

Effective Cross-Platform, Multilevel Parallelism via Dynamic Adaptive Execution *

Walden Ko, Mark Yankelevsky,
Dimitrios S. Nikolopoulos, and Constantine D. Polychronopoulos

Center for Supercomputing Research and Development
Coordinated Science Laboratory

University of Illinois at Urbana-Champaign
1308 West Main St., Urbana, IL, 61801
{w-ko, myankele, dsn, cdp}@csrd.uiuc.edu

Abstract

This paper presents preliminary efforts to develop compilation and execution environments that achieve performance portability of multilevel parallelization on hierarchical architectures. Using the NAS parallel benchmarks, we first illustrate the lack of portable performance on state-of-the-art scalable parallel systems despite the use of two portable programming models, MPI and OpenMP. Then we present a dynamic compilation and execution framework that provides the desired portability through the use of program slices. These slices are used to select the optimal program decomposition on each architecture. Currently, our framework uses a simple incremental algorithm, which effectively identifies single or multi-level program decompositions that maximize performance. This algorithm can be used as a rule of thumb for automatic multilevel parallelization. The effectiveness of the approach is demonstrated on the NAS benchmarks running on two architectural platforms.

1. Introduction

Systems with multiple levels of parallelism are becoming increasingly prevalent. These systems range from proprietary, scalable topologies of interconnected multiprocessor nodes, like commercial ccNUMA and simple-COMA systems, to clusters of multiprocessors built from commodity components. These hierarchical architectures provide the hardware needed to utilize *multigrain* parallelism in programs. Unfortunately, the corresponding programming tools and methodologies are not widely available. Al-

though some studies have explored the potential of multilevel parallelization via the integration of message-passing and shared-memory [1, 2, 7, 10, 13], deciding between single-level and multi-level parallelization is largely an *ad-hoc* procedure, with little evidence demonstrating the correct approach [9]. One reason is the many tradeoffs to consider when selecting a parallelization method. Although both single-level and multi-level parallelization yield performance improvement, they are useful under different circumstances. A number of factors affect the performance of these systems including the characteristics of the memory hierarchy, network latency, CPU speed, performance of system libraries, and operating system overhead. The large number of variables and uncertainty of the contributions of each factor lead to underutilization or misuse of multilevel parallelization.

Another obstacle to effective multilevel parallelization is the absence of performance portability from popular programming models. Recent proof-of-concept studies on the potential of combining MPI and OpenMP on both expensive (fat SMP nodes, proprietary interconnection networks, and custom implementations of MPI libraries) and cheap (thin SMP nodes, commodity interconnects, and open-source implementations of MPI) [3, 4, 5] hierarchical architectures produce conflicting results. Parallelization is highly dependent upon both the hardware resources and their interactions with the system software. Because multilevel systems are a broad classification, different systems are constructed from different types of components. As a result, a program parallelized for one system may not run with the same efficiency on another [3, 4, 5]. This problem is intensified with the introduction of deeper hardware hierarchies comprised of explicitly parallel microprocessors, multithreaded architectures, or chip-multiprocessors.

Despite these uncertainties, multilevel parallelism is in-

*This research is supported by NSF grant EIA-99-75019, a research grant from NSA, and a research grant from Intel Corporation.

deed capable of increasing program performance. However, a better understanding of the issues involved in optimizing for multilevel parallelism is needed to improve both scheduling and optimization methods. This paper presents a dynamic compilation and execution scheme to leverage the potential of multilevel parallelization and enable performance portability across hierarchical architectures.

The rest of this paper is organized as follows. Section 2 presents experiments that evaluate the performance portability of multilevel parallel implementations of the NAS benchmarks on two hierarchical systems, which provide motivation for this work. An important outcome of this study is that multilevel parallelism is most useful when it can overcome the granularity limits of single-level parallelism. Section 3 outlines the idea of a dynamic compilation and execution scheme. Central to this scheme are program slices created by the compiler. These slices are used to select the best multilevel decomposition on each architecture. Section 4 discusses results from a case study in which the framework is used to automatically scale multilevel implementations of the NAS benchmarks on two experimental platforms. Future work and conclusions are presented in sections 5 and 6, respectively.

