Automatic Resource Management using an Adaptive Parallelism Environment [*]

David Wangerin and Isaac D. Scherson
{dwangeri, isaac}@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

The Adaptive Parallelism Environment is introduced as a means of effectively utilizing MPP processing resources in a multi-programmed MIMD or distributed system. It achieves this by dynamically calculating the optimal number of threads for a given program at runtime. The optimality calculation considers both the number of processing elements available to handle additional workloads and the estimated computational speedup gained by using additional threads versus the communications overhead of using additional threads. The Adaptive Parallelism Environment is composed of three sections: a load balancer, which migrates threads and provides information regarding the availability of processing elements, a code analyzer, which estimates the number and composition of instructions in a potential thread as well as the number of communications needed to synchronize the potential thread, and a runtime environment, which gathers information from the load balancer and code analyzer and performs the runtime calculations to estimate the optimal number of threads.

1. Introduction

Programmers face many difficulties when designing programs for use in diverse MPP computing environments. There are differing degrees of parallelism in algorithms with associated performance tradeoffs, design complexity tradeoffs, and testing and debugging differences. If too much parallelism is used, then there will be more threads than processors, and execution times will suffer. Alternately, if too few threads are used, then the MPP machine may be underutilized. As a solution to these problems, programmers usually select a target architecture with a set number of processing elements and then use the appropriate algorithms for that architecture. The result of this targeting and selection process is the creation of programs that are specifically designed for systems with a rigid number of processors. Compounding this problem is a lack of transparent support for using differing numbers of processors in popular programming languages and a lack of general knowledge and expertise about portable parallel programming. To make matters worse, if the MPP system is using a load balancer rather than a gang scheduler, then there may be multiple programs running concurrently. Consequently, not all the processing elements may be available for use, and the system will effectively have fewer processors than are physically in the system.

A second, and equally discouraging, problem is that different MPP architectures have vastly different communication performance characteristics. Parallel programs are usually designed for use on a particular parallel machine whose characteristics are well known so that there is a good balance between communication requirements and computation requirements. For example, a program designed to run on a Cray T3E will typically use a large number of threads with a high ratio of communications to computations, but this same program would be disastrously slow on a Beowulf cluster of PCs using 10Mbit Ethernet. Conversely, a program written for a cluster of PCs would have a low ratio of communications to computations, but would greatly underutilize the potential of a supercomputer. These examples are telling and reveal the importance of communications in the performance of parallel programs.

The Adaptive Parallelism Environment is a novel approach to addressing both programming and archi-

[*]This work was supported in part by the UC MEXUS program and NASA-GSFC under grant number NAG5-9695

tectural issues simultaneously. It uses a combination of runtime program analysis in conjunction with a load-balancer to dynamically determine the optimal number of threads for a program. When a program has the option of creating a new parallel thread, it consults the Adaptive Parallelism Environment to see if the thread should be created. If the Adaptive Parallelism Environment determines that an additional thread could be processed and the overhead is less than the computational speedup, it then instructs the program to create the new thread. This decision is transparent to the programmer, and dynamically adapts to the changing conditions found in multi-programmed parallel systems.

The Adaptive Parallelism Environment consists of three major components: a load-balancing scheduler, a code analyzer, and a runtime environment. The load balancing scheduler is responsible for not only the distribution and scheduling of threads, but also for providing heuristics about the number of available processors and network status. The code analyzer provides a combination of compile-time and runtime heuristics regarding the computation and communication requirements of a thread or section of code. The runtime environment combines the heuristic information from the scheduler and code analyzer and decides the optimal cutoff point for creating new parallel threads.

The Adaptive Parallelism Environment determines the optimal number of threads based upon two underlying principles. In order for a parallel program to achieve a computational speedup, two conditions must be satisfied:

- For each parallel thread, there must be a processing element available to handle the workload in a timely fashion
- The computational speedup of using an additional thread must be greater than the communications overhead of using the thread

If both of the above conditions can be satisfied, then it can be surmised that that a parallel program is operating with less than the optimal number of threads. If one of these conditions fails, the program has the optimal number of threads, and creating additional threads will not lower the execution time.

