.479	1.948%
ft.D.16, pattern7	1.291	1.314	1.730%
sp.D.16, pattern1	1.952	1.976	1.259%
sp.D.16, pattern2	1.956	1.971	0.7839%
sp.D.16, pattern3	1.99	1.955	0.3166%
sp.D.16, pattern6	1.755	1.789	1.951%
sp.D.16, pattern7	1.610	1.628	1.110%

Table 4.1: IPC prediction of computation phases

and memory-boundedness.

We classify computation phases into four categories based on their observed IPC during the execution of the sample aggregation patterns and use separate models for different categories in order to improve prediction accuracy. Specifically, we classify phases into four categories with IPC $[0, 1), [1, 1.5), [1.5, 2.0)$ and $[2.0, +\infty)$. Thus our model is a piecewise linear regression that attempts to describe the relationship between dependent and independent variables more accurately by separately handling phases with low and high scalability characteristics.

We test our model by comparing our predicted IPC with the measured IPC of the computation phases of several NAS MPI parallel benchmarks. We present results from tests on the Virginia Tech System G supercomputer (see Section 4.5) in Figure 4.1. Patterns 4 and 5 from Figure 4.2 serve as our sample aggregation patterns. Our model is highly accurate,

as the results in Table 4.1 show, with worst-case absolute error of 2.109%. The average error in all predictions is 1.079% and the standard deviation is 0.7916.

4.2 Task Grouping Problem

An aggregation pattern determines how many tasks to place on each node and processor; we must also determine which tasks to collocate. If an aggregation groups k tasks per node and a program uses n tasks, there are $\binom{n}{k}$ ways to group the tasks to achieve the aggregation. For nodes with $p \geq k$ processors, we then can place the k tasks on one node in $\binom{p}{k}k! = \frac{p!}{(p-k)!}$ ways on the available cores. The grouping of tasks on nodes and their placement on processors has an impact on the performance of MPI point-to-point communication. Computation phases are only sensitive to how tasks are laid out in each node and not to which subset of tasks is aggregated in each node since we assume SPMD applications with balanced workloads between processors. The impact of task placement for MPI collective operations depends on the characteristics of the network; they are relatively insensitive to task placement with flat networks such as fat trees. Thus, we focus on point-to-point operations as we decide which specific MPI ranks to locate on the same node or processor.

We demonstrate how MPI point-to-point communication is sensitive to locations of communication source and target in Figure 4.3, which shows results of single ping-pong pattern tests with the Intel MPI benchmark [95] using OpenMPI-1.3.2 [140]. Each test has two tasks involved in an MPI point-to-point communication. For the inter-node results, we used two nodes connected with a 40Gb/s InfiniBand network. Each node has two Intel Xeon E5462 quad-core processors. We also test the three possible intra-node placements (same die; different dies, same processor; different processors — see Figure 4.2).

The results reveal that intra-node communication has low latency for small messages, while inter-node high bandwidth communication is more efficient for large messages. These

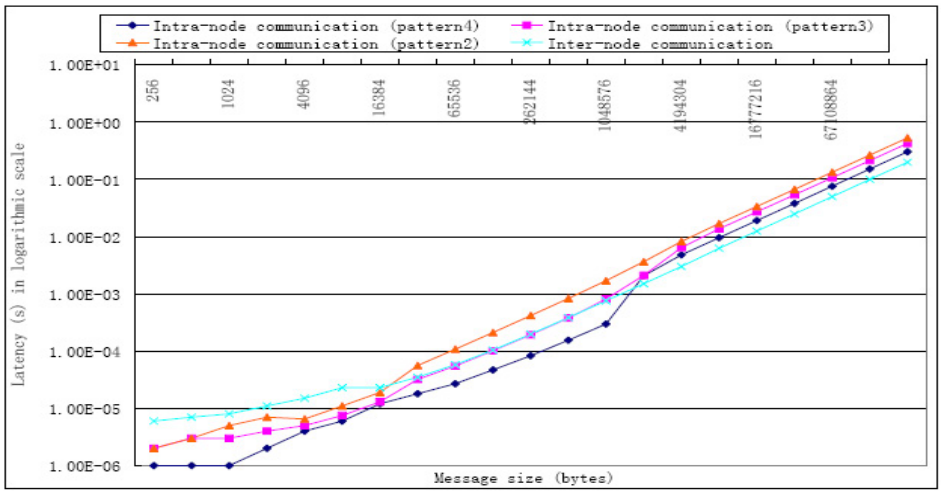


Figure 4.3: Intra-node vs. inter-node latency comparison

conclusions are consistent with previous results on the Myrinet2000 network [118]. In theory, intra-node communication can leverage the low latency and high throughput of the node memory system. Therefore, it should outperform inter-node communication. In practice, sharing the node's memory bandwidth between communicating tasks while they exchange large messages incurs sufficient overhead to make it less efficient than inter-node communication. We also find that the performance of intra-node communication is sensitive to how the tasks are laid out within a node: intra-node communication can benefit from cache sharing due to processor die sharing or whole processor sharing.

Based on these results, we prefer aggregations that colocate tasks based on whether their communication is in the latency or bandwidth regime. However, we cannot decide whether to colocate a given pair of tasks based only on individual point-to-point communications between them. Instead, we must consider all communication performed between those tasks and all communication between all tasks. Overall performance may be best even though some (or all) point-to-point communication between two specific tasks is not optimized.

Task grouping is an NP-complete problem [145]. We formalize the problem as a graph partitioning problem and use an efficient heuristic algorithm [106] to solve it. We briefly

review this algorithm in the following section.

4.2.1 Algorithm Review

The algorithm partitions a graph G of kn nodes with associated edge costs into k subpartitions, such that the total cost of the edge cut, the edges connecting subpartitions, is minimized. The algorithm starts with an arbitrary partitioning into k sets of size n and then tries to bring the partitioning as close as possible to being pairwise optimal by repeated application of a 2-way partitioning procedure.

The 2-way partitioning procedure starts with an arbitrary partitioning $\{A, B\}$ of a graph G and tries to decrease the initial external cost T (i.e., the total cost of the edge cut) by a series of interchanges of subsets of A and B. The algorithm stops when it cannot find further pair-wise improvements. To choose the subsets of A and B, the algorithm first selects two graph nodes a_1, b_1 such that the gain g_1 after interchanging a_1 with b_1 is maximal. The algorithm temporarily sets aside a_1 and b_1 and chooses the pair a_2, b_2 from $A - \{a_1\}$ and $B - \{b_1\}$ that maximizes the gain g_2. The algorithm continues until it has exhausted the graph nodes. Then, the algorithm chooses m to maximize the partial sum $\sum_{i=1}^{m} g_i$. The corresponding nodes $a_1, a_2, ..., a_m$ and $b_1, b_2, ..., b_m$ are exchanged.

This algorithm has a reasonable probability of finding the optimal partition. The number of subset exchanges before the algorithm converges to a final partition for a 2-way partitioning is between 2 and 4 for a graph with 360 edges [106]. In our experiments we execute programs with at most 128 tasks (as shown in Section 4.5). In this case, the subset exchanges for 2-way partitioning (2 cluster nodes) is at most 2 and the number of total subset exchanges for 16-way partitioning is at most 1200.

4.2.2 Applying the Algorithm

Task aggregation must group T tasks into n partitions, where n is the number of nodes of a specific aggregation pattern. We regard each MPI task as a graph node and communication between tasks as edges. Aggregating tasks into the same node is equivalent to placing graph nodes into the same partition. We now define an edge cost based on the communication between task pairs.

The original algorithm tries to minimize the total cost of the edge cut. In other words, it tries to place graph nodes with a small edge cost into different partitions. Thus, we must assign a small (large) cost value on an edge that favors inter-node (intra-node) communication. We observe two further edge cost requirements:

1. The difference between the small cost value (for inter-node communication) and the large cost value (for intra-node communication) should be large;

2. The edge values should form a range that reflects the relative benefit of intra-node communication.

A large difference between edge costs reduces the probability of the heuristic algorithm selecting a poor partitioning. The range of values reflects that colocation benefits some task pairs that communicate frequently more than others.

To assign edge costs, we measure the size of every message between each task pair during execution to obtain a communication table. We then estimate the communication time for each pair of communicating tasks i and j if we place them in the same partition (t_{ij}^{intra}) and if we place them in different partitions (t_{ij}^{inter}). We estimate these communication times experimentally by using data similar to that shown in Figure 4.3. Our intra-node communication time prediction is conservative, since we use the worst-case intra-node communication (i.e., two tasks with no processor or die sharing). Finally, we compare t_{ij}^{intra} and t_{ij}^{inter} to decide

whether the tasks i and j benefit from colocation. If $t_{ij}^{intra} > t_{ij}^{inter}$ then we set the edge cost to $c_{ij} = 1.0/(t_{ij}^{intra} - t_{ij}^{inter})$. Alternatively, if $t_{ij}^{intra} \leq t_{ij}^{inter}$, we set $c_{ij} = C + (t_{ij}^{inter} - t_{ij}^{intra})$. These edge costs provide a range of values that reflect the relative benefit of intra-node communication as needed.

C is a parameter that ensures the difference of edge costs for pairs of tasks that favor intra-node communication and pairs of tasks that favor inter-node communication is large. We define C as:

$$C = k^2 \Delta t \tag{4.4}$$

where k is the number of tasks per node (i.e., $k = T/n$). k^2 is the maximum number of edge cuts between the two partitions. Δt is defined as $\max\{1.0/(t_{ij}^{intra} - t_{ij}^{inter})\}$ between all task pairs (i, j) that benefit from inter-node communication.

Overall, our edge costs reflect whether the communication between a task pair is in the latency or bandwidth regime. We apply the graph partitioning algorithm based on these edge costs to group tasks into n nodes. We then use the same algorithm to determine the placement of tasks on processors within a node. Thus, this algorithm calculates a task placement for each aggregation pattern.

4.3 Predicting Communication Performance

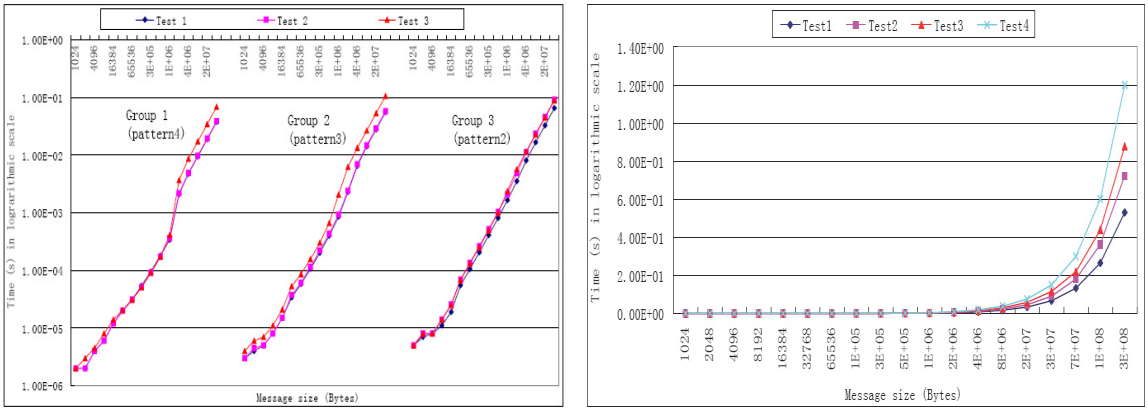
Communication performance prediction must estimate the impact of sharing resources such as memory, buses or other node-level interconnects, network interfaces, links, and switches. We use the term *communication interference* if this sharing causes performance loss.

We study how communication operations performed by tasks in the same node affect performance. In particular, we investigate if the placement of tasks in the node (i.e., how MPI tasks are distributed between processors, sockets, and dies in the node) and task inten-

sity (i.e., the number of tasks assigned to the node) affect performance. Figure 4.4 displays the performance of intra-node MPI point-to-point communication between two tasks, which we call the "observed communication operation", while there is interference from other concurrent intra-node communication operations, which we call "noise". We use the Intel MPI benchmark to perform concurrent ping-pong tests within a node, and present results from the same system as in Section 4.2. The observed communication operation and noise start at the same time and use the same message size.

Figure 4.4(a) shows that task placement impacts communication performance. In the following, we refer to the numbering of patterns presented in Figure 4.2 for conciseness. The intra-node communication in groups 1, 2, and 3 follows communication patterns 4, 3 and 2 respectively. Each group has three tests: test 1 is a reference with no noise; test 2 and test 3 each have a task pair introducing noise by performing intra-node communication. In test 2, the layout of the task pair introducing noise and the layout of the observed task pair follow pattern 6 for group 1 and pattern 5 for group 2 and group 3. In test 3, the layout of the task pair introducing noise and the layout of the observed task pair follow pattern 7 in group 1 and 2, and pattern 6 in group 3. The performance penalty of intra-node communication under noise can range from negligible to as high as 182%, depending on where the tasks that introduce noise are located.

Figure 4.4(b) shows that task intensity has a significant impact on communication performance. In these tests, the two tasks performing the observed communication operation do not share a processor. The tasks introducing noise do not share a processor either. Test 1 is again a reference with no noise, while test 2 has one pair of tasks introducing noise and test 3 has two pairs of tasks introducing noise. The intra-node communication of tasks introducing noise in test 2 and test 3 occupy a different processor than the task pair performing the observed communication. Test 4 has three task pairs introducing noise by performing intra-node communication that, together with the observed pair, fully occupy all cores. The



(a) Impact of task placement

(b) Impact of task intensity

Figure 4.4: Impact of communication interference

performance of intra-node communication is significantly affected by other intra-node communication operations running concurrently on the same node, especially if the message size is large.

We conducted exhaustive tests to cover all combinations of intra-node communication and inter-node communication under different aggregation patterns. The tests show that both intra-node communication and inter-node communication are sensitive to interference from concurrent communication operations. They also show that the performance of MPI communication is sensitive to task placement and task intensity. Thus, we must capture the impact of these factors and integrate them into our prediction framework.

Modeling and predicting communication time in the presence of noise is challenging, due to the following reasons:

- Computation/communication overlap;
- Overlap and interference of concurrent communication operations;
- Even in the absence of overlap, many factors, including task placement, task intensity, communication type (i.e., intra-node or inter-node), and communication volume and

intensity impact communication interference.

Thus, we propose an empirical method to predict a reasonable upper bound for MPI point-to-point communication time.

We trace MPI point-to-point operations to gather the endpoints of communication operations. We also estimate potential interference based on the proximity of the calls. We use this information to estimate parameters for task placement and task intensity that interfere with each communication operation for each aggregation pattern. Since we predict an upper bound, we assume that the entire MPI latency overlaps with noise from other concurrent communication operations. This assumption is reasonable for well-balanced SPMD applications, because of their bulk-synchronous execution pattern.

We construct a prediction table based on our extracted parameters, namely type of communication (intra-node/inter-node), task intensity, task placement for both communicating and interfering tasks, and message size. We populate the table by running MPI point-to-point communication tests under various combinations of input parameters. We reduce the space that we must consider for the table by considering groups of task placements with small performance difference as symmetric. The symmetric task placements have identical hardware sharing characteristics with respect to observed communication and noise communication. Figure 4.5 depicts two symmetric examples with one observed intra-node task pair and one task pair introducing noise. We mark the cores occupied by task pairs that introduce noise with dots and the cores occupied by the observed task pairs with stripes. Placement A and placement B are symmetric, so are placement C and placement D.