2. Trends in Cross-Platform Multilevel Parallelization

To understand the issues in multilevel parallelism more clearly, experiments were performed on two-level parallel versions of the NAS benchmarks. MPI and OpenMP were selected for this study because of their popularity among programmers, to allow direct comparisons with similar studies on different architectures, and to examine the extent of performance portability of these programming models. MPI was used at the outer level of the benchmarks to exploit coarse-grain parallelism across the interconnection network, while OpenMP was applied in the inner level to utilize the fine-grain parallelism within tightly-coupled SMP nodes. Despite being parallelized for maximum scalability and being embarrassingly parallel, the benchmarks exhibited inconsistent performance on different parallel architectures.

Two hardware platforms were selected for their different performance properties. The first system was a 128-processor Origin2000 installed at NCSA. The Origin2000 is a commercially successful ccNUMA architecture with scalable communication through shared memory. The NCSA Origin2000 cluster on which we ran experiments uses MIPS R10000 processors running at 250 MHz, organized in a 64-node fat hypercube topology with two processors per node and two nodes per edge of the hypercube. The system is configured with 70 Gigabytes of uniformly distributed memory. The Origin comes with a custom implementation

of MPI, which exploits the underlying shared memory architecture for faster communication without changing standard MPI semantics. Parallel programs are still process-based (as opposed to thread-based) and each MPI node executes in a different address space. Previous studies have shown that the NAS benchmarks scale well up to 64 processors on the Origin2000. Some studies report problems when attempting to scale the benchmarks for 128 processors due to hidden interferences between the MPI runtime layer and operating system. The same problems occurred during execution of the benchmarks on our platform.

The second platform was a cluster of 4, 4-way SMPs connected with both Myrinet and Ethernet. This cluster used Pentium Pro processors clocked at 200 MHz, each with 512 Kilobytes of L2 cache per processor and a total of 2 Gigabytes of memory. The NAS benchmarks were parallelized for the cluster using the PGI OpenMP compiler and two implementations of MPI, MPICH v.1.2.2.2 over Fast Ethernet and GM MPICH v. 1.2.1..7 over Myrinet. The effects of parallelism were observed by simultaneously varying the number of MPI nodes and the number of OpenMP threads in the codes.

Hybrid parallelization of the NAS benchmarks was fairly straightforward. Each benchmark was first parallelized with MPI. OpenMP directives were then inserted in parallel regions within the computational portions of each MPI node. Although all the NAS benchmarks contained a considerable amount of intranode parallelism, the effectiveness of OpenMP parallelization was limited by the *coverage* of the parallelized code (i.e., the fraction of sequential code within an MPI node which is parallelized with OpenMP). In three benchmarks, (MG, CG, FT) OpenMP covered more than 90% of the sequential code within an MPI node. In BT, SP, and LU, the coverage was below 80%. These percentages decreased rapidly as the number of MPI nodes increased. As a result, OpenMP could not produce more than a four-fold speedup within MPI nodes. Note that this upper bound can be accurately identified through compiler analysis of the codes. For the sake of brevity, we provide the results for BT and CG that demonstrate the important trade-offs. The complete set of results—presented in [15]—is available online at <http://www.csrd.uiuc.edu/~myankele/hips02>.