2. Related Work

A related area of research is automatic management of parallel resources. Two projects have been working on creating distributed Java Virtual Machines (JVM) that automatically manage parallel resources. IBM's cJVM[1] focuses on creating a Single System Image (SSI) for a JVM and allowing all threads to execute in a distributed environment while synchronizing threads and resources through the SSI. IBM's target platforms use very fast communications (microsecond latency messaging) and do not contain any fault tolerance capabilities. The University of Hong Kong has been working on a similar project but without the assumption of fast communications. Their project is titled JESSICA: Java Enabled Single System Image Computing Architecture[11]. The software is middleware for a JVM that supplies a SSI and transparent thread management. JESSICA does not provide any services for fault tolerance or dynamic addition and removal of processing nodes.

Nicholas Carriero and others at Yale University have been working on a system called Piranha that uses a form of Adaptive Parallelism[4][3]. Piranha maintains a list of available processing resources and creates parallel treads to utilize the resources. The list of processing elements is updated at regular intervals, and thus Piranha can dynamically adapt to changing environments. Programs must be specifically written to use Piranha through a set of library calls, and programmers are responsible for finding and exploiting parallelism in a program.

Perhaps the work most similar to the Adaptive Parallelism Environment is Cilk[15]. Cilk is an extension to C that allows programmers to easily create parallel programs with a minimal amount of effort and knowledge of parallel programming. Cilk automatically creates parallel threads at runtime and only creates new threads if it believes that the computational speedup will be greater than the overhead of spawning a new thread and transferring it to a new processing element. The decision algorithm for creating new threads uses an estimate of the number of instructions in the proposed thread and weighs this against a cutoff point. Cilk uses a work stealing load balancer to facilitate thread migration. Cilk has been used in several successful programs, such as StarTech, *Socrates, and Cilkchess.

3. Load Balancer

Load Balancers attempt to improve an applications performance by redistributing its workload among processing elements. However, the load balancing activity comes at the expense of useful computation, incurs communication overhead and requires memory space to maintain load balancing information. To justify the use of load balancing strategies, the accuracy of each balancing decision must be weighed against the amount of added processing and communication incurred by the balancing process.

The Rate-of-Change Load Balancer (RoC-LB) is the most efficient load balancing algorithm for distributed and MIMD systems [2]. It is a fully distributed algorithm that maximizes processor utilization while maintaining good responsiveness. It works on the principle that workload units (e.g. threads, tasks, jobs, etc.) should only be moved from one processing element to another if the source PE is overloaded with workload units and the destination PE can handle more workload units. Under this scheme, the system spends the least possible time moving workload units and doing other balancing functions that do not contribute to the productivity of the system.

In order for the Rate-of-Change Load Balancer to determine when to move workload units, each processing element records the number of tasks and average execution time of the tasks it has run and uses this information to estimate its future workload. If the tasks have a high rate of change and it appears that the processor will face starvation, it will begin searching for more workload units. Conversely, if the tasks have a low rate of change and it appears that the processor will have a more than adequate workload in the future, the processor will allow some of its workload units to be migrated to other processing elements. It is important to note that the responsibility of searching for excess workload units and obtaining additional workload units falls on the underutilized processors. This method, commonly known as workload stealing [15], minimizes the work performed by processors that are heavily utilized. By having starving processors actively looking for workload units and overloaded processors shedding their workload units, the overall system attempts to reach a balanced state where all processors are utilized to the greatest possible degree.

Processing elements broadcast their workload status by sharing a simple flag with neighboring processing elements. These flags are propagated throughout the system by having neighbors periodically update and exchange their status information. The flags can indicate one of three states for each processing element: underloaded, loaded, or overloaded. Although technically the underloaded flag is not needed for the proper execution of the algorithm, it will become an important part of the Adaptive Parallelism Environment, as will be seen in section 5. While the processing element is sharing its status flag, the communication latency times to each of its neighbors can be calculated. This information becomes another important metric to the Adaptive Parallelism Environment.

Beyond the desire to use Rate-of-Change Load Balancing for its proven efficiency, it also has three characteristics that are key to the Adaptive Parallelism Environment. First, it is a fully distributed algorithm. Each processing element maintains its own load balancing information and thus the system can scale to work on any number of processing elements. Second, it is inherently fault tolerant. Processing elements can be added and removed from the overall system during runtime, and RoC-LB will dynamically adjust to the changes. Last, the algorithm maintains a list of the workload status of each processor and the communication latency times to each processor. This information will help guide the Adaptive Parallelism Environment in the creation of new threads.