We use a similar empirical scheme for MPI collectives. However, the problem is simplified since collectives on MPI_COMM_WORLD involve all tasks; we leave extending our framework to handle collective operations on derived communicators as future work. Thus, we only need to test the possible task placements for specific task counts per node for the

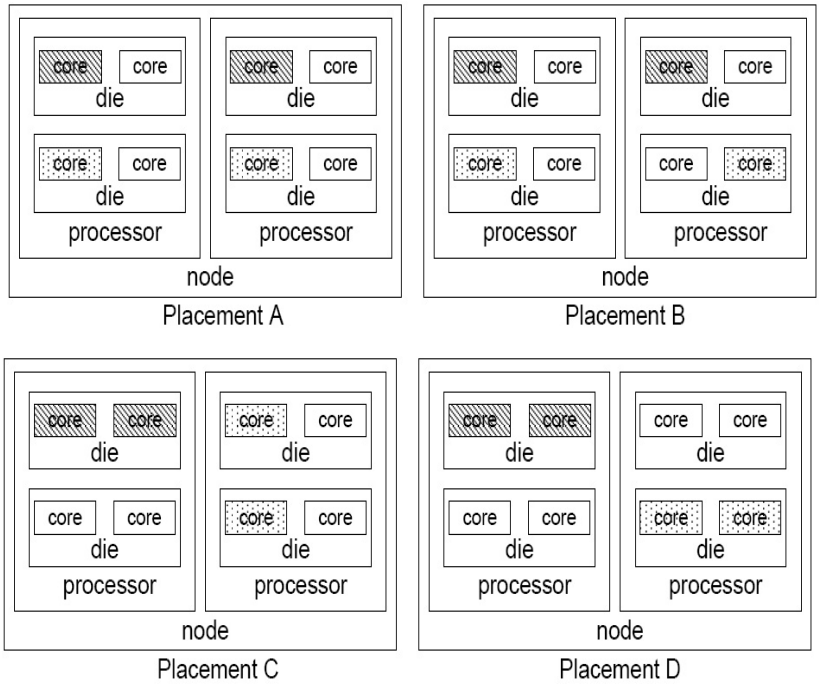


Figure 4.5: Examples of symmetric task placements

observed communication.

We apply our communication prediction methodology to the NAS 3.2 MPI parallel benchmarks and compare it with the communication time measured and reported by mpiP [183]. Figure 4.6 shows a subset of the results. We use 10 iterations of the main computation loop in bt.D.16 and ft.D.16, and 50 iterations of the main computation loop in mg.D.16, to minimize measurement error. Most MPI operations in BT and MG are point-to-point operations. We clearly overpredict communication overhead due to the overlap of computation and communication and our pessimistic prediction of the overlap between interfering communication operations. On the contrary, MPI operations in FT are collective operations and our prediction methodology is accurate in these cases.

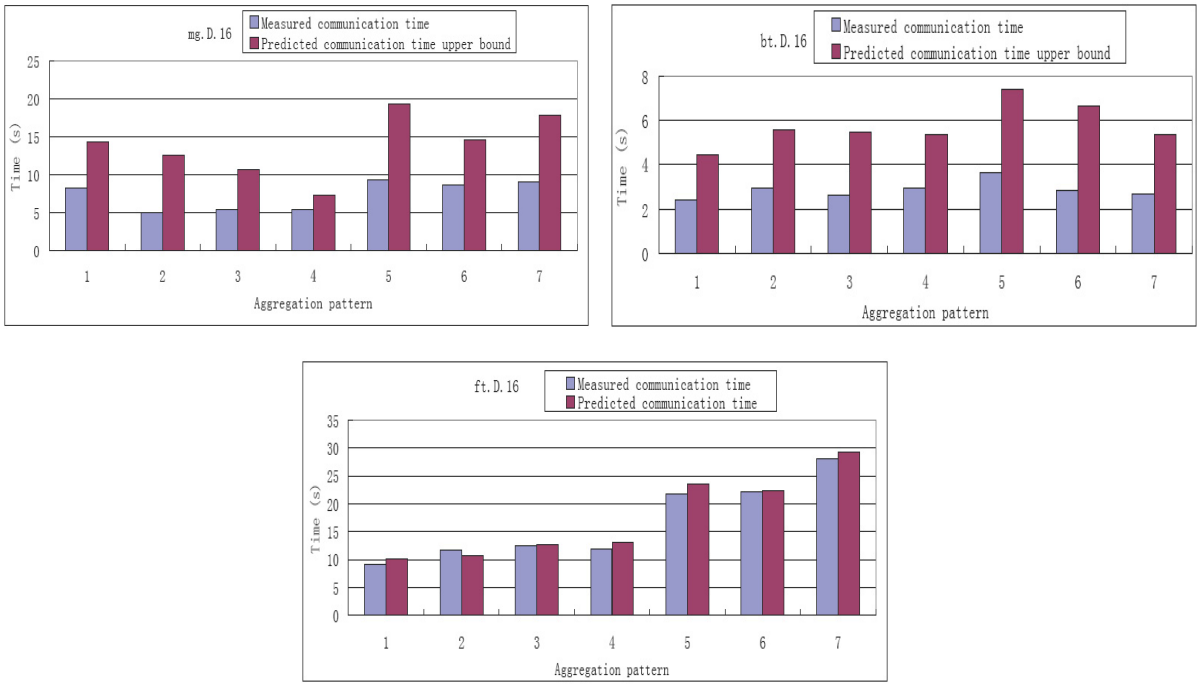


Figure 4.6: Measured vs. predicted communication time

4.4 Choosing an Aggregation Pattern

Our prediction framework allows us to predict the aggregation pattern that either optimizes performance, or optimizes energy under a given performance constraint.

We predict the best aggregation pattern based on our computation and communication performance predictions. Since our goal is to minimize energy under a performance constraint, we pick candidates based on their predicted performance and then rank them considering a ranking of their energy consumption.

We predict performance in terms of IPC (Section 4.1). To predict performance in terms of time, we measure the number of instructions executed with one aggregation pattern and assume that this number remains constant across aggregation patterns. We verify this assumption by counting the number of instructions under different aggregation patterns for 10 iterations of all NAS MPI parallel benchmarks on a node of our cluster. The maximum

variance in the number of instructions executed between different aggregation patterns is a negligible 8.5E-05%.

We compare aggregation patterns by measuring their difference to a reference pattern, where there is no aggregation of tasks in a node. We compute the difference as:

$$\Delta t = t_1^{comp} + t_1^{comm} - t_0^{comp} - t_0^{comm} \tag{4.5}$$

where t_1^{comp}, t_1^{comm} is our estimated computation time and communication time upper bound for the given aggregation pattern respectively and t_0^{comp}, t_0^{comm} is the computation and communication time for the reference pattern respectively. Comparing patterns in terms of difference with a reference pattern partially compensates for the effect of overlap and other errors of time prediction, such as the gap between the actual and predicted communication time. Our analysis in Sections 4.1 and 4.3 estimates performance for each task. For a specific aggregation pattern, Equation (4.5) uses the average computation time of all tasks and the longest communication time.

We choose candidate patterns for aggregation using a threshold of 5% for the performance penalty that any aggregation pattern may introduce when compared to the reference pattern. We discard any aggregation pattern with a higher performance penalty, which ensures that we select aggregations that minimally impact user experience. An aggregation may actually *improve* performance; obviously, we consider any such aggregations.

We choose the best aggregation candidate by considering energy consumption. Instead of estimating actual energy consumption, we rank aggregation patterns based on how many nodes, processors, sockets, and dies they use. We rank aggregation patterns that use fewer nodes (more tasks per node) higher. Among aggregation patterns that use the same number of nodes, we prefer aggregation patterns that use fewer processors. Finally, among aggregation patterns that use the same number of nodes and processors per node, we rank

aggregation patterns that use fewer dies per processor higher. In the event of a tie, we prefer the aggregation pattern with the better predicted performance. According to this ranking method, the energy ranking of the eight aggregation patterns for our platform in Figure 4.2 from most energy-friendly to least energy-friendly corresponds with their pattern IDs.

4.5 Performance

We implemented a tool suite for task aggregation in MPI programs. The suite consists of a PMPI wrapper library that collects communication metadata, an implementation of the graph partitioning algorithm, and a tool to predict computation and communication performance and to choose aggregation patterns. To facilitate collection of hardware event rates for computation phases, we instrument applications with calls to a hardware performance monitoring and sampling library.

We evaluate our framework with the NAS 3.2 MPI parallel benchmark suite, using OpenMPI-1.3.2 as the MPI communication library. We present experiments from the System G supercomputer at Virginia Tech. We described hardware of System G in Section 3.7.

We set the threshold of performance loss to 5% and use one task per node as the reference aggregation pattern. The choice of the reference aggregation pattern is intuitive, since our goal is to demonstrate the potential energy and performance advantages of aggregation and our reference performs no task aggregation. More specifically, energy consumption is intuitively greatest with one task per node since it uses the maximum number of nodes for a given run. Task aggregation attempts to reduce energy consumption through reduction of the node count. Given that each node consumes a few hundred Watts, we will save energy if we can reduce the node count without sacrificing performance. Using one task per node will often improve performance since that choice eliminates destructive interference during computation or communication phases between tasks running on the same node. However,

using more than one task per node can improve performance, e.g., if tasks exchange data through a shared cache. Since the overall performance impact of aggregation varies with the application, our choice of the reference aggregation pattern enables exploration of the energy saving potential of various aggregation patterns.

Figure 4.7 shows that our prediction selects the best observed aggregation pattern, namely the pattern that minimizes energy while not violating the performance constraint, in all cases. We indicate the best observed and predicted task aggregations with stripes. The performance loss threshold is shown with a dotted line. We achieve the maximum energy savings with sp.D.16 (70.03%) and average energy savings of 64.87%. Our prediction of the time difference between aggregation patterns for both computation and communication follows the variance of actual measured time. For FT and BT, we measure performance gains from some aggregation patterns in computation phases and our predictions correctly capture these gains.

The applications exhibit widely varied computation to communication ratios, ranging from communication-intensive (FT) to computation-intensive (LU). The communication time difference across the different aggregation patterns depends on message size and communication frequency. Small messages or less frequent communication result in a smaller communication time difference. For example, 99.98% of the MPI communication operations in lu.D.16 transfer small messages of size close to 4KB. The communication time differences in patterns 2–6 are all less than 10.0%; the communication time differences in patterns 7 and 8 (the most intensive aggregation patterns) are less than 22.7%.

The FT benchmark runs with an input of size $1024 \times 512 \times 1024$ and has MPI_Alltoall operations, in which each task sends 134MB data to other tasks and receives 134MB data from other tasks. The communication time differences in patterns 2–6 range between 28.96% and 144.1%. The communication time difference in pattern 7 (the most intensive aggregation pattern) is as much as 209.7%.

We also observe that CG is very sensitive to the aggregation pattern: different patterns can have significant performance differences due to CG's memory intensity [193]. Colocating tasks saturates the available memory bandwidth, resulting in significant performance penalties. Finally, we observe MG communication can benefit from task aggregation due to the low latency of communicating through shared-memory. In particular, communication time at patterns 2, 3 and 4 reduce by 12.08%, 25.68% and 48.81% respectively.

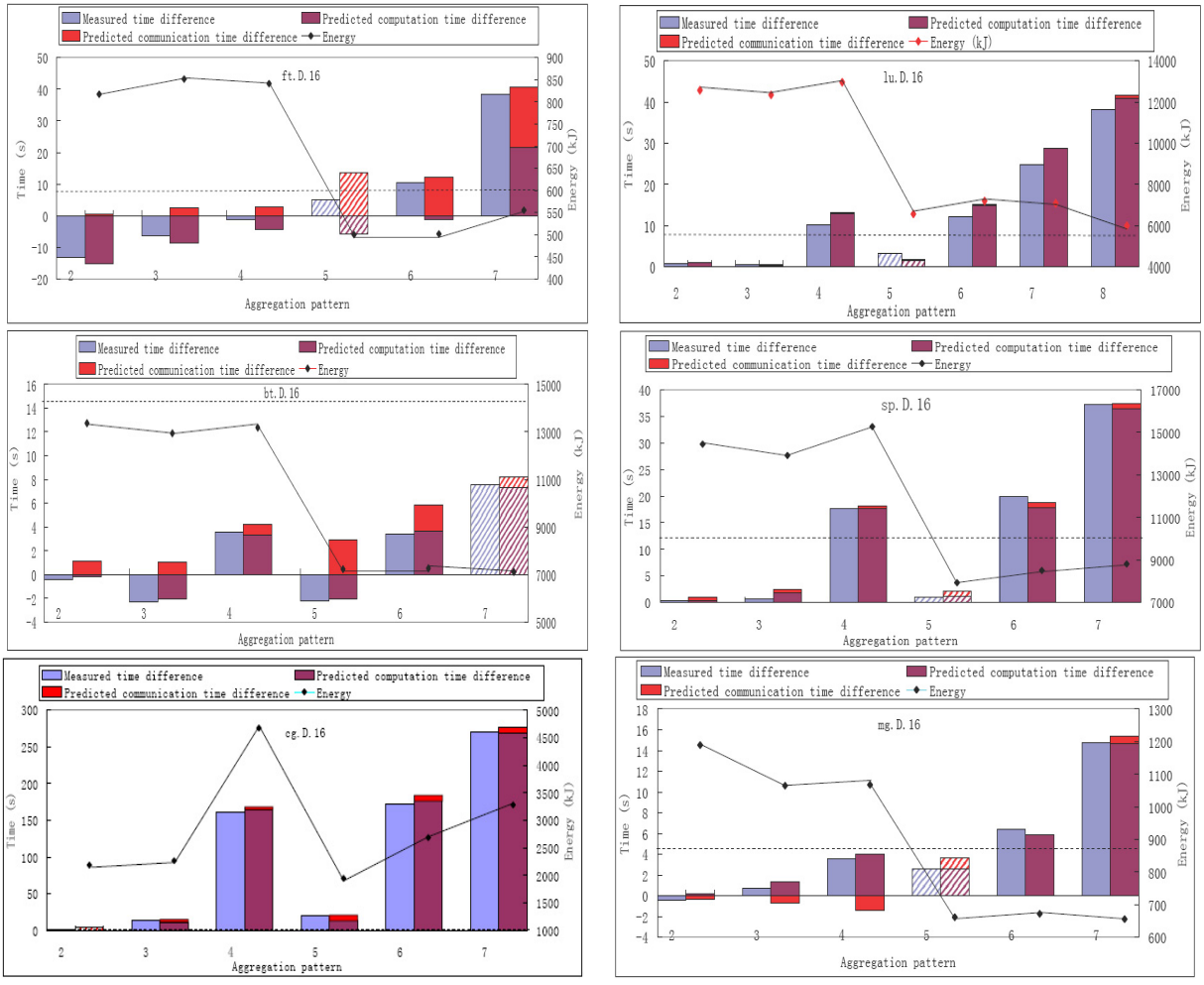


Figure 4.7: Results for the NAS 3.2 MPI parallel benchmark suite

To investigate how aggregation affects energy consumption on a larger system scale, Figure 4.8 shows the results for LU (Class D) with more nodes in strong scaling tests. As we

scale up the processor count, performance improves with more aggregation. In particular, lu.D.32 has an optimal aggregation pattern of four tasks per node with each task occupying a separate die, while both lu.D.64 and lu.D.128 have optimal aggregation patterns of eight tasks per node. This difference occurs because tasks have smaller workloads and therefore exercise less pressure on shared resources at large processor counts. This result confirms the intuition that maximal aggregation is the preferred way of running parallel jobs on large-scale clusters, for the purpose of economizing on energy and hardware resources, while sustaining performance.