The results are shown in Figure 1, Figure 2, and Figure 3. In each figure, the notation x/y along the x-axis represents the number of MPI/OpenMP threads. For example $1/4$ represents 1 MPI thread internally parallelized with 4 OpenMP threads, while $4/1$ represents 4 MPI threads each running sequential code. Because each benchmark in the suite requires either n^2 or 2^n threads at the outer level, some combinations are not possible and, therefore, do not appear on the graph. On the Origin graphs in Figure 1, the bars are arranged in increasing order of MPI threads. The total number of CPU's utilized are listed below each MPI/OpenMP com-

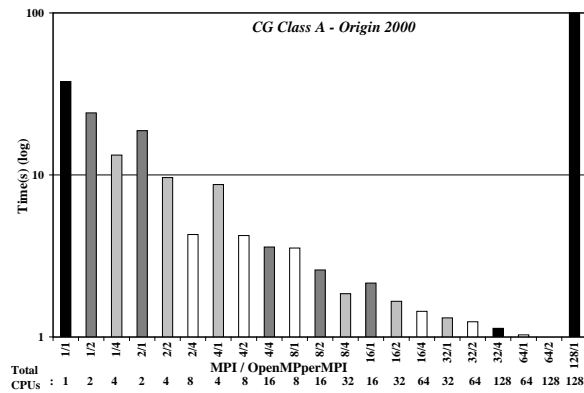
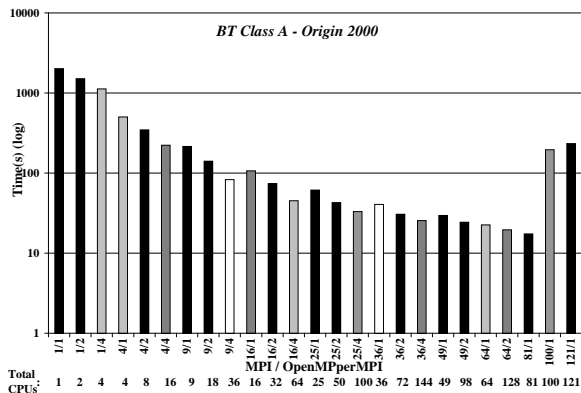


Figure 1. Performance of BT and CG on the Origin2000

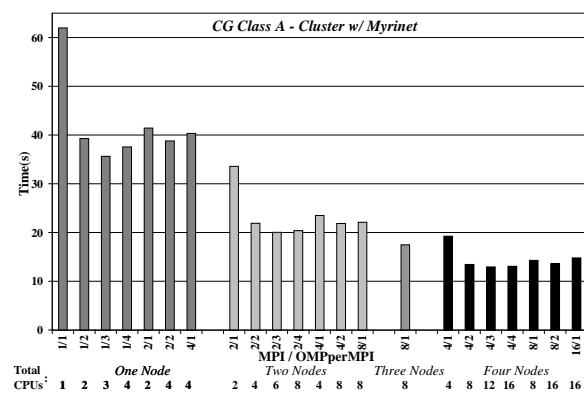
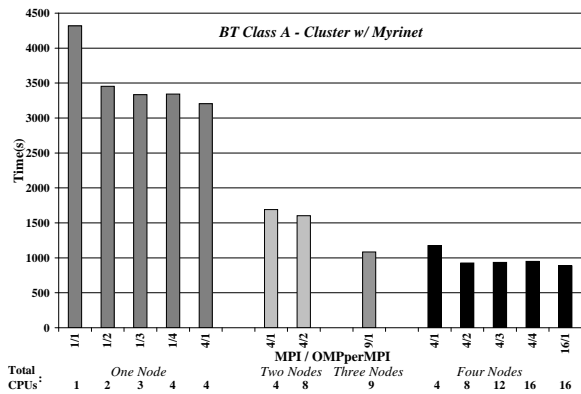


Figure 2. Performance of BT and CG on the cluster of SMPs over Myrinet

bination. On the cluster graphs, the bars are grouped by the number of nodes utilized. As in the Origin graphs, the total number of CPU's is listed at the bottom below the node sizes. Finally, note that the scale of the y-axes on the Origin graphs are logarithmic to improve legibility.