4. Code Analyzer

In order for the Adaptive Parallelism Environment to estimate the runtime of a potential thread, it must know an estimate of the number of instructions in the new thread and approximately how long each instruction will take to execute. The second issue has been addressed in a graceful fashion using Performance Vectors, which will be reviewed in the following section. The first issue is still an open-ended problem, although current research may provide a possible solution[13].

4.1. Performance Vectors

Performance Vectors[10][9] are a powerful and relatively simple way of measuring the performance of a given system. Performance Vectors work by dividing instructions into classes and then profiling the execution times of instruction classes. Given a sufficiently large set of profiling data, a geometric model can be created and analyzed to yield a mean execution time for each instruction. These mean execution times can then be applied to other programs to quickly and accurately estimate their runtime.

Performance Vectors are useful because they are flexible and they reflect the actual performance of real programs. The instruction classes are not fixed at a specific level of granularity; rather, the user can define what composes each instruction class. Thus, instruction classes can be very course-grain (e.g. arithmetic instructions, memory instructions, and branch instructions), or they can be fine-grain (e.g. integer addition, floating point multiplication, etc.). These classes of instructions are measured using low-overhead and non-intrusive techniques, such as timers and profilers. The performance information is gathered by running standard benchmarks and other programs, not synthetic workload tools. The mean execution times of instruction classes therefore reflect delivered performance and not peak performance.

Performance Vectors also include a quality measurement to gauge the accuracy and confidence of a performance vector. For example if a performance vector is comprised of mostly integer arithmetic measurements, then the analysis of a program that is floating point intensive will give a low quality measurement.

The creation and use of Performance Vectors is critical to systems in an Adaptive Parallelism Environment. Performance Vectors form the basis for estimating the execution time of potential new threads, so the Performance Vectors must be both accurate and sufficiently course-grain to keep execution time estimates simple. Luckily, Performance Vectors can be generated for a given platform at the initialization of the Adaptive Parallelism Environment and do not need to be regenerated unless the system is changed.

4.2. Code Analysis

In order for the Adaptive Parallelism Environment to properly gauge the execution time of a thread, it must know an estimate of the number and composition of instructions in a thread, as well as an estimate of the number of communications to be performed by the thread. The code analyzer is a component of a compiler that can generate equations to describe this information.

Static analysis is straightforward for code without conditionals. The instructions are separated into their respective instruction classes and then tallied. Loops represent multipliers to the contents of the loop. If loops are based upon a simple metric, such as data set size, then describing the instruction count and composition at runtime is a simple linear equation. To assess the effects of conditionals, the conditional code can be viewed as irregularity in the program. To address program irregularity, it becomes is necessary to develop a model of the program. A novel approach is to treat the program as a set of events. It then follows that the program may be described using a series set of distributions, which can in turn be analyzed to reveal the typical instruction count and composition. A framework for handling this irregularity is addressed in [12].

Inter-thread communications can be treated as a separate instruction class and thus analyzed in the same fashion as other instructions. Equations to describe communications may need additional information regarding the number of threads for the program, but this is a simple parameter when handled at runtime.

Static code analysis results in a set of linear equations and probability distributions to describe the resource requirements of a program. These sets of equations are passed to the Adaptive Parallelism Environment whenever a potential thread is under consideration. The Adaptive Parallelism Environment evaluates the equations by applying the appropriate parameters and then factoring in the execution times provided by the Performance Vectors.

4.3. Candidate Programming Languages and Methodologies

Three programming languages are excellent candidates for effectively programming in an Adaptive Parallelism Environment. Each candidate language emphasizes a different programming methodology and exposes a different level of parallelism to the programmer. However, all of the languages abstract the details of parallel programming, and thus can be used in the dynamically changing environment created by the Adaptive Parallelism Environment. Below is a brief overview of each language and the qualities of interest.

4.3.1. Explicit Parallelism

Traditionally, the programmer has been responsible for declaring the serial and parallel sections of code. This approach will be referred to as explicit parallelism. Explicit parallelism can be used in an Adaptive Parallel Environment with some constraints. Certain programming constraints need to be enforced with this method to ensure that the code will work under the dynamic nature of the Adaptive Parallelism Environment. First, the program must be written in a language that clearly specifies where communications should occur. If communications are made in a fashion that is difficult to detect, then code analysis may fail. Second, the programmer must ensure that all parallel code can function properly with a variable number of active threads. Since the adaptive parallelism environment is designed to run with a varying number of threads, algorithms that require a specific number of threads (often called non-malleable) will not function properly. Last, every time the program has the possibility to create a new thread, the program should call a decision subroutine from the dynamic parallelism environment. If the program creates parallel threads without consulting the adaptive parallelism environment, then it defeats the purpose of the adaptive parallelism environment.