We predict a different best aggregation than the observed best for lu.D.64 and lu.D.128 although our choices perform similarly. Our predictions choose the second best aggregations with energy savings of 68.46% (lu.D.128) and 67.77% (lu.D.64), while the optimal aggregations have energy savings of 80.70% (lu.D.128) and 80.65% (lu.D.64).

In both cases, our prediction of the time difference for the observed best is higher than the real time difference. As a result, we eliminate that aggregation pattern as exceeding the maximum acceptable performance loss. In general, prediction error increases as the time scale during which the program runs decreases. Using more samples for prediction is a potential solution to this problem.

4.6 Chapter Summary

High-end computing systems continuously scale up to more nodes, with more processors per node and more cores per processor. The multitude of options for mapping parallel programs to these systems creates optimization challenges. In this chapter we show that varying the aggregation and placement of MPI tasks can provide significant energy savings and occasional performance gains. Our framework predicts a task aggregation pattern that minimizes energy under performance constraints in MPI applications. We derive an empirical

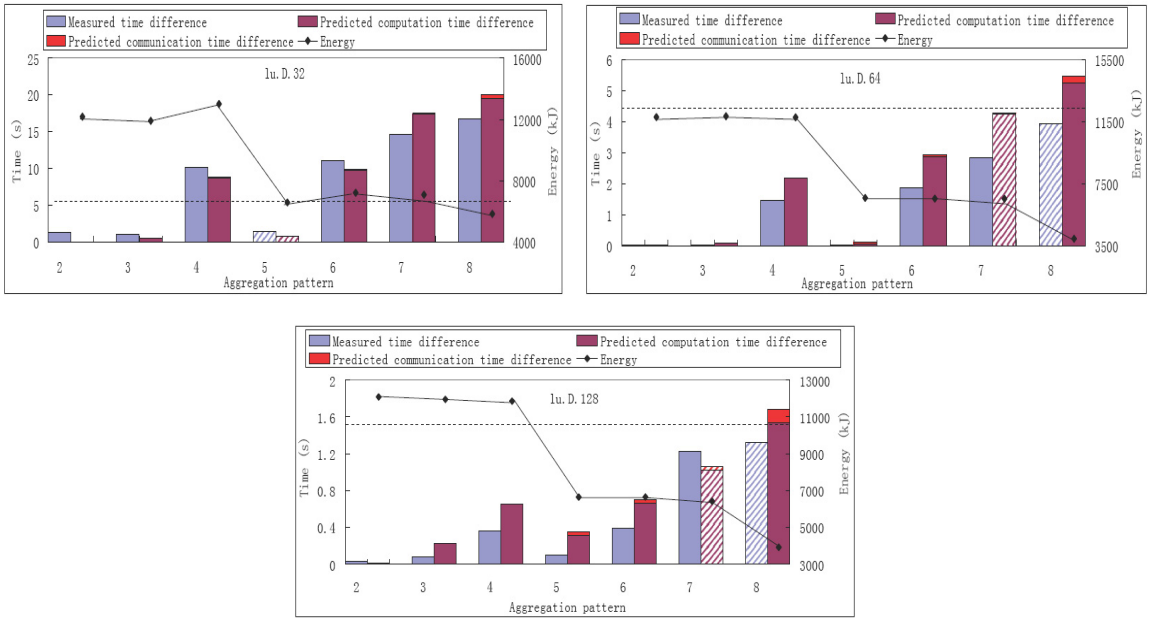


Figure 4.8: Strong scaling and task aggregation for lu.D

model to predict computation time under all feasible aggregation patterns. We formalize the problem of grouping tasks as a graph partitioning problem and solve it with a heuristic algorithm that considers communication patterns. We also derive a communication time upper bound for concurrent MPI point-to-point communication. Overall, our predictions capture the performance impact of aggregation for both computation and communication phases. Based on our prediction model, we further propose a method to select aggregations. We apply our framework to the NAS 3.2 MPI parallel benchmark suite and observe significant energy savings (64.87% on average and up to 70.03%). We also apply it at scales of up to 128 nodes and observe increasing energy saving opportunities, which allow more intensive task aggregation under tight performance constraints.

This page intentionally left blank.

Chapter 5

Scalable Memory Registration Using Helper Threads

Continuing progress in hardware technology enables the integration of multiple processor cores on a single chip. CMPs bring opportunities and challenges to high performance applications, programming models, and OSs. CMPs allow aggregation of multiple tasks within a node in order to save system power and even to improve performance [120], which is discussed in the previous chapter. CMPs also enable novel techniques that leverage idle cores to boost performance [52, 175]. However, CMPs increase pressure on shared resources at the node level, such as the network interface and the memory hierarchy. Several software [157, 158] and hardware [171, 172] schemes attempt to address this problem to achieve scaling to a steadily increasing number of cores.

Many high performance networks, such as Infiniband [92], Myrinet [27], and iWARP over TCP [35], leverage remote direct memory access (RDMA) to achieve high bandwidth transfers. RDMA enables data transfer from the address space of an application process to a peer process across the network fabric without requiring host CPU involvement. RDMA operations require *memory registration*, which explicitly identifies the memory used in these

transfers. Further, every page of the communication buffer must be pinned to prevent swapping by the OS. Pinning memory limits adaptability of the memory system and reduces the amount of memory that user processes can allocate. Unnecessarily pinning pages can underutilize memory. Pinned memory is also a precious resource: some systems constrain the maximum amount of memory that a user process can lock and some network adapters, such as the Infiniband Host Channel Adapter (HCA), limit the number of simultaneously pinned separate regions of contiguous virtual pages. The increasing popularity of GPUs for HPC makes the memory pinning problem even worse, since GPUs may also require pinning host memory mapped into the GPU address space to perform memory copies concurrently with kernel execution.

CMPs increase the challenge of the memory registration problem since the demand for registered memory grows linearly with the number of processes on the node. Some resource managers like SLURM [114] and Torque [3] can limit the amount of memory that jobs can pin. However an HPC application can require a large amount of pinned memory (see section 5.6). If we attempt to use more than one task per node on fewer nodes, aiming to reduce power consumption or to improve performance through shared memory communication [120], the application can easily exceed the pinned memory limit, which prohibits using the idle cores.

We could simply deregister the communication buffer after the RDMA operation completes. However, HPC applications often exhibit repeating communication patterns [67, 97]. Also, memory registration/deregistration is expensive [51, 132, 146] since it requires a system call and an overall linear cost in the number of pages [188]. Thus, this simple solution can incur a significant performance loss by failing to reuse the buffer.

In this chapter, we propose a novel scalable memory registration strategy that is sufficiently general to satisfy the most demanding RDMA requirements. Dynamic memory management via a helper thread introduces several challenges. First, we must identify and extract parallelism that exists between the main application thread and the helper thread,

to hide operation latency through proactive memory registrations and asynchronous deregistrations. Thus, the helper thread must predict *when* it should register *which* memory locations. Second, we must minimize communication and synchronization between the main thread and the helper thread to avoid increasing latencies on the communication critical path. Third, a new memory registration scheme should transparently benefit the application and parallel programming API (e.g., MPI). A fully transparent and dynamic solution should not require profiling information, compiler support, programmer hints, source code modification or hardware support. The helper thread mechanism should apply to any memory registration/deregistration implementation and RDMA communication protocol.

The rest of this chapter is organized as follows. Section 5.1 provides a detailed analysis of the memory registration scalability problem. Section 5.2 presents our communication context-aware predictor and compares it with an existing predictor. Section 5.3 reviews the design issues of our helper thread. Section 5.4 presents the implementation details. Section 5.5 discusses the design policies of distributing helper threads to idle cores. We evaluate our system and display the results in Section 5.6. We conclude this chapter in section 5.7.

5.1 Scalable Memory Registration

High performance networks combine DMA engines in the network adapter with user level communication protocols to achieve low CPU utilization, high bandwidth, and low latency. The DMA engine transfers data from physical addresses in main memory to the network adapter. Virtual addresses of communication buffers must be translated into physical addresses. The memory pages corresponding to these addresses must be pinned for the duration of the transfer to prevent the OS from paging them to disk. *Memory registration* refers to the process of address translation and pinning. We now outline common memory registration

strategies.

Bounce Buffer: MPICH [136], MVAPICH [88] and Open MPI [71] use the *bounce buffer* method in which the connection set-up phase pre-registers a set of RDMA buffers and exchanges their addresses and associated keys between communication peers. MPI communication then copies from/to the user's buffer in user space to/from the bounce buffers and uses RDMA on the bounce buffers to transfer the data. This method limits the registered memory size to the size of the bounce buffers. It incurs registration costs only at startup, which potentially amortizes them over many RDMA operations. However, this method incurs significant memory copy costs, particularly for large messages. Thus, it best suits small messages, including messages for protocol control. We could apply our helper thread strategy to small messages to eliminate the bounce buffers. However searching preregistered buffers and identifying the communication call site, as with our helper threads (see section 5.3), entails sufficient overhead that the bounce buffers are preferable. Thus, we only use helper threads for messages larger than a threshold (16KB in our experiments).

Registering Memory on Demand: Many RDMA libraries register memory on demand [71, 88, 136]. This alternative scheme registers user buffers at the communication call site, which adds the overhead of memory registration to the communication critical path. Some implementations [71, 136] leave user buffers pinned after the communication, which can amortize the cost if the user buffer is reused often. However, this choice increases registered memory size, even if the buffer is only used once.

Hardware-Assisted Memory Registration: Quadrics [4] combines a hardware TLB in the network adapter with a tuned virtual memory subsystem to allow hardware to pin pages, to initiate page faults and to track page table changes. This approach avoids memory registration overhead and minimizes pinned memory size. However, it increases hardware complexity and cost and requires platform and OS-specific modifications.

Other techniques, such as the RDMA pipeline protocol, focus on reducing memory regis-

tration cost. For example, the RDMA caching pipeline [188] combines the pipeline protocol with on demand memory registration, leaving registered memory pinned. Our helper thread approach aims at reducing registered memory size and avoiding (de)registration overhead.

Figure 5.1 depicts a microbenchmark that captures the limitations of prior memory registration schemes that our helper thread approach overcomes. In this common execution pattern of HPC applications, point-to-point communication repeatedly exchanges data between locations. Figure 5.2 shows the registered memory size over time on the left and the communication time on the right when we run the microbenchmark on two nodes connected with Infiniband using Open MPI 1.4 and two memory registration schemes (Section 5.6 describes additional hardware configuration details). We measure sender side time for a 5MB message. The bounce buffer occupies region 1 of registered memory, while memory registered for communication with large messages occupies region 2. With the "leave pinned" registration on demand approach, the registered memory increases steadily and later remains constant. The "no leave pinned" approach registers less memory, but takes 25.83% longer. Our helper thread approach breaks this dichotomy.

If each MPI communication call site i of M total sites must register m_i bytes for RDMA communication then the maximum registered memory size is $\sum_{1 \leq i \leq M} m_i$ with the leave pinned strategy. In theory, the helper thread registers memory for a communication call site in advance and deregisters the memory after the communication. Thus, its peak registered memory size is $\max_{1 \leq i \leq M} m_i$.

As further motivation, consider a system in which each node has 8GB of physical memory and 8 cores and we limit the registered memory size per process to 3GB, which is at least as large as would be applied in practice. Otherwise we can incur a severe shortage of physical memory – using two MPI processes per node, each of which pins 3GB memory, would pin 75% of the physical memory. As many large scale systems do not include local disk, this large registered memory footprint could leave too little unregistered memory for the application

```
/*Sender: */

/*The sender and the receiver begin
  communication at the same time*/
MPI_Barrier(...);

for(iter=0; iter<10; iter++)
{
 MPI_Send(..., buffer_1, ...);
 random_comp(); /*lasting 1 sec*/
 MPI_Send(..., buffer_2, ...);
 random_comp();
 MPI_Send(..., buffer_3, ...);
 random_comp();
}
```

```
/*Receiver: */

/*The sender and the receiver begin
  communication at the same time*/
MPI_Barrier(...);

for(iter=0; iter<10; iter++)
{
 MPI_Recv(..., buffer_1, ...);
 random_comp(); /*lasting 1 sec*/
 MPI_Recv(..., buffer_2, ...);
 random_comp();
 MPI_Recv(..., buffer_3, ...);
 random_comp();
}
```

Figure 5.1: Microbenchmark pseudocode

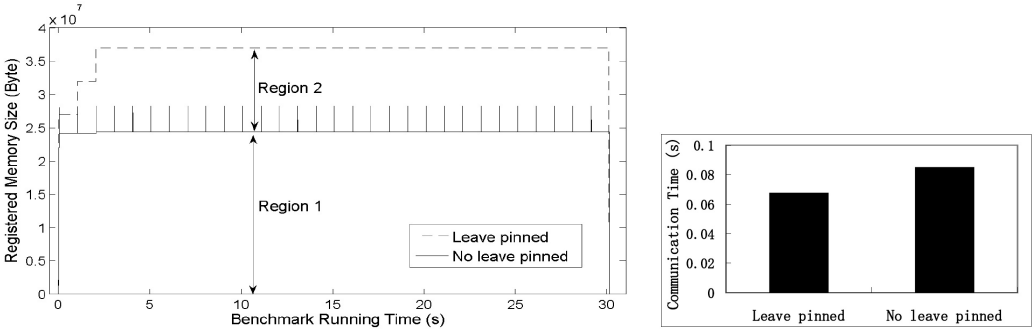


Figure 5.2: Traditional memory registration sizes and communication times

to run. Even if paging is possible, it would entail a substantial performance loss. However, even with this large memory registration limit, we cannot run problem size D of NAS PB FT with a 2x16 2D processor layout with the standard Open MPI installation, as Figure 5.3 shows, since a single FT task requires 3.58GB of pinned memory. As we show in section 5.6, our approach enables the execution of the application even with the memory registration limit.

5.2 Context-Aware Predictor

HPC applications interleave communication and computation and attempt to overlap communication with computation using techniques such as non-blocking MPI communication and progress threads [158]. Our helper thread scheme offloads memory registration overhead to overlap it with computation. Thus, we register and deregister memory outside of

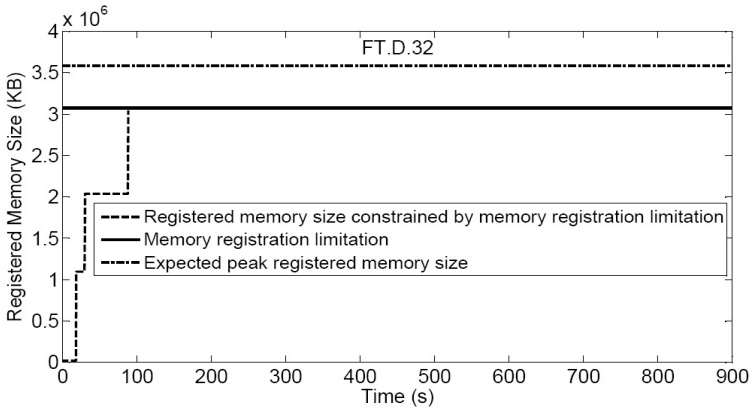


Figure 5.3: Memory registration limit prevents application execution

communication phases. We adopt a prediction-based approach. Our design attempts to predict execution points that require memory based on iterative execution patterns in HPC applications [67].

We present a time-based, context-aware predictor that attempts to identify iterative usage patterns of each specific registered memory region from MPI message streams and to predict the future registration time of each region. The predictor must dynamically detect the registration period of each region. We then use this period to predict the next registration time.