On the Origin2000, MPI scales well enough that two-level parallelization is only beneficial for more than 64 processors. This is due to the low MPI communication latencies on the Origin2000, which are very close to the raw hardware shared-memory latencies. MPI threads only have a small additional overhead from the software layer of the MPI implementation. As a result, MPI threads and OpenMP threads communicate at approximately the same rate through shared memory with an additional latency factor to account for the distance between the communicating

threads in terms of the number hops in the interconnection network. Because of the organization of the processors, OpenMP threads are concentrated in at most two nodes, with uncontended memory access latencies varying from 300 to 400 ns. The latencies of MPI threads, however, vary between 300 and 900 ns because MPI threads span across the system. Therefore, the remote-to-local memory access ratio is as much as 3:1. However, this is not a significant disadvantage. Because the coverage of MPI parallelization at the outer level is close to 100% and significantly better than the coverage of OpenMP code at the inner level, one-level parallelization with MPI yields better results until the point where MPI communication thrashes due to insufficient amount of work within each MPI thread. The substantial increase in overhead, which occurs with more than

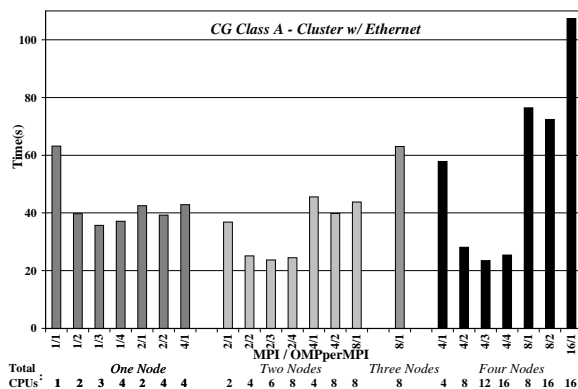
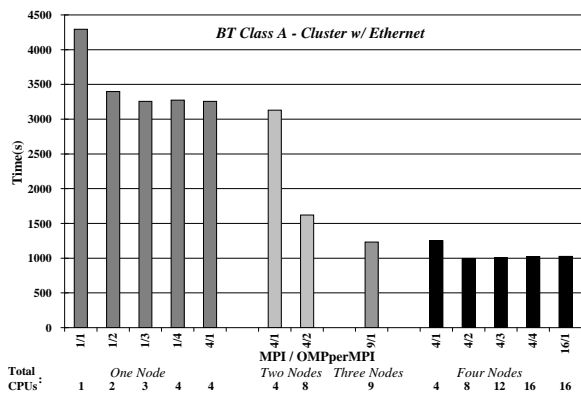


Figure 3. Performance of BT and CG on the cluster of SMPs over Fast Ethernet

64 processors, is a result of OS interference in operations that poll for the scheduling status of other MPI processes. It causes severe load imbalances between threads with different latencies. The problem occurs for all configurations of the benchmarks that use 100 or more MPI nodes.

On the cluster, only BT exhibits trends similar to those observed on the Origin2000—although for different reasons. The low coverage of OpenMP in BT limits its potential speedup to less than 2. CG, however, does benefit from OpenMP. This is somewhat surprising because the MPICH implementation on the cluster exploits shared memory for communication within the SMP. There are two explanations for this effect. First, the ratio of communication latency over Myrinet compared to the communication latency within an SMP is much higher compared to the ratio of remote to local communication on the Origin2000. This means that codes that utilize a single level of parallelism over MPI easily become unbalanced. Note that this effect is exacerbated with TCP/IP and Fast Ethernet, which slows communication by approximately one order of magnitude. Better balance is achieved by using MPI for internode communication between SMPs and OpenMP for intranode communication within SMPs. The second reason for the performance inequities of OpenMP and MPI is that the MPI layer within the SMP incurs contention and consumes excessive memory bandwidth at communication points. Because data for both computation and communication share the same SMP bus, there is a non-negligible amount of interference between the two. OpenMP alone does not suffer from this effect because there is little communication of data between the OpenMP threads within a node (the computation is cache-friendly and data sharing is minimal, yielding small cache coherence overhead) and the synchronization operations are efficiently implemented

through shared memory.