Explicit parallelism is advantageous since it requires little runtime overhead (the parallelism has already been discovered), it compliments current parallel programming paradigms, and it does not require many changes to the compiler. However, it places some difficult constraints on the programmer to find and exploit the parallelism in a program.

aCe (the adaptive C extension)[5] is an excellent example of a programming language that uses explicit parallelism. In aCe, threads are treated as arrays of records. The details of parallel programming (e.g. inter-thread communications, synchronization, deadlock prevention, etc.) are hidden from the programmer and handled automatically by the compiler. aCe was developed by Dr. John Dorband at the NASA Goddard Space Flight Center for use on Beowulf clusters, Crays, and other MPP supercomputers. It emphasizes a data-parallel programming paradigm by essentially simulating a SIMD environment on top of any parallel architecture.

Modification of aCe for use in an Adaptive Parallelism Environment would be a small matter of adding a code analyzer, including calls to the Adaptive Parallelism Environment for creating threads, and removing programming conventions for users to fix the number of threads.

4.3.2. Automated Parallelism

Another approach to parallel programming is to have the programmer write sequential code and let the compiler do the work of attempting to discover parallel sections of code. The compiler then handles the creation and analysis of threads. This exact approach has proven to be successful at vector computing but faces some difficulties with pointer analysis and other ambiguous conditions. The underlying problem is that the compiler does not know anything about the runtime data set of the program, so it must take a conservative approach to creating new parallel threads. In other words, automated parallelism compilers will only create a parallel thread if it can prove that it is safe to do so, so much of the potential parallelism of a program is overlooked.

Automated parallelism has good potential for use in an adaptive parallelism environment, as it relieves the burden of enforcing the constraints found in explicit parallelism. In addition, since automated parallelism does not require the programmer to specify parallel sections of code, the front-end of the compiler would not have to be changed. This is significant since the existing code would not need to be modified in order to work with the Adaptive Parallelism Environment.

ZPL[14] is a prime candidate for including an Adaptive Parallelism Environment into an automated parallelism compiler. ZPL performs analysis on arrays and other data structures containing potential parallelism and automatically creates parallel threads. ZPL is in use at supercomputer centers including the Arctic Region Supercomputing Center, Los Alamos National Lab, and the Maui High-Performance Computing Center.

4.3.3. Dynamic Parallelism

Another new and promising form of parallel programming, called Dynamic Parallelism, performs online analysis of a program during runtime to search for parallel threads. One of the limits of the effectiveness of automated parallelism is that the data set is unknown at compile time, and thus no assumptions can be made about the status of a program during runtime. Dynamic parallelism removes this barrier by using runtime information about the data set to perform aggressive but safe searches for potential parallelism.

The first flavor of dynamic parallelism is Lambda Tagging[6]. Lambda tagging inserts information tags into code at compile time and then uses lambda calculus to rearrange statements at runtime. The resulting rearranged code often contains large sections of code without dependencies, and thus can be run in parallel. Since lambda tagging is based upon lambda calculus, it only works on code produced from functional languages. Initial tests of lambda tagging have shown speedups from 30% to 100% in sorting and 50% to 200% in matrix multiplication.

The second flavor of dynamic parallelism is called Dynamic Resolution [7][8][6]. Under dynamic resolution, the compiler inserts tags to detect conflicts in shared data, and these tags are examined and updated during runtime. By examining conflicts at runtime, a huge amount of transient parallelism can be extracted and exploited. Testing of dynamic resolution has shown speedups from 10% to 200% over optimized sequential code, and it has even proven comparable in performance to explicitly parallel code.

One of the largest problems with dynamic resolution is that the algorithm creates too many threads and the computational speedups are overshadowed by the communication overhead. This problem is perfectly addressed by an Adaptive Parallelism Environment. The only potential drawback is that all code analysis must be performed at runtime, so there will be relatively large overhead costs from running dynamic resolution along with the code analyzer.