Alternatively, we could leverage temporal patterns in MPI messages [67], thus saving registered memory without considering the time between communication operations. Specifically, we could register a set of recently referenced memory regions, similarly to an LRU cache. However, this method cannot minimize registered memory size. An application may have large time gaps between RDMA communication operations, during which we can deregister previously registered memory. We require the time information to design an effective strategy to minimize registered memory size throughout the lifetime of an application.

A simple predictor design would record the registration time for each memory region and then use the time between consecutive registrations. However, this design cannot han-

```
main computation loop
{
  ...
  inner loop1
  {
    ...
    MPI_Comm1(..., buf1, ...);
    ...
  }
  ...
  inner loop2
  {
    ...
    MPI_Comm2(..., buf2, ...);
    ...
  }
}
```

Figure 5.4: Loop nests

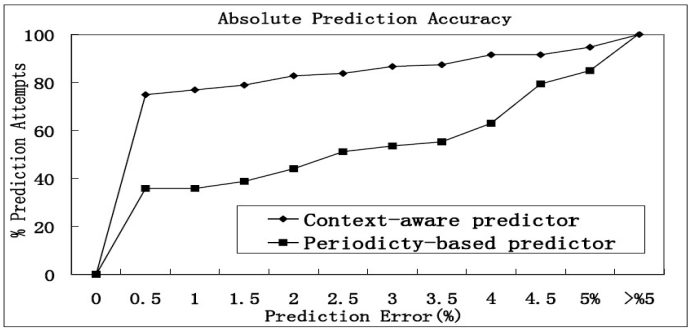


Figure 5.5: Prediction accuracy comparison

dle complex memory access patterns. Different control paths may use the same memory region with different periods. Thus, we associate call site information with each registered memory region to detect the periodicity, instead of only identifying the memory region. We hash the call site stack frame addresses and the registered memory region address in a call site's *signature*. This solution distinguishes between uses of the same memory region with different registration periods. The helper thread performs the hashing operations so that we minimize latency into the application critical path, which only includes recording stack frame addresses.

This approach generally works well but can poorly predict the registration period for cases such as the nested loops that Figure 5.4 shows. We cannot predict the registration time of the communication call site of the first iteration of inner loop 1 from the per iteration time of that loop. However, that time is the registration period of all other instances of that call site. The first iteration of inner loop 1 follows a full execution of loop 2 except for when it is part of the first iteration of the outer loop. This call site has three different registration periods. Thus, we extend the communication context information to include the previous communication point to handle call sites with multiple periods. By hashing call site information from the previous (i.e., communication type and memory address) and current (i.e., stack frame addresses and the registered memory address) communication call sites in

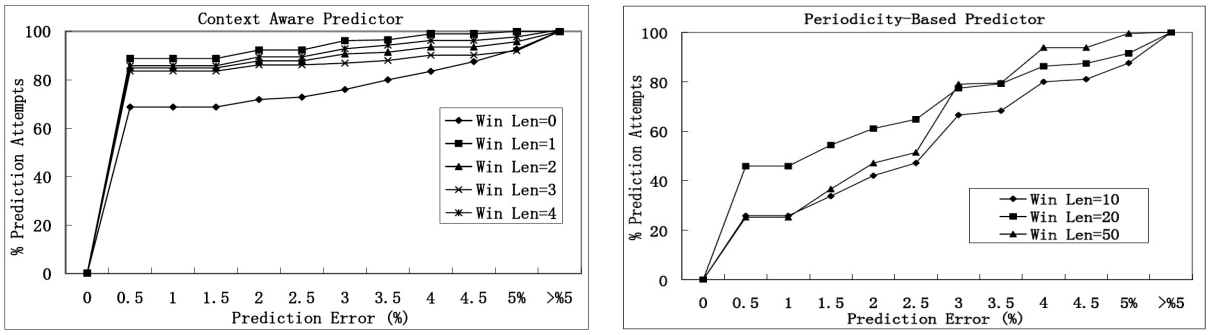
the signature, we distinguish between communications with the same stack frame addresses but different registration periods due to control flow differences.

We use a communication context window to record communication streams (Section 5.4). We analyze the sensitivity to the window length (i.e., how many adjacent call sites we consider) for the NAS CG benchmark, as Figure 5.6(a) shows. "Win Len=0" indicates that we only use current call site information, which decreases the percentage of predictions with error less than 0.5% by 20%. Thus, we must consider previous call sites although accuracy increases little if we use more than the immediately preceding one. Based on this and similar results for five other NPB benchmarks, we use a window length of two, which incurs little additional cost for potentially greater accuracy in rare cases, for the rest of the chapter.

We compare our predictor with a periodicity-based predictor that records a window of communication streams to detect iterative patterns [67]. This predictor compares a communication subsequence with another subsequence shifted by m samples, where m is a parameter less than the window size. It detects an iterative pattern with period m if the two subsequences are identical. We apply this technique to detect patterns and their time periods to predict memory registration time. We analyzed the sensitivity of this predictor to its window length. We found that it is highly sensitive to this parameter at any desired accuracy, as Figure 5.6(b) shows. Further, it has a long learning process that requires a full window and a complete pattern. Our predictor has a short learning process since its context window is short and its predictions are independent of the pattern frequency.

Figure 5.5 compares results of our predictor with the periodicity-based one for six NAS parallel benchmarks. Both approaches are accurate: 84.93% of periodicity-based and 94.68% of our predictions are within 5% of the observed period. However, 74.89% of our predictions have error less than 0.5% while only less than 40% of the periodicity-based predictions achieve error that low.

Discussion: Our approach so far assumes that the workload per iteration is stable



(a) Context-aware predictor (b) Periodicity-based predictor

Figure 5.6: Sensitivity analysis of the two predictors

and, therefore, memory registration is periodic. Some HPC applications (e.g., AMG in the Sequoia benchmark suite [113]) do not have a constant workload across iterations. Although these applications do exhibit iterative memory registration patterns, the registrations may not be periodic. In this case, we use the shortest time between registrations in the same context.

5.3 Helper Thread Design

General Description: Given the predicted registration time, we can design an algorithm for the helper thread. This design must balance between registering a memory region before a communication operation uses it and delaying memory registration as long as possible. If we do not register memory before a communication uses it, the main thread must register it on demand at the cost of additional latency; registering memory too soon increases the peak memory registration size.

Our design uses two queues: a deregistration queue (DQ) and a registration queue (RQ). The helper thread processes these queues in turn and switches between them based on a cost estimate for processing items from each. We use fine-grained time control to ensure that we achieve the necessary balance.

Memory Deregistration: The main thread places information about registered memory (registration time, address, size and adjacent communication point information) into the DQ, after the completion of a communication operation. The helper thread removes this information and computes its signature, which it uses to extract information from the *call site hash table*, which stores predicted registration periods for call sites. The helper thread must complete the registration of each call site before its *registration deadline*, which is its last registration time plus its period.

The helper thread uses the registration deadline to determine if it should deregister a memory region, which multiple communication call sites may use. We deregister a memory region and insert it into the RQ with its registration time set to the registration deadline if a *deregistration condition* is satisfied. This condition requires that the current time plus the estimated processing delay for memory registration is not later than the earliest registration deadline associated with the region. The helper thread finds the nearest future registration time for a region in the RQ after registering it and updates the earliest registration time. The first time we register a memory region, we do not have a predicted registration period for it. Thus, we always deregister it, which saves registered memory from those operations that register regions only once. After processing one item from the DQ, the helper thread switches to the RQ to check registration requests, in case any registration deadline is near.

Memory Registration: the RQ is ordered based on the registration time of each item in the list. The helper thread processes the RQ starting from the item with the earliest registration time. It delays registering a memory region and switches to DQ processing if a *registration condition* is satisfied: the current time plus the estimated time for processing one DQ item plus the estimated processing delay for memory registration is earlier than the region's registration deadline. Otherwise, the helper thread registers the memory and updates its associated earliest registration time. We track registered memory information in a *registration cache* for fast reference.

Adjusting Registration Deadlines: Some HPC applications have communication-intensive phases in which multiple communication sites have similar registration deadlines. Since the helper thread tries to delay memory registration until the deadline, it may not be able to register all memory regions on time, in which case we incur additional latency on the communication critical path. We avoid this cost by adjusting deadlines when we insert a new item into the RQ. The registration time gap between any two items must be longer than the cost of processing one RQ item plus the cost of processing one DQ item, so that we can process both the DQ and the RQ in time to make the registration deadlines. We decrease registration deadlines when necessary to meet this condition.

Time Control and Estimation: Our design critically depends on estimated DQ and RQ processing times. The following inequality describes our registration and deregistration conditions:

$$t_c + t_p \leq t_d \tag{5.1}$$

Where t_c is the current time, t_p is the processing time by the helper thread and t_d is the registration deadline.

We can perform deregistration if the inequality is true. For the deregistration condition, t_p includes: (1) the deregistration cost ($t_{deregister}$); (2) the cost of inserting the registration request into the RQ (t_{insert}); (3) registration cost ($t_{register}$); and (4) the cost of updating the earliest registration time for a memory region (t_{update}). Similarly, we can delay registration and switch to processing the DQ if the inequality is true. For the registration condition, t_p includes: (1) the cost of searching the hash table and updating call site information (t_{hash}); (2) the cost of searching the registration cache (t_{cache}); (3) $t_{deregister}$; (4) t_{insert}; (5) $t_{register}$; and (6) t_{update}.

We model the memory (de)registration cost with $t_{(de)register} = a * p + b$, where a is the (de)registration cost per page, b is the overhead per operation, and p is the size of the memory

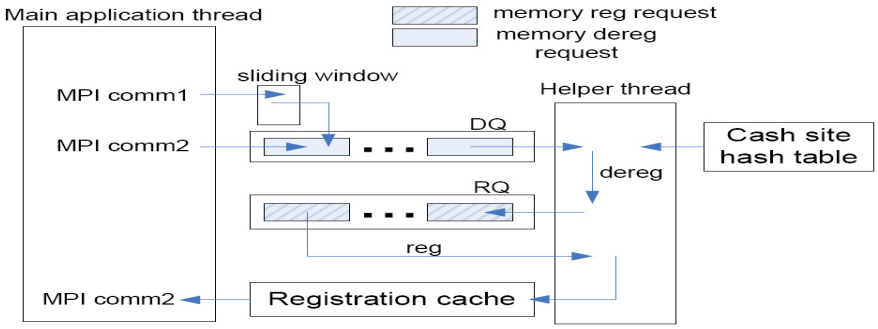


Figure 5.7: An illustration of the helper thread implementation

region in pages [189]. We determine a and b through linear regression. The RDMA library or device driver may include other optimizations to improve the performance of (de)registration. While these optimizations can make the cost of (de)registration irregular, we attempt to keep our algorithm independent of these optimizations by estimating the maximum potential cost. Alternatively, we could use a more complex (de)registration model.

The number of queue items that we must compare in each step impacts t_{insert} and t_{update}, which represent the time for queue operations. Each comparison has a nearly fixed number of instructions, which is easy to estimate. We conservatively assume that we compare every item in the queue for the insert/update operation to simplify the overall estimates, although an algorithm like binary insertion may involve fewer items. We can also easily estimate t_{hash} if we assume that hash collisions are rare. We estimate t_{cache}, which is related to the number of cache items that the algorithm must compare, similarly to t_{insert} and t_{update}.

5.4 Implementation

Figure 5.7 illustrates our helper thread implementation, which we discuss in this section.

5.4.1 Synchronization Issues

Thread synchronization latency is an important design consideration for helper thread software. We implement the DQ as a shared, lock-less circular queue. The main thread moves the head pointer when inserting an item and the helper thread moves the tail pointer when removing an item. Initially, the head pointer and the tail pointer point to the same position. The helper thread compares the two pointers to determine if the DQ has new items. If so, it moves the tail pointer forward and dequeues one item for processing.

A circular queue needs a lock to prevent the head pointer from overwriting unused data. However, most HPC applications interleave communication with computation. Since the computation phase is usually much longer than the processing of a queue item in the helper thread, we usually move the tail much faster than the head so overwriting is unlikely. Further, if an application has short (or no) computation phases, so that we move the head pointer more quickly then the time between the communication call sites is too small to perform memory deregistration and registration without performance loss. Thus, we can safely overwrite and, as a result, skip old queue entries. Currently, we only use one helper thread and hence we do not need a lock for the tail pointer. We also do not need a lock for the head pointer if the MPI implementation does not provide `MPI_THREAD_MULTIPLE` support, which ensures only one thread will access it at any time. Since the helper thread reads the head pointer, RAW and WAR hazards may happen. In our case, a RAW hazard only temporarily delays the processing of a new item, which does not add any latency to the critical path. A WAR hazard also has no negative impact since the helper thread tests the difference between the head and the tail pointers.

The main thread and the helper thread share the registration cache as well as the DQ. The helper thread puts registered memory information (address, size and a pointer to the registered memory region) into the cache prior to communication. The main thread obtains

the registered memory region pointer from the cache. Thus, we must lock the memory region pointer. We use atomic instructions (compare and swap) to implement a lightweight lock and spin if it is held. With this lock, the synchronization latency is $0.072\mu s$ when two threads contend for the lock and $0.018\mu s$ without contention. This lock overhead is low since MPI communication latency for large messages (larger than our 16KB threshold) is at least tens of microseconds with state-of-art hardware [112, 168].

5.4.2 Predictor Implementation

The predictor must collect context information by using a sliding window with length equal to the number of communication call sites for which it constructs signatures. Each call site adds information to the window. When the predictor computes a call site signature, it obtains data for adjacent call sites from the window. We associate each call site leading to memory registration with a call site hash table entry. The hash table entry records the last memory registration time and the predicted period. We update the last memory registration time when we begin memory registration for the call site. We also compute the time between the current memory registration time and the last one, which we record as the new prediction if it is less than the current predicted period. We use the shortest observed time because effects such as networking perturbation, OS noise and computation workload vibration may make the period irregular. Choosing the shortest period reduces the probability that we miss the real registration deadline.

5.4.3 Overhead Analysis

We introduce four operations on the critical path of the main thread. First, we read the time stamp counter at the RDMA communication call sites, which typically costs between 150 and 200 clock cycles — or less than $1\mu s$ on a 1 GHz processor. Second, we acquire a syn-

chronization lock for a registered memory pointer, which takes less than $0.1\mu s$, as discussed in Section 5.4.1. Third, we record the communication type and the buffer address in the context window, which is negligible compared to the cost of the actual MPI implementation and communication operation. Fourth, we record the stack frame address at the RDMA communication call site, which involves a few accesses of the frame pointer register and copying several bytes of data. This overhead is also small compared to the communication time of large messages. Nonetheless, we can overlap these overheads, other than collecting the context window information, with computation for nonblocking MPI communication.

We measure our scheme's overhead with the microbenchmark that Figure 5.1 shows. We use a 17KB message size (marginally larger than the minimum size at which we use our helper thread). We disable the helper thread, but keep the above four operations in the main thread, so that any performance benefit of the helper thread does not offset its overhead. We compare performance with the original implementation without the four operations. We run the test 100 times and compute the average communication time at the sender site and find that the overhead is 14.02%, which primarily comes from recording stack frame addresses. This test gives an upper bound on overhead since real applications are likely to be less communication-intensive or to use larger message sizes. The same microbenchmark finds much lower overhead with larger messages (4.58% for 1MB messages and 1.22% for 4MB messages).

5.5 Leveraging Idle Cores Efficiently

In principle, helper threads use idle core cycles that are not used for computation or communication. We should not sacrifice application performance by dedicating cores to helper threads. If no idle core is available, we do not use the helper thread. However, with an increasing number of cores per node in high-end computing environments, idle cores are

often available due to application scalability limits [120]. In this section, we explore design alternatives for using idle cores to run helper threads, while varying the numbers of helper threads and idle cores dedicated to them.