These widely divergent results demonstrate that performance portability necessitates compilation and execution tools that capture the machine and program parameters that affect performance on different architectures. Analyzing these parameters is a non-trivial procedure and can require exhaustive experimentation from the programmer. An ideal solution enables performance portability of parallel programming models like MPI and OpenMP. We propose a framework to automate this process in the compiler and runtime system that relieves the programmer from learning the intricacies of the hardware/software interface on each target system.

3. Dynamic Compilation and Adaptive Execution

Runtime inspection of software and hardware behavior has proven to be a powerful tool in several areas of high-performance compilation, enabling optimizations such as speculative parallelization [12], dynamic control of parallelism [8], and dynamic data distribution [11] to name a few. We use runtime inspection as one of the means to create *self-configured portable parallel programs (SCP³)*. We define an *SCP³* as a parallel program that meets two requirements. First, its code is portable across parallel architectures with different hardware configurations, topologies, memory models, and communication mechanisms. Second, the program maintains its performance properties across platforms, despite differences in scale, speed, and communication media. Maintaining performance properties implies that the program has the ability to self-schedule its computation in multiple, flexible ways to match differ-

ent levels of hardware parallelism and varying computation and communication latencies. Performance portability is enabled with *dynamic compilation* and *adaptive execution*. The idea is similar to previous work on dynamic compilation [14]. However, it addresses a fundamentally different problem. Although portable parallel programming models enable application portability, portable performance is often absent. As demonstrated by the discussion of the NAS benchmarks, different implementations of compiler and runtime libraries on different systems combined with differences in hardware resources produce widely varying optimization criteria for the same codes when run on different system. The same is true when programming models are combined for multilevel parallelization. Because multilevel parallelization is even less understood, optimizing for these systems is even more difficult. The goal of this framework is to maximize performance of different programming models on different systems.

The idea behind our dynamic compilation and adaptive execution scheme is to use information at runtime to determine the decomposition and schedule that maximizes the performance of a program on the given architecture. This is a two-step process that starts with the extraction of a *slice* of the program (i.e., a small code fragment which is likely to reflect the performance of the whole program when executed in isolation), followed by the generation and execution of several slice decompositions that are likely to maximize performance. The best performing decomposition is then applied to the whole program. Note that this is a runtime procedure intended to select program configurations on the fly across different hierarchical architectures. The procedure is outlined in the following sections.

3.1. Creating Program Slices

In this step—performed entirely by the compiler—the hardware probe is created. The input program is assumed to be parallelized. In these experiments MPI and OpenMP were used. Although this scheme can be used in combination with automatic parallelization, it is not absolutely necessary. The probe is a *slice* of the program. A slice is defined as a small code segment isolated from a program that is representative of the behavior of the entire program. In our implementation, a slice is obtained by extracting one iteration of the most computationally intensive loop of the full program, along with the parallelization and communication library calls. Intuitively, a slice represents the path of the program that consumes the majority of resources and execution time. This simple definition works extremely well for a large number of numerical applications. In addition to being loop-oriented, the majority of these programs iterate over the same computation. In this way, one slice captures the properties of the entire program and, furthermore, tim-

ing the slice is sufficient to select the best program configuration. In the more general scenario a slice can be defined as the most frequently occurring computation-communication sequence within the critical path of the program (e.g., a computation step followed by a barrier). In that case, the slice gives an indication of the computation to communication ratio, after cycle-counting the computation instructions and probing the hardware to identify communication volume and traffic. Extracting slices from generic programs is an area for further investigation.

Because the hardware probe selects a decomposition based on the performance of the slice, a slice that accurately represents program behavior will produce the best results. In this setting, this means that a slice should contain a sufficient number of iterations of parallel loops to negate startup effects in the cache and network. However, a large slice will increase the amount of time that is spent selecting the program version. For this implementation each slice contained two iterations of the outer loop in order to gauge cold-start effects.