5. Adaptive Parallelism Environment

The Adaptive Parallelism Environment is responsible for calculating the optimal number of threads for a given program. It performs this calculation based upon heuristic information provided by the load balancer and the code analyzer. The basic algorithm for

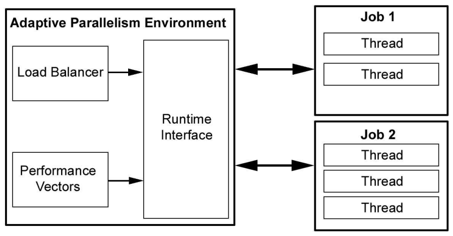

Figure 1. Overview of the Adaptive Parallelism Environment

deciding whether or not to generate a new thread is simple and operates in two phases.

- Consult the RoC-LB to determine if any processing elements are underloaded
- Compare the estimated computational speedup from generating an additional parallel thread to the estimated communication overhead

If there are available processing elements and the speedup is greater than the overhead, then a new parallel thread is created. The load balancer then automatically handles the migration of the thread.

The Adaptive Parallelism Environment is a simple service provided by a runtime environment. The Adaptive Parallelism Environment is invoked through system calls from programs. The system call contains two parameters, which are provided by the code analyzer. The first parameter is the set of equations describing the execution time of the current (single) thread. The second parameter is the set of equations describing the execution time of each of the potential new threads. The call returns a Boolean value indicating true if the program should fork the new thread or false if it would be unwise to use the additional thread.

To reach its decision, the Adaptive Runtime Environment first checks with the load balancer to determine if any processing elements are available to handle an additional workload. If no processors are available, then the call returns false. Otherwise, the Adaptive Parallelism Environment selects an available processing element, finds its communications latency, and solves the set of equations describing the execution times of the threads. If the execution time for using the single current thread is less than the execution time for using multiple threads, then the call returns false. Otherwise, the two conditions for achieving a computational speedup are satisfied and the call returns true.

To demonstrate how the Adaptive Parallelism Environment operates, consider the following examples. In Figure 2, the target architecture has four processing elements that are fully interconnected with a communication latency of 10 time units. The programming running on this architecture will take 800 time units to execute and has the potential to be broken into as many as 8 parallel threads. Assume that each thread will only need to make one communication to begin its execution and one communication at its termination.

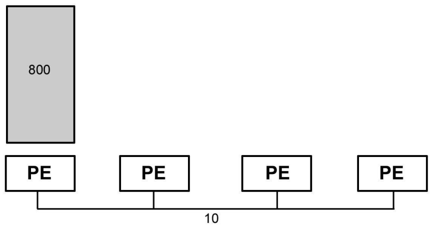

Figure 2. Initial state of a 4-processor system. Job 1 requires 800 time units of execution, and can potentially be divided into 8 threads.

Since there are four processors and the communication times are very small compared to the computation times for each potential thread, the Adaptive Parallelism Environment would fork each thread in half twice, resulting in four threads with 200 work time units each. Note that even though the remaining four threads could be forked in half, it is not advantageous to do so, since there are no additional processing elements to handle the increased number of threads. The results of the Adaptive Parallelism Environment can be seen in Figure 3

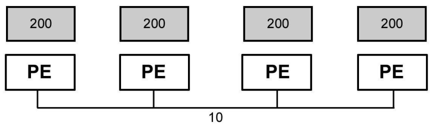

Figure 3. The Adaptive Parallelism Environment divides Job 1 into 4 threads.

Now consider that two jobs are running on the system. Each job has identical properties to the job described in the first example. Assuming that each job will be divided at the same time, the Adaptive Parallelism Environment will only divide each job once, resulting in two threads per job. Figure 4 shows the initial state of the system, and Figure 5 shows the resulting state.

In the final example, the workload is much smaller, needing only 80 time units for execution. Since each thread will need one communication to begin and one

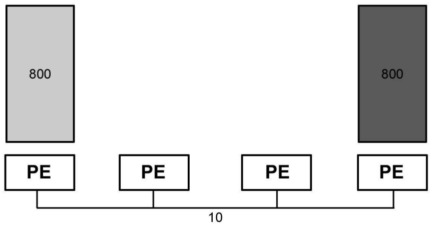

Figure 4. Two Jobs, each of which can potentially be divided into 8 threads.