When we have at least as many idle cores as helper threads, each helper thread solely occupies a core. This unshared core distribution allows each helper thread to explore registered memory reduction opportunities fully. We show in section 5.6 how helper threads reduce the registered memory size for each MPI task under this distribution.

When we have fewer idle cores than helper threads, distributing helper threads across the idle cores becomes more challenging; we explore two possible policies. The *aggressive policy* tolerates the excess helper threads. Each task spawns a helper thread, without considering resource consumption. Helper threads that reside on the same core take turns conducting memory operations, depending on the OS scheduler. The *conservative policy* restricts helper threads to at most one per idle core through the coordinated elimination of helper threads in some processes if necesssary.

The aggressive policy must adjust the registration/deregistration condition (section 5.3) to consider concurrent helper threads. Assuming a round robin scheduling policy with a time slice of length t_{slice} for helper threads on the same core, we change the registration/deregistration condition as follows:

$$t_c + t_p + n(t_{slice} + t_{cs}) \leq t_d \tag{5.2}$$

where n is the number of helper threads on the core and t_{cs} is the context switch overhead. We assume that an active helper thread does not know when its current time slice will end. We guarantee that we do not miss deadlines by requiring t_d to be no earlier than the thread's *next* time slice to meet the (de)registration condition. This conservative assumption ensures that we can safely deregister any item for which the inequality holds now and register it again in a future time slice (the deregistration condition), or delay its registration to a future time

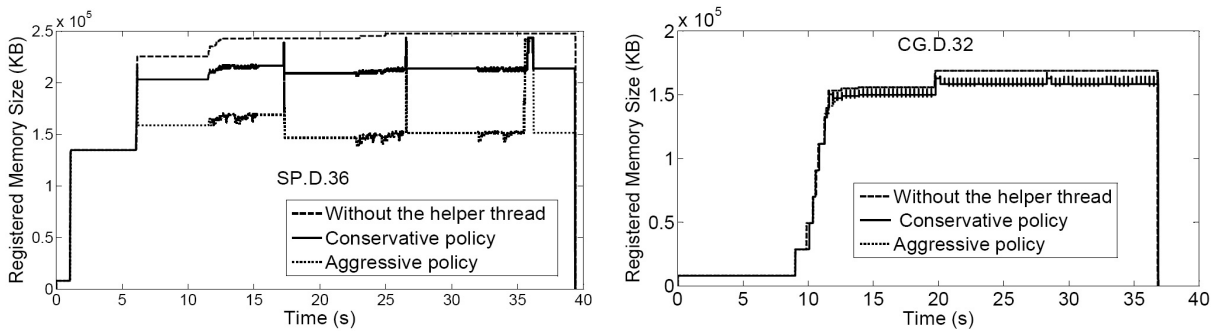


Figure 5.8: Registered memory sizes with different distribution policies

slice (the registration condition).

The conservative policy uses our original registration/deregistration condition since it still has only one helper thread per idle core. Instead, we must coordinate tasks to decide which ones retain helper threads. We chose to provide helper threads for the tasks that register the most memory. We implement task coordination in shared memory, where each helper thread posts the registered memory size after finishing a fixed number of communication operations. We continue running the threads associated with the largest memory registrations and stop all others (we randomly break ties).

We apply both policies to NAS parallel benchmarks SP and CG. Figure 5.8 and Table 5.1 show the results. Our test platform has 8 cores per node (see Section 5.6 for more details). We use 6 nodes with 6 tasks per node for SP. For 32 CG tasks, we use 6 nodes total: 2 with 6 tasks and 4 with 5 tasks (the task distribution is unbalanced because CG requires a power of 2 task count and we require that each node has at least 5 tasks so that we have fewer idle cores than tasks). Figure 5.8 shows results from the nodes with 6 tasks per node. We display only two time steps or two computation iterations in order to illustrate the variance clearly. We report the registered memory size for the entire node. Under the aggressive policy, we distribute the helper threads as evenly as possible between the available idle cores so each idle core hosts 3 helper threads.

The results show that the aggressive policy saves more registered memory on average than

	No helper thread	Conservative policy	Aggressive policy
SP.D.36	18.60	18.60	18.62
CG.D.32	17.08	17.09	17.09

Table 5.1: Execution time (s) with different distribution policies.

the conservative policy for SP. For CG, the aggressive policy does not save memory, while the conservative policy does. SP does not generally interleave communication with computation and has a relatively long time between communication calls. Thus, the aggressive policy does not miss memory saving opportunities even though the helper threads share cores. The conservative policy saves less memory since only a third of the tasks have helper threads. Alternatively, CG is communication intensive so the aggressive policy cannot deregister memory without introducing latency into the communication critical path.

In general, performance of the distribution policies depends on application characteristics and OS scheduling. The aggressive policy assumes round robin scheduling with identical time slices. Linux determines the time slice length based on process priority. Although helper threads may have the same static priority, the dynamic priority may change, depending on how often the threads submit and wait on I/O requests. Thus, helper threads may not be scheduled round robin since the OS attempts to schedule threads with higher priority earlier. Our design also critically depends on time information but a helper thread may be descheduled immediately after reading the clock, making the time stale when it resumes execution. Since the thread has no knowledge of OS scheduling decisions, it can compute an inaccurate (de)registration condition. The conservative policy does not have these challenges but cannot save as much memory when they are not relevant, as with applications that are not communication intensive. Since memory registration sizes are likely to be more significant for communication intensive applications, we favor the conservative policy.

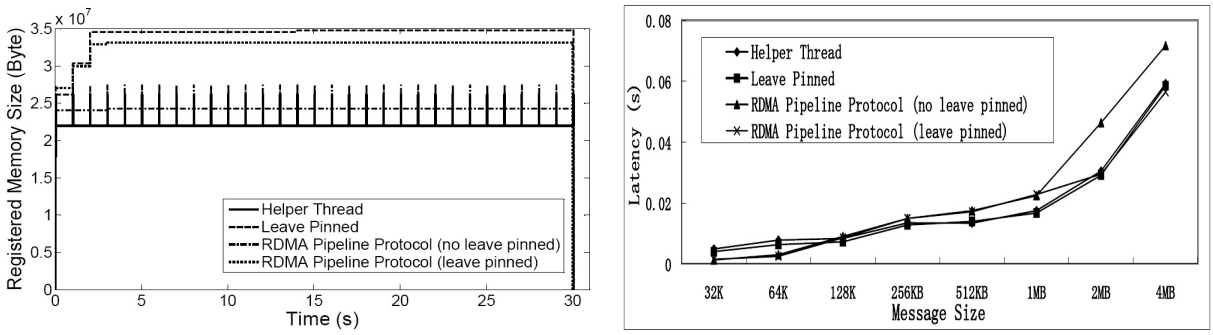
5.6 Performance

We implement our memory registration scheme in Open MPI 1.4 and evaluate its performance with microbenchmarks and the NAS parallel benchmarks. Our evaluation platform is Virginia Tech Systemg G. We described the hardware of System G in Section 3.7. Our registration scheme provides a helper thread per MPI task. The helper thread shares the L2 cache with the associated task. In particular, the task and helper thread share the 5.6KB DQ data structure, which easily fits in L2.

5.6.1 Microbenchmark Tests

Our first set of experiments uses the benchmark that Figure 5.1 shows. We compare the performance of the memory registration scheme using helper threads, the Open MPI default scheme and the RDMA pipeline protocol. The Open MPI default setting registers the memory on demand and leaves registered memory pinned for future use (our charts label this scheme "Leave Pinned"), which many communication libraries use. The RDMA pipeline protocol represents a technique to reduce memory registration overhead. We both leave memory pinned and unpinned with the pipeline protocol to capture its full potential range. We conduct tests with both buffer reuse and no buffer reuse. In these tests "buffer reuse" means that we use the same user buffer for communication across loop iterations; "no buffer reuse" means the opposite. We report the average value of 100 runs of each test.

Figure 5.9 depicts the results for the buffer reuse tests. Our helper thread approach performs similarly to Leave Pinned, which demonstrates the low helper thread overhead. With message sizes larger than 1MB, the overhead is less than 5%. Also, the helper thread scheme reduces the registered memory size compared to Leave Pinned by 24.75%. The pipeline protocol implementation outperforms the other schemes for message sizes less than 256KB. Open MPI does not register memory for these messages with this protocol. Instead,



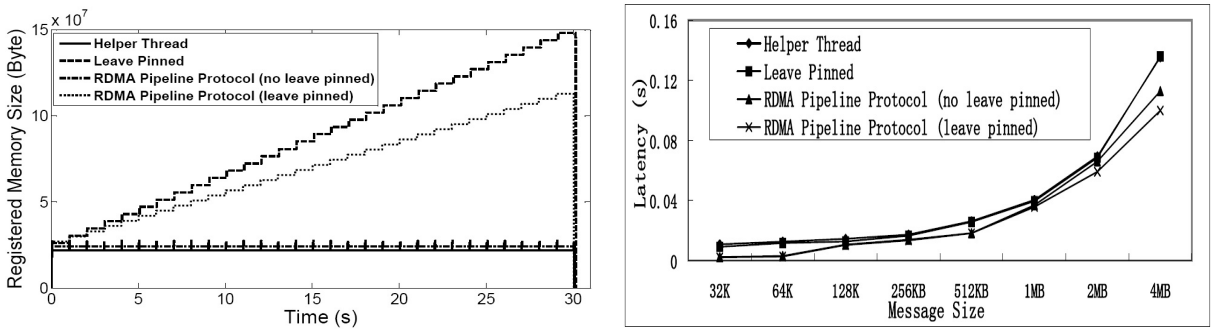
(a) Registered memory size (4MB messages)

(b) Message latency

Figure 5.9: Buffer reuse tests

it uses the bounce buffer and avoids registration and deregistration costs. Leave Pinned and our helper thread scheme do not perform as well because they search a memory registration cache to find preregistered memory and have a longer function call chain. However, these schemes outperform the pipeline protocol for message sizes of 512KB and 1MB, despite the latter still using bounce buffers, which incurs data copying overhead. The pipeline protocol registers memory regions for larger message sizes. The pipeline protocol has low performance if it does not leave memory pinned since it repeatedly pays registration and deregistration costs. However, the pipeline protocol does not clearly benefit from its memory registration optimization if it leaves memory pinned. In terms of registered memory size, the helper thread saves more memory than the pipeline protocol (by 3.84% for no leave pinned and 21.04% for leave pinned). The pipeline protocol (leave pinned) uses less registered memory than Leave Pinned and uses more registered memory than the helper thread, because the pipeline protocol registers an extra bounce buffer that it leaves pinned to transfer a message data segment for the first communication. This buffer reduces the registered memory for later communication.

Figure 5.10 depicts the results for the no buffer reuse tests. Figure 5.10 shows that the registered memory size for Leave Pinned grows throughout the no buffer reuse test. The peak registered memory is 4.62× more than with the helper thread scheme, while the two



(a) Registered memory size (4MB messages)

(b) Message latency

Figure 5.10: No buffer reuse tests

schemes have comparable performance. Also, the helper thread scheme and Leave Pinned both register memory on the critical path since the memory access pattern is irregular. Their performance is worse than the pipeline protocol, which demonstrates the pipeline protocol effectively reduces registration costs. However, the pipeline protocol (no leave pinned and leave pinned) uses more registered memory (3.84% and 3.31× more respectively) than the helper thread scheme. The pipeline protocol (leave pinned) uses less registered memory than Leave Pinned, due to the bounce buffer mentioned above.

To summarize, the helper thread scheme has the minimum registered memory size in all cases. With predictable memory usage, as HPC applications often exhibit, the helper thread performs comparably to the scheme with the best performance for large messages. Even with irregular memory usage, the helper thread achieves performance similar to the Leave Pinned scheme typically used in most communication libraries.

5.6.2 NAS Parallel Benchmarks

We apply the helper thread scheme for memory registration to the NAS parallel benchmarks. Figure 5.11 shows registered memory variance from MPI_Init to MPI_Finalize for Leave Pinned and our helper thread scheme. We only show two time steps or two iterations of the main computation loop (except FT (1D processor layout) having complete iterations)

	BT.D.4	LU.D.4	SP.D.4	CG.D.4	FT.B.4
Helper thread	265.98	219.39	140.59	200.04	79.47
Leave Pinned	266.87	218.80	141.10	202.31	79.50

Table 5.2: Execution time (s) for 5 NAS benchmarks.

to display the variance clearly. The other time steps or computation iterations have similar variance. We run one MPI task per node with four nodes total.

Our helper thread scheme reduces registered memory by 16.13%, 18.43% and 8.68% for BT, LU and SP. These benchmarks have communication phases before/after long computation phases, repeating patterns that the registered memory variance clearly shows. Our predictor captures these patterns and directs the helper thread to register/deregister memory on time, which reduces the peak registered memory size. CG and FT are both communication intensive applications that interleave computation with frequent communication. The helper thread scheme does not reduce the peak registered memory size for CG, due to its communication intensity. However, it does reduce the average registered memory size compared to Leave Pinned. Collective operations consume significant registered memory with FT, for which we reduce the peak registered memory size by 31.0%. We incur negligible performance loss for all benchmarks (Table 5.2). Proactive memory deregistration (instead of leaving memory registered until MPI_Finalize) slightly improves performance (1.12%) for CG. On a system with a higher deregistration cost, such as Myrinet/GM [57], we expect a larger performance improvement.

Figure 5.11(f) displays the scenario that we have shown in section 5.1. Without the helper thread, the application cannot continue execution since a single task pins 3.58GB memory; with the helper thread, we can constrain the registered memory size beneath the memory registration limit so that the application completes.

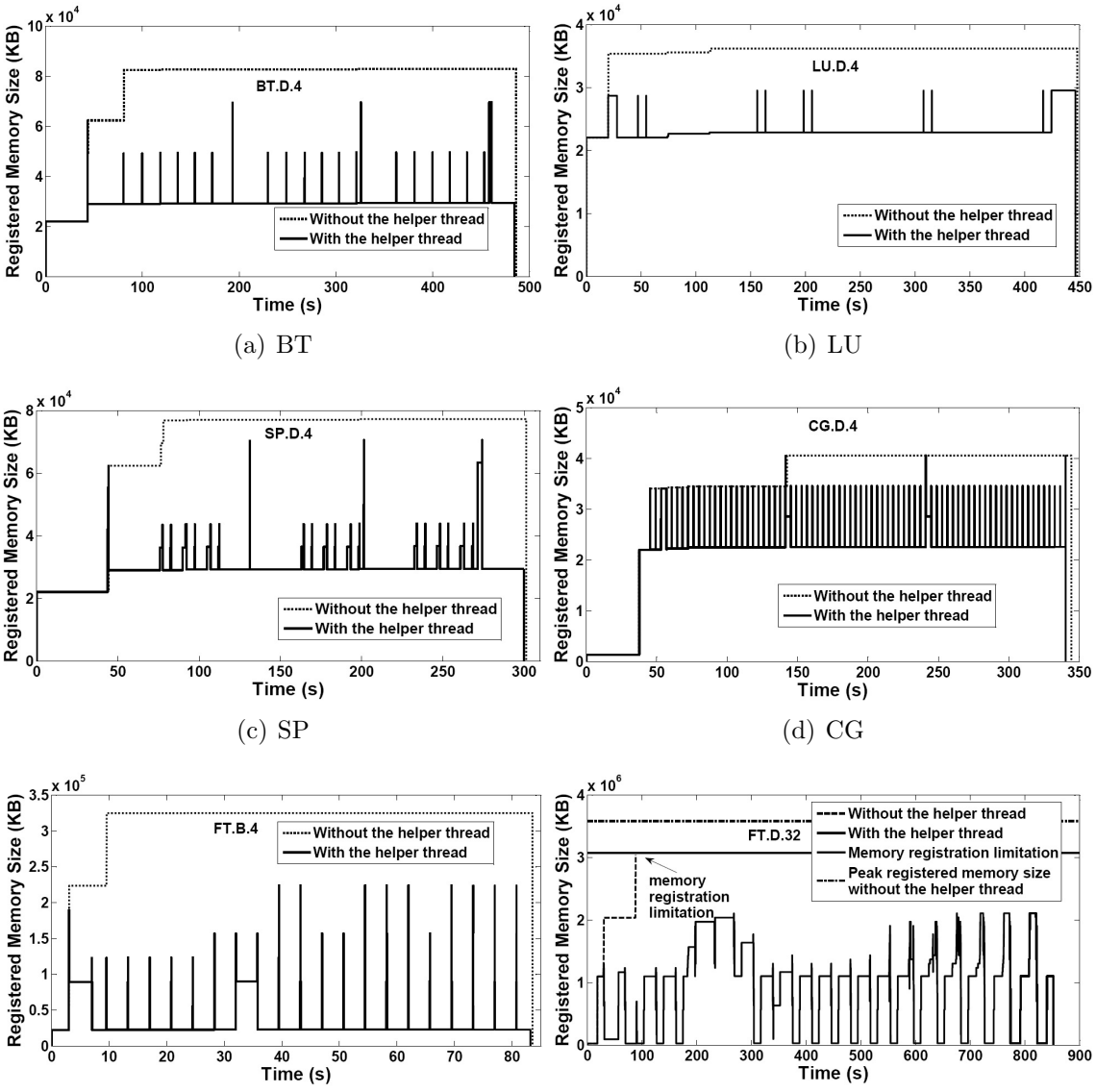
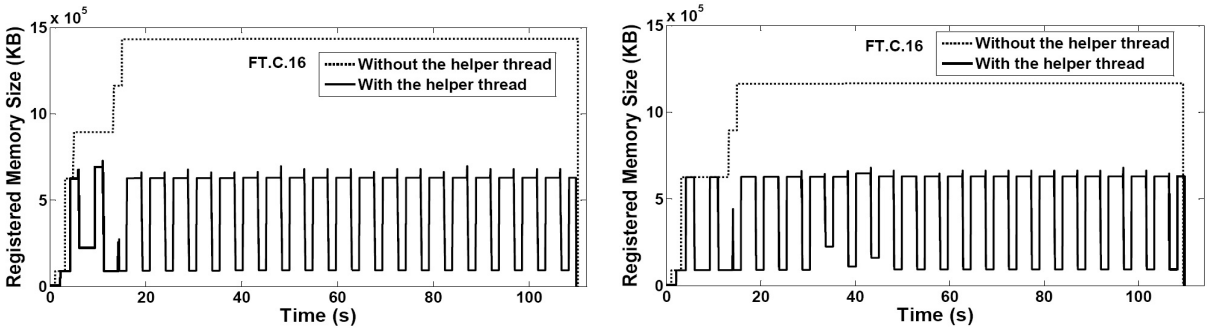


Figure 5.11: Memory registration sizes of the NAS parallel benchmarks



(a) Test 1 (b) Test 2

Figure 5.12: Task aggregation tests

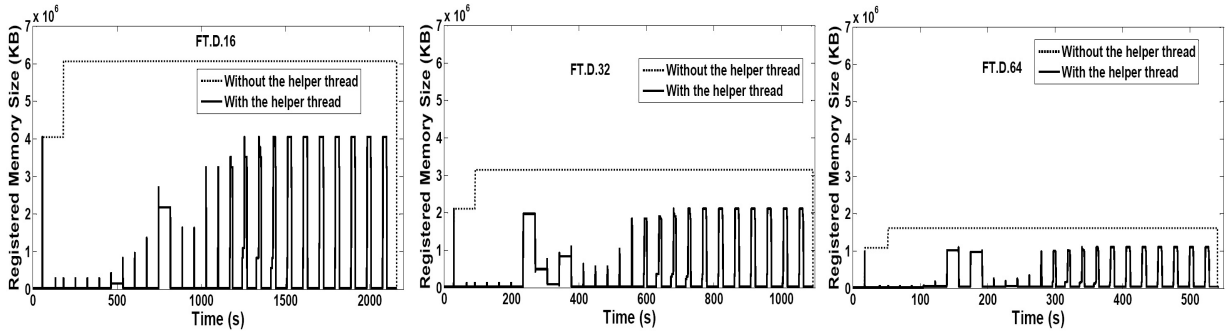
5.6.3 Task Aggregation Tests

This section examines the impact of dynamic memory registration via helper threads when aggregating multiple MPI tasks per node to save power or to improve performance [120]. We conduct experiments with four tasks per node that we randomly group before assigning them to nodes. Figure 5.12 depicts the performance of FT, problem size C with a 2×8 2D processor layout. We compare the performance of our helper thread scheme to Leave Pinned. We report the registered memory size per node from two tests with different random task groupings.

The helper thread approach reduces registered memory size significantly (49.39% and 41.70%), while also improving performance slightly (Table 5.3). We also find that the registered memory size between the two tests varies substantially with Leave Pinned, while it is consistent with helper threads. Further analysis reveals that different task groupings change the characteristics of internode communication. Registered memory size varies substantially with Leave Pinned when the volume and frequency of internode communication changes due to poor task placement. By managing memory through helper threads, we can reduce registered memory independently of how tasks are grouped.

	Test 1	Test 2
Helper thread	102.62	102.64
Leave Pinned	104.05	103.36

Table 5.3: Execution time (s) for FT.C.16 aggregation tests

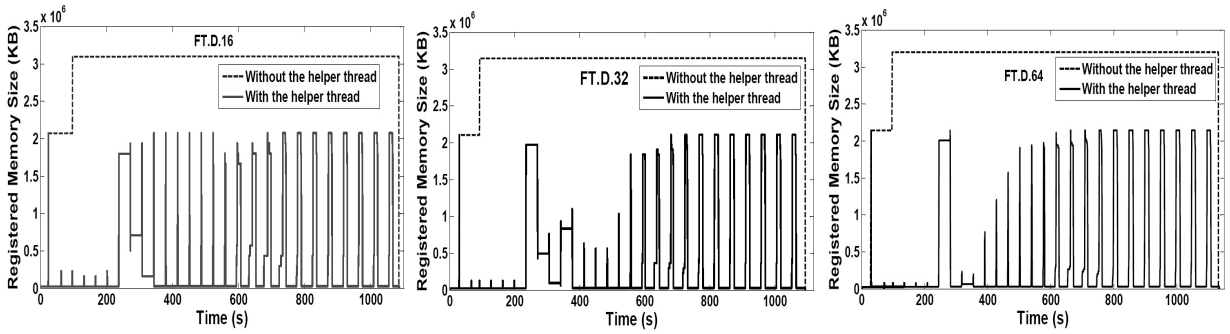


(a) 16 processes (b) 32 processes (c) 64 processes

Figure 5.13: Strong scaling tests

5.6.4 Scaling Tests

We extend our analysis to larger system scales to investigate how our memory registration scheme behaves as the node count changes. Figure 5.13 displays the results of FT with a 1D layout and one task per node under strong scaling (maintaining the same total problem size for all node counts). Although the registered memory size decreases as we increase the process count, as expected, our memory registration scheme achieves proportional memory savings in all cases. We save 33.19% for 16 nodes, 34.08% for 32 nodes and 31.70% for 64 nodes. Figure 5.14 displays results with weak scaling, which adjusts the input problem size to keep the workload per task approximately constant. Unlike the strong scaling tests, the weak scaling tests have an almost constant registered memory size as we increase the process count. We save 33.05%, 34.08%, and 33.09% for 16 nodes, 32 nodes and 64 nodes. Because the workload per task is approximately stable, each task has approximately the same amount of data to communicate across all scales. In all cases, our scheme reduces registered memory size at comparable performance (Table 5.4).



(a) 16 processes (b) 32 processes (c) 64 processes

Figure 5.14: Weak scaling tests

	Strong Scaling (node number)		
	16	32	64
Helper thread	2082.73	1055.95	519.57
Leave Pinned	2085.86	1056.00	518.53

	Weak Scaling (node number)		
	16	32	64
Helper thread	1041.58	1055.95	1093.16
Leave pinned	1039.46	1056.00	1092.23

Table 5.4: Execution time (s) of FT scaling tests.

5.7 Chapter Summary

This chapter presented a novel scalable memory registration/deregistration scheme to reduce registered memory sizes for RDMA-based communication on HPC clusters based on multi-core processors. We leverage a helper thread to register and to deregister user buffers according to a registration time prediction. Our approach reduces the amount of registered memory while avoiding memory registration/deregistraton overhead, a result that prior methods cannot achieve. We design a context-aware predictor to provide highly accurate time prediction. By combining information from the current call site and adjacent communication call sites, the predictor captures repeated communication patterns from complex program structures. We presented a time-oriented helper thread design. Our design delays memory registration and avoids the accumulation of registered memory. We discussed design issues, such as synchronization, how to minimize the latency introduced to the communication critical path, and how to reduce the resource requirements of helper threads. These design issues are

important for applying helper threads to offload non-data-related communication overheads in MPI [13]. We investigate how to distribute helper threads between limited idle cores to minimize registered memory size. We apply our framework to the NAS parallel benchmarks and reduced registered memory by 23.62% on average and up to 49.39%. Our mechanisms outperform existing memory efficient solutions and achieve similar performance while using much less memory than existing high performance solutions.

This page intentionally left blank.

Chapter 6

Future Work

In the previous chapters, we covered a number of methods and technologies to enable scalable and energy efficient execution of parallel applications on multicore systems. However, there is still a wide variety of future projects. We discuss them in this chapter.

6.1 Dynamic Concurrency Throttling on NUMA Multicore Systems

Microprocessor design is moving toward putting the memory controller on the same die as the CPU [98, 160]. On-chip memory controllers introduce Non-Uniform Memory Access (NUMA) to SMPs, because cores experience lower latency when accessing memory attached to the memory controller on their chip than when accessing memory attached to another chip. In addition, with the increasing number of cores per chip, multiple memory controllers that can be located in different parts of the chip will be introduced to avoid bottlenecks with on-chip memory controllers. The presence of multiple memory controllers will cause asymmetric access latencies for cores within the socket.

Our current DCT policy does not consider the memory design in NUMA multicores and

has the potential of the performance loss when applying DCT at a fine granularity. In particular, the default memory policy on many OSs is first-touch. Under this policy, a page of data is allocated in the memory of the first core to touch a word of the page [129]. So if we change the number of threads across OpenMP phases, either during sampling iterations or non-sampling iterations, we have the risk of accessing data in non-local memory and suffering performance loss.

To solve the above problem, we must expose more architecture information to our runtime system. When the concurrency configuration must be changed across OpenMP phases, the data location should be considered, so that we can always have a lower latency of data access.

6.2 Communication Performance Prediction for Task Aggregation

As we aggregate multiple MPI tasks within the same node, communication operations can happen concurrently and interfere with each other. As we have discussed in section 4.3, our current method is based on static analysis to learn the proximity of MPI communication calls and then to predict a performance upper bound. We do not precisely know the state of the system when a communication event happens. Here, the state of the system at any time is a combination of the best achievable bandwidths and latencies and traffic in the network. We can derive the state of the system by replaying the communication and computation event traces (including time information).

Motivated by prior work [14, 174], we can develop a simulator to improve the prediction accuracy of communication performance. The simulator replays the communication events captured in MPI program traces. To estimate the communication performance for untested aggregation patterns, we replace the computation times in the event traces collected from

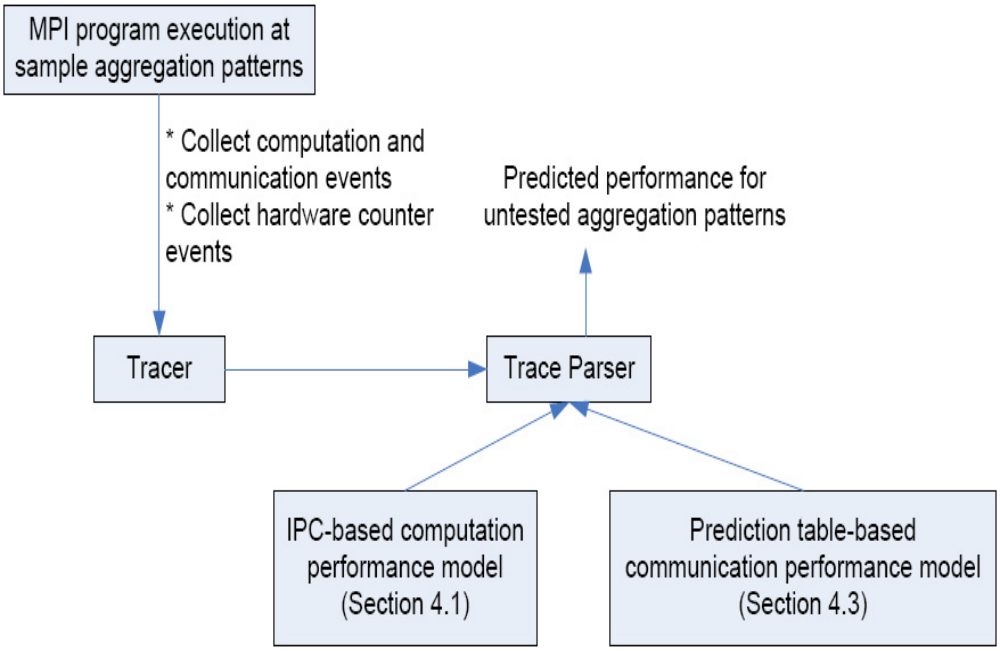


Figure 6.1: Performance prediction for untested aggregation patterns

sample aggregation patterns with the predicted computation time, so that we can estimate the state of the system when the communication event happens. Based on the more accurate description of the state of the system from the simulator, our input parameters (section 4.3) for the prediction table can better capture the performance of concurrent communication. The whole prediction process for communication performance is depicted in Figure 6.1.