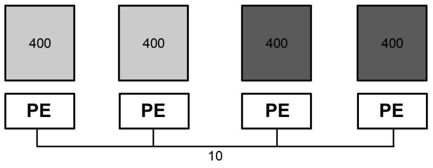

Figure 5. The Adaptive Parallelism Environment divides each Job into 2 threads.

communication to end, the communication overhead will be much greater than in the previous examples. Assume that the job can potentially be divided into 8 threads. The Adaptive Parallelism Environment would decide that the first division would be advantageous, since the execution time would drop to 60 units total (40 for the computation, 10 for the initial communication, and 10 for the termination communication). Further divisions would not be advantageous, and would in fact raise the overall execution time, since the communications overhead would overwhelm the computational speedup. For example, dividing each thread in half again would result in an execution time of 80 time units (20 for the computation, 3x10 for the initial communications, and 3x10 for the termination communications. Figures 6 and 7 show the system in its initial and resulting states.

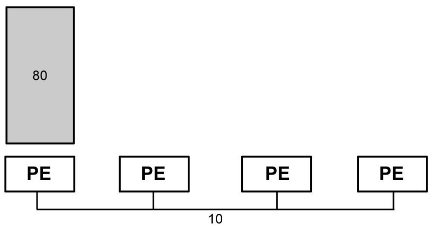

Figure 6. Communications have a larger relative overhead for smaller jobs.

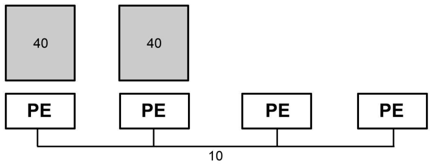

Figure 7. The Adaptive Parallelism Environment only divides the Job into 2 threads, since communications become too significant with more threads.

6. Conclusions

The Adaptive Parallelism Environment has been shown to be an effective method for dynamically determining the optimal number of parallel threads to use for each program in a multi-programmed MIMD or distributed system. It is compatible with a number of different parallel programming methodologies, and since it operates in an environment that does not interfere with other programs, it can be used in conjunction with other sequential and parallel programs that do not utilize the Adaptive Parallelism Environment.

Future work includes rigorous proofs of the optimality of thread utilization, a detailed report of the code analyzer, and simulation results of using the Adaptive Parallelism Environment versus other parallel processing paradigms.

References

[1] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, September 1999.

[2] Luis Miguel Campos. *Resource Management Techniques for Multiprogrammed Distributed Systems.* PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 1999.

[3] Nicholas Carriero, Eric Freeman, David Gelernter, and David Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, 1995.

[4] Nicholas Carriero, David Gelernter, David Kaminsky, and Jeffery Westbrook. Adaptive Parallelism with Piranha. Technical report, Yale University, 1991.

[5] John Dorband. aCe Manual.

[6] Lorenz Heulesbergen. *Dynamic Language Parallelization*. PhD thesis, University of Wisconsin-Madison, 1993.

[7] Lorenz Huelsbergen. Dynamic Parallelization of Modifications to Directed Acyclic Graphs. In *Parallel Architectures and Compilation Techniques*, 1996.

[8] Lorenz Huelsbergen and James R. Laurus. Dynamic Program Parallelization. In *ACM Conference on Lisp and Functional Programming*, pages 311–323, June 1992.

[9] Umesh Krishnaswamy. *Computer Evaluation Using Performance Vectors*. PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 1995.

[10] Umesh Krishnaswamy and Isaac D. Scherson. Micro-architecture Evaluation Using Performance Vectors. In *SIGMETRICS'96*, 1996.

[11] M.J.M. Ma, C.L. Wang, F.C.M. Lau, and Z. Xu. JESSICA: Java-Enabled Single-System-Image Computing Architecture. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 2781–2787, 1999.

[12] Shean T. McMahon and Isaac D. Scherson. A Statistical Mechanical Approach to a Framework for Modeling Irregular Programs on Distributed or Cluster Computers. In *35th Annual Simulation Symposium*, San Diego, April 2002. IEEE Press.

[13] Shean T. McMahon and Isaac D. Scherson. A General Case Performance Analysis Methodology for Parallel and Distributed Architectures and Applications. In *International Parallel and Distributed Processing Symposium*, 2003. Forthcoming work.

[14] Lawrence Snyder. A ZPL Programming Guide. Technical report, University of Washington, January 1999.

[15] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001.