6.3 Medium-Grain Concurrency Throttling

In Section 3.1, we researched the performance implications of concurrency throttling. One important conclusion we reached is that the previous DCT method (i.e., fine-grain DCT) can suffer from performance loss due to cache misses caused by the change of concurrency configuration. To solve the problem, we proposed a one phase approach, a coarse-grain DCT. Although our one phase approach avoids performance loss, it might lose the optimal concurrency configuration, from which we can get better performance. The optimal con-

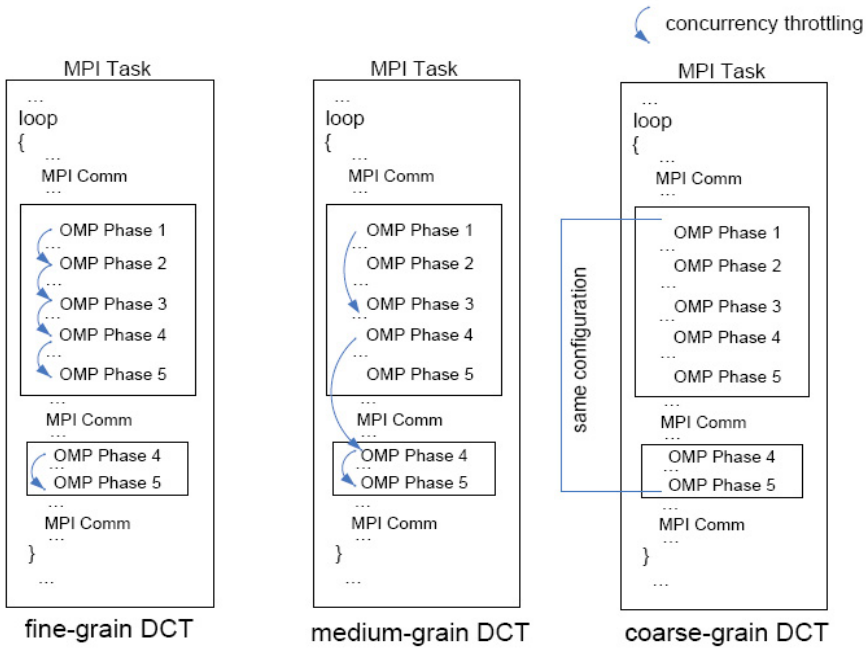


Figure 6.2: DCTs with different granularity

currency configuration may require *medium-grain concurrency throttling*, where OpenMP phases are appropriately clustered and use the same concurrency configuration within the cluster to avoid cache misses. Different clusters of OpenMP phases use different concurrency configurations to achieve optimal performance. We depict different granularities for DCT in Figure 6.2.

To identify the appropriate concurrency cluster, we must extend our DCT performance model. The current model only considers the performance of single OpenMP phases and ignores the impact of previous OpenMP phases. The new model should capture those factors that determine the interactions of neighbor OpenMP phases. The model should be able to avoid any performance loss caused by cache misses across the OpenMP phase clusters. Alternatively, it should predict the performance benefit that is sufficient to offset the potential performance loss of cache misses, so that using DCT still improves performance.

6.4 Power-Aware Task Aggregation for Hybrid Programming Models

In Chapter 3, we discussed power-aware hybrid MPI/OpenMP programming models with the assumption that each node has only one MPI task. In Chapter 4, we discussed power-aware MPI task aggregation with emphasis on single MPI programming model. Based on the research from these two chapters, we further propose power-aware task aggregation for the hybrid programming model. In other words, we allow the placement of multiple MPI tasks within the same node, with each MPI task potentially using OpenMP directives to parallelize computation. Chapters 3 and 4 in fact are the subset of this new research, because they assume either the number of tasks per node or the number of computing threads per task is one.

Our new research will extend the space available for multicore resource management by adding a new dimension, i.e., determining the appropriate ratio of task and computing threads. The new dimension provides a new opportunity to study power-aware computing. It also poses new challenges for performance modeling. We must capture those factors that impact both multithreading efficiency and task scaling. The previous communication performance model for task aggregation might be reusable, because multithreading computation can have little impact on communication performance.

6.5 Effects of Task Aggregation on Networking

Chapter 4 discussed power-aware task aggregation. Servers and their processors are the targets to improve energy efficiency, because they are today's primary power consumers. We ignore the effects of task aggregation on networking. However, the impact of network power in a target system is becoming a significant fraction of the total system power. The intercon-

nection network in HPC systems or data centers can contribute to around 30% [122] and up to 50% [2] of the total system power. The emergence of faster storage technologies, like flash and phase-change memory [34, 79], drives the increase of cluster network bandwidth, which in turn requires significantly more switching chips with faster and more power-consuming links. Therefore, it is necessary to consider aggregation effects on networking power consumption when we deploy our task aggregation framework in future HPC systems.

There are two interesting topics in this research field. The first is power-aware dynamic networking topology changes. We can selectively disable links to change the network topology to adapt to task aggregation requirements to save energy. This adaptation would require the changes to routing algorithms. We then must evaluate how effectively we can change the algorithm to avoid performance degradations. The second topic considers the problem in the opposite way. Instead of changing the topology, we could distribute tasks according to network topology. By distributing tasks wisely, we can better leverage bandwidth and probably disable more links in order to save energy.

6.6 Heterogeneous Power-Aware Hybrid MPI/OpenMP

OpenMP, a platform-independent programming model, has been established for years for programming shared-memory parallel computers. With the emerging of heterogeneous computing environment, OpenMP has been extended to other hardware, such as GPUs [116, 117].

This extension of OpenMP provides a new dimension of energy saving for the hybrid MPI/OpenMP programming model. Besides leveraging concurrency throttling to save energy, we can choose the appropriate hardware on which to perform computation (i.e., OpenMP phases) to save energy. Although GPUs requires significant power [43], they can greatly improve performance of some computation. If the performance benefits offset the extra power consumption, offloading work to GPUs can potentially lead to energy saving, since energy is

the product of power and execution time. To determine when offloading work is beneficial, we must extend our current performance and power models to include GPU architectures. With the extension of OpenMP, we can also make this offloading transparent by using pre-compiled GPU computation kernels.

This page intentionally left blank.

Chapter 7

Conclusions

The popularity of multicore architectures demands reconsideration of high performance application and system designs. Scalable and power-aware execution of parallel applications are two major challenges on multicores. This dissertation presents a series of methods and techniques to solve the problems of scalability and energy efficiency for high-end computing systems. We start with research into a power-aware hybrid MPI/OpenMP programming model, based on the observation that the hybrid MPI/OpenMP model is gaining popularity. Then, we study power-aware MPI task aggregation, with the goal to improve energy efficiency further. Since task aggregation can impose scalability concerns on shared resources, we need a scalable system design. Therefore, in the third step we pay close attention to scalable memory registration for high performance networking using helper threads. The major conclusions and contributions for the three research parts are summarized in the following.

Our power-aware hybrid MPI/OpenMP programming model solves the problem of performance optimization and energy efficiency in a two dimensional plane (i.e., DCT+DVFS). We found that the performance loss of applying DCT at a fine granularity and propose a novel approach to avoid performance loss. To develop a power-aware parallel programming model, we found that the effects of power-saving techniques on local nodes should be

considered, as well as the effects on other nodes (i.e., global effects). We introduce task coordination to account for these global effects. We also extend the previous IPC-based DCT performance model to predict execution time, based on the requirements of energy estimation and slack time computing. We formalize three strategies of applying DCT and DVFS: DCT first, DVFS first and the combination. We identify the combination strategy as the best strategy in terms of both performance and energy saving. We also propose a new power estimation method that can predict system power consumption for various concurrency and frequency configurations. To leverage our model, we develop a runtime system and classify hybrid applications in order to facilitate the identification of slack time and to characterize the slack available for DVFS, considering both intra-node and inter-node interaction effects. By applying our system to realistic applications at a large scale (up to 1024 cores), we achieve substantial energy savings (8.74% on average and up to 13.8%) with either performance gain (up to 7.5%) or negligible performance loss. Our scaling study demonstrates that power saving opportunities continue or increase under weak scaling but diminish under strong scaling.

We further research power-aware MPI task aggregation. To predict the impact of task aggregation, we propose a series of methods and techniques. We first predict computation performance. We base our prediction on the information collected from sample aggregation patterns. From this information we learn execution properties and predict the performance of untested aggregation patterns. We then identify a task grouping problem, on which communication performance relies. By appropriately grouping tasks, we can improve communication performance for a specific aggregation pattern. We formalize the task grouping problem and map it to a classic graph partitioning problem. Given the task grouping, we predict an upper bound of communication time for different aggregation levels. The prediction method is based on our analysis of the effects of concurrent inter-task communication. The prediction method abstracts the characteristics of concurrent communication and builds

a prediction table. We reduce the space that we must consider for the table by considering task placements with small performance differences as symmetric. We evaluate our method across different aggregation patterns on system scales of up to 1024 cores. Our methods lead to substantial energy saving through aggregation (64.87% on average and up to 70.03%) with tolerable performance loss (under 5%). We observe increasing energy saving opportunities that allow more intensive task aggregation under strong scaling.

Scalable memory registration is based on an observation that the demand for registered memory grows linearly with the number of processes on the node while the registered memory is a precious resource. We propose a novel helper thread strategy to reduce the size of registered memory for high-performance networking. The helper thread offloads memory registration and deregistration from the main application thread. The helper thread identifies the periodicity of memory registration and relies on a communication context-aware predictor to predict registration time. The predictor achieves higher accuracy with a shorter learning process, compared to the state-of-art periodicity-based predictor. Our helper thread employs a unique time-oriented design. In particular, we formalize the memory registration and deregistration conditions, and quantify the execution time of each operation. We choose the right operations to perform according to the time requirement of memory registration. To implement our helper thread, we discuss several issues to avoid increasing critical path latencies when using a communication helper thread, including synchronization between the helper thread and the main thread and overhead analysis. To dedicate idle cores to helper thread efficiently, we study two policies, aggressive and conservative. We discuss which policy performs better with specific application characteristics. We also perform a detailed study of registered memory usage of large-scale parallel applications under strong and weak scaling. We found that by managing memory through helper threads we can reduce registered memory independent of how tasks are grouped. We evaluate our method and significantly reduce the registered memory (23.62% on average and up to 49.39%) and avoid memory

registration/deregistration cost for reused communication memory. Our system enables the execution of the application inputs, that could not otherwise run to completion with the memory registration limitation.

This page intentionally left blank.

Bibliography

[1] *Infiniband Network Architecture*. Addison-Wesley Professional, 2002.

[2] D. Abts, M. Marty, P. Wells, P. Klausler, and H. Liu. Energy Proportional Datacenter Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.

[3] Adaptive Computing Company. TORQUE Resource Manager. http://https://computing.llnl.gov/linux/slurm.

[4] D. Addison, J. Beecroft, D. Hewson, M. McLaren, and F. Petrini. Quadrics QsNet II: A Network for Supercomputing Applications. In *Proceedings of the High Performance Chips Symposium (HotChips)*, 2003.

[5] L. Adhianto and B. Chapman. Performance Modeling of Communication and Computation in Hybrid MPI and OpenMP Applications. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, 2006.

[6] A. Afsahi and N. J. Dimopoulos. Efficient Communication Using Message Prediction for Cluster Multiprocessors. In *Proceedings of the International Workshop on Network-Based Parallel Computing*, 2000.

[7] A. Agbaria, Y. Ben-Asher, and I. Newman. Communication-Processor Tradeoffs in Limited Resources PRAM. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[8] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1989.

[9] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.

[10] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1995.

[11] R.B. Krste Asanovic, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View From Berkeley. Technical report ucb/eecs-2006-183, EECS Department, University of California at Berkeley, December 2006.

[12] Infiniband Trade Association. Infiniband Architecture Specification. http://www.infinibandta.org, 2000.

[13] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk. Non-data Communication Overheads in MPI: Analysis on Blue Gene/P. In *Proceedings of the European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Dublin, Ireland, 2008. Springer-Verlag.

[14] Barcelona Supercomputing Center. Dimemas: Predict Parallel Performance Using A Single CPU Machine. http://www.erdc.hpc.mil/hardSoft/Software/dimemas.

[15] L. Barroso. The Price of Performance: An Economic Case for Chip Multiprocessing. *ACM Queue*, 3:48–53, 2005.

[16] L. Barsanti and A. Sodan. Adaptive Job Scheduling via Predictive Job Resource Allocation. *Lecture Notes in Computer Science*, 4376:115–140, 2007.

[17] A. Baumker and W. Dittrich. Fully Dynamic Search Trees for An Extension of the BSP Model. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1996.

[18] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha. QsNet^{II}: Defining High-Performance Network Design. *IEEE Micro*, 25(4):34–47, January-February 2005.

[19] C. Bell and D. Bonachea. A New RDMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[20] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An Evaluation of Current High-Performance Networks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[21] M. Bernaschi and G. Iannello. Collective Communication Operations: Experimental Results vs. Theory. *Concurrency - Practice and Experience*, 5:359–386, 1998.

[22] M. Bernaschi, G. Iannello, and M. Lauria. Efficient Implementation of Reduce-Scatter in MPI. *Journal of System Architecture*, 49:89–108, 1998.

[23] A. Bhatele and L. V. Kale. Application-specific Topology-aware Mapping for Three Dimensional Topologies. In *the Workshop on Large-Scale Parallel Processing*, 2008.

[24] A. Bhatele and L. V. Kale. Benefits of Topology-aware Mapping for Mesh Topologies. *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, 18(4):549–566, 2008.

[25] A. Bhatele and L. V. Kale. An Evaluative Study on the Effect of Contention on Message Latencies in Large Supercomputers. In *the Workshop on Large-Scale Parallel Processing*, 2009.

[26] A. Bhatele and L. V. Kale. Quantifying Network Contention on Large Parallel Machines. *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, 19(4):553–572, 2009.

[27] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15:29–36, 1995.

[28] S. Borkar. Low Power Design Challenges For the Decade. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, 2001.

[29] R. Brightwell, S. Goudy, A. Rodrigues, and K. Underwood. Implications of Application Usage Characteristics for Collective Communication Offload. *International Journal of High-Performance Computing and Networking*, 4:104–116, 2006.

[30] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

[31] K. Cameron, R. Ge, and Xian he Sun. LognP and Log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Transactions on Computers*, 56, 2007.

[32] K. W. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In *Proceedings of the the ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Supercomputing)*, 2004.

[33] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of the the ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Supercomputing)*, 2000.

[34] A. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2010.

[35] M. Chadalapaka, H. Shah, U. Elzur, P. Thaler, and M. Ko. A Study of iSCSI Extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, 2003.

[36] L. Chai, Q. Gao, and D. K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[37] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *the IEEE International Conference on Cluster Computing*, 2006.

[38] L. Chai, P. Lai, H. Jin, and D. K. Panda. Designing an Efficient Kernel-Level and User-Level Hybrid Approach for MPI Intra-Node Communication on Multi-Core Systems. In *Proceedings of the International Conference on Parallel Processing(ICPP)*, 2008.

[39] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and Its First Implementation: A Performance View. *IBM Journal of Research and Development*, 51:559–572, 2007.

[40] W. H. Cheng. Approaches and Designs of Dynamic Voltage and Frequency Scaling. Master's thesis, University of California, Davis, 2008.

[41] J. Choi, C. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-Aware Task Scheduling at the System Software Level. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*, 2007.

[42] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[43] S. Collange, D. Defour, and A. Tisserand. Power Consumption of GPUs from a Software Perspective . In *the International Conference Computational Science (ICCS)*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2009.

[44] D. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: From Theory to Practice. In *Proceedings of the Workshop on Portability and Performance for Parallel Processing*, 1994.

[45] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards A Realistic Model of Parallel Computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993.

[46] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1996.

[47] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2008.

[48] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs Using Event-Based Pre-

diction. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, 2006.

[49] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[50] M. F. Curtis-Maury. *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems*. PhD thesis, Virginia Tech, 2008.

[51] D. Dalessandro and P. Wyckoff. A Performance Analysis of the Ammasso RDMA Enabled Ethernet Adapter and its iWARP API. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2005.

[52] Y. Ding, M. Kandemir, P. Raghavan, and M. Irwin. A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[53] Y. Dong, J. Chen, X. Yang, L. Deng, and X. Zhang. Energy-Oriented OpenMP Parallel Loop Scheduling. In *the International Symposium on Parallel and Distributed Processing with Applications*, 2008.

[54] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[55] X. Fan, W. Weber, and L. A. Barroso. Full-System Power Analysis and Modeling for Server Environments. In *the Workshop on Modeling Benchmarking and Simulation*, 2006.

[56] X. Fan, W. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-Sized Computer. In *Proceedings of the International Symposium on Computer architecture (ISCA)*, 2007.

[57] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes. Scalable RDMA Performance in PGAS Languages. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[58] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Schueduling - A Status Report. *Lecture Notes in Computer Science*, 3277:1–16, 2005.

[59] W. Feng and K. Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *IEEE Transactions on Computer*, 40:50–55, 2007.

[60] X. Feng, R. Ge, and K. Cameron. Power and Energy Profiling of Scientific Applications on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[61] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, 1999.

[62] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 1978.

[63] MPI Forum. MPI Documents. http://www.mpi-forum.org/docs, 2009.

[64] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Proceedings of the Sixth Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1997.

[65] V. W. Freeh and D. K. Lowenthal. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[66] V. W. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer. Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[67] F. Freitag, J. Caubet, M. Farrera, T. Cortes, and J. Labarta. Exploring the Predictability of MPI Messages. In *Proceedings of IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[68] F. Freitag, J. Corbalan, and J. Labarta. A Dynamic Predictor Detector: Application to Speedup Computation. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[69] F. Freitag, M. Farrera, T. Cortes, and J. Labarta. Predicting MPI Buffer Addresses. In *Proceeding of the International Conference on Computational Science*, 2004.

[70] A. Friedley, T. Hoefler, M. Leininger, and A. Lumsdaine. Scalable High Performance Message Passing over InfiniBand for Open MPI. In *Proceedings of the 3rd KiCC Workshop 2007*. RWTH Aachen, 2007.

[71] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[72] R. Ge and K. Cameron. Power-Aware Speedup. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[73] R. Ge, X. Feng, and K. Cameron. Modeling and Evaluating Energy-Performance Efficiency of Parallel Processing on Multicore Based Power Aware Systems. In *the International Workshop on High-Performance, Power-Aware Computing (HPPAC)*, 2009.

[74] R. Ge, X. Feng, W. Feng, and K. Cameron. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters . In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2007.

[75] R. Ge, X. Feng, S. Song, H. Chang, D. Li, and K. W. Cameron. PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21:658–671, 2010.

[76] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[77] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1999.

[78] M. W. Hall and M. Martonosi. Adaptive Parallelism in Compiler-Parallelized Code. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.

[79] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a Recipe for a Flash-based Data Intensive Supercomputer. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2010.

[80] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[81] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[82] T. Heath, W. Meira Jr., B. Diniz, R. Bianchini, and E. V. Carrera. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[83] V. E. Henson and U. M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, 41, 2000.

[84] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. A Practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 562–569, June 2005.

[85] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogfP- A Model for Small Messages in InfiniBand. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[86] T. Horvath and K. Skadron. Multi-Mode Energy Management for Multi-Tier Server Clusters. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[87] C. Hsu and W. Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC)*, 2005.

[88] W. Huang, G. Santhanaraman, H. Jin, Q. Gao, and D. K. Panda. Design and Implementation of High Performance MVAPICH2: MPI2 over Infiniband. In *Proceedings of the International Symposium on Cluster Computing and the Grid*, 2006.

[89] G. Iannello, M. Lauria, and S. Mercolino. LogP Performance Characterization of Fast Messages atop Myrinet. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 395–401, 1998.

[90] AMD Inc. AMD's Six-Core Phenom II X6 1090T and 1055T. http://www.anandtech.com/show/3674/amds-sixcore-phenom-ii-x6-1090t-1055t-reviewed, 2010.

[91] Tilera Inc. Tilera TILE-Gx processor. http://www.tilera.com/products/processors.php, 2009.

[92] Infiniband Trade Association. Infiniband Roadmap. http://www.infinibandta.org.

[93] insideHPC.com. Bill Dally on Moores Law. http://insidehpc.com/2010/05/03/bill-dally-on-moores-law, 2010.

[94] National Instruments. Labview. http://www.ni.com/labview, 2010.

[95] Intel Software Development Products. Intel MPI Benchmarks 3.2. http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[96] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2003.

[97] Y. Iwamoto, K. Suga, K. Ootsu, T. Yokota, and T. Baba. Receiving Message Prediction Method. *Parallel Computing*, 29:1509–1538, 2003.

[98] J. Casazza. First the Tick, Now the Tock: Next Generation Intel Microarchitecture. 2009.

[99] H. Jang and H. Jin. MiAMI: Multi-core Aware Processor Affinity for TCP/IP over Multiple Network Interfaces. In *Proceedings of the IEEE Symposium on High Performance Interconnects*, 2009.

[100] V. Jerome, M. Maxime, V. Jean-Marc, and M. Jean-Francois. Predictive Models for Bandwith Sharing in High Performance Clusters. In *the International Conference on Cluster Computing*, 2008.

[101] H. Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *the International Conference on Parallel Processing (ICPP)*, pages 184–191, 2005.

[102] H. Jin and R. Van der Wijngaart. Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks. In *Proceedings of of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[103] C. Jung and D. Lim. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[104] R. Kalla, B. Sinharnoy, and J. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2), 2004.

[105] N. Kappiah, V. Freeh, and D. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[106] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–308, 1970.

[107] T. Kielmann, H. E. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *Lecture Notes in Computer Science*, pages 1176–1183. Springer-Verlag, 2000.

[108] J. Kim and D. Lilja. Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs. In *Network-Based Parallel Computing Communication, Architecture, and Applications*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2006.

[109] S. C. Kim and S. Lee. Measurement and Prediction of Communication Delays in Myrinet Networks. *Journal of Parallel and Distributed Computing*, 61, 2001.

[110] P. Kongetira. A 32-Way Multithreaded SPARC(R) Processor. In *Proceedings of the High Performance Chips Symposium (HotChips)*, 2004.

[111] M. Koop, M. Luo, and D. Panda. Reducing Network Contention with Mixed Workloads on Modern Multicore Clusters. In *Proceedings of the IEEE Symposium on High Performance Interconnects*, 2009.

[112] M. Koop, P. Shamis, I. Rabinovitz, and D. K. Panda. Designing High Performance and Resilient Message Passing on Infiniband. In *Proceedings of Workshop on Communication Architecture for Clusters*, 2010.

[113] Lawrence Livermore National Lab. ASC Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks.

[114] Lawrence Livermore National Lab. SLURM: A Highly Scalable Resource Manager. http://https://computing.llnl.gov/linux/slurm.

[115] C. B Lee and A. E. Snavely. Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, 2007.

[116] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2010.

[117] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[118] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. Performance Impact of Process Mapping on Small-Scale SMP Clusters–a Case Study Using High Performance Linpack. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[119] D. Li, R. Ge, and K. Cameron. Unified Dynamic Thermal Control in High End Computing. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2010.

[120] D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and M. Schulz. Power-aware MPI Task Aggregation Prediction for High-End Computing Systems. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[121] J. Li and J. F. Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proceedings of the Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[122] J. Li, L. Zhang, C. Lefurgy, R. Treumann, and W. E. Denzel. Thrifty Interconnection Network for HPC Systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2009.

[123] T. Li and L. K. John. Run-Time Modeling and Estimation of Operating System Power Consumption. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[124] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *Proceedings of the Internaitonal Symposium on Cluster Computing and the Grid*, 2008.

[125] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 85–97, 1997.

[126] M. Martinasso and J. Mehaut. Prediction of Communication Latency Over Complex Network Behaviors on SMP Clusters. *Lecture Notes in Computer Science*, 3670, 2005.

[127] M. Martinasso and J. Mehaut. Model of Concurrent MPI Communications Over SMP Clusters. Research Report RR-5910, INRIA, 2006.

[128] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-Core SCC Processor: The Programmer's View. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Supercomputing)*, 2010.

[129] C. McCurdy and J. Vetter. Memphis: Finding and Fixing NUMA-Related Performance Problems on Multi-core Platforms. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[130] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[131] A. Merkel and F. Bellosa. Task Activity Vectors: a New Metric for Temperature-Aware Scheduling. In *Third ACM SIGOPS EuroSys Conference*, 2008.

[132] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Proceedings of the European Conference on Parallel Computing (EURO-PAR)*, 2006.

[133] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, pages 35–44, 2002.

[134] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. In *Proceedings of LACSI*, 2001.

[135] C. Andras Moritz, M. Matthew, and I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12, 1998.

[136] MPICH2. A High-Performance and Widely Portable Implementation of MPI Standard. http://www.mcs.anl.gov/research/projects/mpich2.

[137] Myricom. Myrinet Open Specification and Documentation. http://www.myri.com/open-specs, 2002.

[138] G. Narayanaswamy, P. Balaji, and W. Feng. Impact of Network Sharing in Multicore Architectures. In *the 17th International Conference on Computer Communications and Networks (ICCCN)*, 2008.

[139] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[140] Open MPI Project Team. Open MPI: Open Source High Performance Computing. http://www.open-mpi.org.

[141] OpenMP.org. The OpenMP API Specification For Parallel Programming. http://openmp.org/wp/openmp-specifications, 2008.

[142] J. M. Orduna, V. Arnau, and J. Duato. Characterization of Communications between Processes in Message-Passing Applications. In *the International Conference on Cluster Computing*, 2000.

[143] J. M. Orduna, V. Arnau, A. Ruiz, R. Valero, and J. Duato. On the Design of Communication-Aware Task Scheduling Strategies for Heterogeneous Systems. In *the International Conference on Parallel Processing (ICPP)*, 2000.

[144] J. M. Orduna, F. Silla, and J. Duato. A New Task Mapping Technique for Communication-Aware Scheduling Strategies. In *the International Conference on Cluster Computing Workshops*, 2000.

[145] J. M. Orduna, F. Silla, and J. Duato. On the Development of a Communication-Aware Task Mapping Technique. *Journal of Systems Architecture*, 50(4):207–220, 2004.

[146] L. Ou, X. He, and J. Han. An Efficient Design for Fast Memory Registration in RDMA. *Journal of Network and Comptuer Applications*, 32:642–651, 2009.

[147] C. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 1988.

[148] R. Rabenseifner and G. Wellein. Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. *International Journal of High Performance Computing Applications*, 17:49–62, 2002.

[149] L. Ramos and R. Bianchini. C-Oracle: Predictive Thermal Management for Data Centers. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[150] A. Rane and D. Stanzione. Experiences in Tuning Performance of Hybrid MPI/OpenMP Applications on Quad-core Systems. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[151] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving Time in Large-Scale Communication Traces. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2008.

[152] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, 2009.

[153] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding Energy Consumption in Large-Scale MPI Programs. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Supercomputing)*, 2007.

[154] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles. Predicting Multiprocessor Memory Access Patterns with Learning Models. In *Proceedings of the International Conference on Machine Learning*, 1997.

[155] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric Interactions in Symmetric Multicore Systems: Analysis, Enhancements and Evaluation. In *ACM/IEEE SC—08: The International Conference on High-Performance Computing, Networking, Storage, and Analysis*, 2008.

[156] G. Shipman, R. Brightwell, B. Barrett, J. M. Squyres, and G. Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In *European PVM/MPI Users' Group Meeting*, 2007.

[157] G. M. Shipman, S. Poole, P. Shamis, and I. Rabinovitz. X-SRQ - Improving Scalability and Performance of Multi-Core InfiniBand Clusters. In *Proceedings of Euro PVM/MPI*, 2008.

[158] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. InfiniBand Scalability in Open MPI. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[159] M. Silvano and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[160] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. POWER5 System microarchitecture. *IBM Journal of Research and Development*, 49:505–521, 2005.

[161] TOP 500 Supercomputers Sites. TOP 500. http://www.top500.org, 2010.

[162] L. Smith and M. Bull. Development of Mixed Mode MPI / OpenMP Applications. *Journal of Scientific Programming*, 9, 2001.

[163] O. Sonmez, H. Mohamed, and D. Epema. Communication-Aware Job Placement Policies for the KOALA Grid Scheduler. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, 2006.

[164] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[165] S. Srinivasan, R. Keetimuthu, V. Subramani, and P. Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Proceedings of the 2002 International Workshops on Parallel Processing*, 2002.

[166] Standard Performance Evaluation Corporation. SPEC MPI 2007. http://www.spec.org/mpi.

[167] C. Stunkel. Exascale: Parallelism gone wild! IPDPS Key Note, http://www.ipdps.org/ipdps2010/ipdps2010-slides/keynote, 2010.

[168] H. Subramoni, M. Koop, and D. K. Panda. Design Next Generation Clusters: Evaluation of Infiniband DDR/QDR on Intel Computing Platforms. In *Proceedings of the High Performance Interconnects*, 2009.

[169] H. Subramoni, P. Lai, and D. Panda. Designing QoS Aware MPI for InfiniBand. Technical report, Ohio State University, 2009.

[170] M. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[171] S. Sur, M. Koop, L. Chai, and D. K. Panda. Performance Analysis and Evaluation of Mellanox ConnectX Infiniband Architecture with Multi-core Platforms. In *the 15th Symposium on High Performance Interconnects*, 2007.

[172] S. Sur, A. Vishnu, H. Jin, W. Huang, and D. K. Panda. Can Memory-less Network Adapter Benefit Next-Generation Infiniband Systems? In *the 13th Symposium on High Performance Interconnects*, 2005.

[173] H. Tezuka, F. Carroll, and A. Hori. Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication. In *Proceedings of the International Parallel Processing Symposium*, 1998.

[174] M. M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proceedings of Euro-Par*, 2009.

[175] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin. MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[176] P. Torre and C. P. Kruskal. Submachine Locality in the Bulk Synchronous Setting. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, 1996.

[177] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *Communication Architecture for Clusters*, 2008.

[178] D. Tsafrir, D. G. Feitelson, and Y. Etsion. Backfiling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18:789–803, 2007.

[179] B. Tu, J. Fan, J. Zhan, and X. Zhao. Accurate Analytical Models for Message Passing on Multi-core Clusters. In *the Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009.

[180] B. Tu, M. Zou, J. Zhan, X. Zhao, and J. Fan. Multi-Core Aware Optimization for MPI Collectives. In *the IEEE International Conference on Cluster Computing*, 2008.

[181] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33:103–111, 1990.

[182] M. Velamati, A. Kumar, N. Jayam, G. Senthikumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan. Optimization of Collective Communication in Intra-Cell MPI. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC)*, 2008.

[183] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net.

[184] J. Vetter and F. Mueller. Communication Charateristics of Large-Scale Scientific Applications for Contemporary Cluster Architecttures. In *Proceedings of IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[185] M. Voss and R. Eigenmann. Reducing Parallel Overheads Through Dynamic Serialization. In *Proceedings of the 13th International Symposium on Parallel Processing and Symposium on Parallel and Distributed Processing*, 1999.

[186] A. Weissel and F. Bellosa. Process Cruise Control-Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2002.

[187] F. Wolf and B. Mohr. Autmatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49:421–439, 2003.

[188] T. S. Woodall, G. M. Shipman, G. Bosilca, and A. B. Maccabe. High Performance RDMA Protocols in HPC. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[189] P. Wyckoff, D. K. Panda, and J. Wu. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. In *Proceedings of Cluster Computing*, 2003.

[190] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic Thermal Management through Task Scheduling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[191] R. Zamani and A. Afsahi. Communication Charateristics of Message-Passing Scientific and Engineering Applications. In *Proceedings of Parallel and Distributed Computing and Systems*, 2005.

[192] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, 2004.

[193] Y. Zhang, V. Tipparaju, J. Nieplocha, and S. Hariri. Parallelization of the NAS Conjugate Gradient Benchmark Using the Global Arrays Shared Memory Programming Model. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[194] Y. Zhang and M. Voss. Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[195] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubincki, J. Philibin, and K. Li. Experience with VI Communication for Database Storage. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.