3.2. Selecting a Decomposition and Executing the Full Program

The second step of the parallelization process searches for the decomposition which is likely to maximize performance. The runtime system probes the performance of different slice decompositions using an incremental algorithm. The algorithm is executed at runtime, working its way from single-level parallelization to two-level parallelization, three-level parallelization and so on.

The algorithm begins by executing a slice at a single level using as many threads as the number of processors available for execution, independent of the topology of these processors and the relative communication latencies between or within them. In a multigrain system, a single-level decomposition of the program will yield diminishing returns when the amount of work performed within each thread is dominated by the overhead of communicating and managing parallelism, or when the ratio of communication latencies at different levels of the hardware hierarchy is such that the work within threads becomes unbalanced. The former effect is a well-known scaling problem that occurs on a regular basis depending on the problem size and the machine size. The latter effect is unique to multigrain systems and occurs if the decomposition of the program produces threads that perform the same computation but communicate with widely varying latencies, resulting in load imbalance.

To investigate the performance of a two-level decomposition, the algorithm identifies the granularity/processors break-even point by working backwards from slices parallelized with as many threads as possible at the outer level

and no threads at the inner level to slices parallelized with fewer threads at the outer level and more threads at the inner level. The number of threads at the inner level is increased stepwise by an amount that divides evenly the number of threads used at the outer level. The algorithm evaluates the performance of each decomposed slice and proceeds until it detects a local minimum in the execution time of the slices. Note that for the sake of reducing runtime cost the algorithm does not guarantee the optimal solution.

The number of threads used at the inner level is naturally bounded by the number of processors used at the second level of the hardware hierarchy. For example, in a cluster of dual SMPs no more than two threads are used for inner parallelization, while in a cluster of quads the number of threads used for inner parallelization is at most four. Although simple and intuitive, this restriction is crucial for enforcing data locality, communication isolation, and other desirable properties by mapping the program parallelism to the physical parallelism of the system.

The process of extracting and executing slices with different decompositions resembles dynamic compilation, but strives for efficiency, quick adaptation, and fast response time. The slices are executed as complete program binaries. Timers are automatically inserted in the slices to measure absolute performance. After timing the slices, the best slice decomposition is applied to the program. In the current implementation, the compiler selects from precompiled program decompositions based on the potential slice decompositions. An alternative solution is to compile the best program decomposition after identifying the best slice decomposition.

4. Performance of the Dynamic Adaptive Execution Scheme

A proof-of-concept experiment was conducted to gauge the effectiveness of dynamic adaptive execution in selecting the correct two-level MPI/OpenMP parallelization. In this experiment, the slicing and probing mechanism was used to identify the best configuration of hybrid MPI/OpenMP implementations of the NAS benchmarks on our two experimental platforms, the 128-processor Origin2000 and the symmetric 4×4 cluster of SMPs, using the maximum number of processors available in each system. The purpose of this experiment was to show that dynamic compilation and execution could achieve automatic performance portability at a reasonable cost. To prove this point, the dynamic execution scheme was compared to two extreme alternatives. The *oracle* alternative selects the best program configuration for the given number of processors in zero time. The *exhaustive search* alternative runs all possible program configurations on the given number of processors to identify the best configuration for the architecture. The dynamic compilation

scheme used a slice equal to two iterations of the outermost loop of each benchmark to select the best configuration by timing the slices.

Table 1 illustrates the results for six NAS benchmarks on two platforms, the Origin2000 and the PC cluster. Columns 2 – 4 show the stand-alone execution times of candidate program decompositions probed by the dynamic execution scheme. The best performing combination is shown in bold-face. Column 5 shows the decomposition selected by the dynamic execution mechanism. Column 6 shows the total time for running the decision making process of the dynamic execution mechanism. This includes the time to select and execute the best program decomposition. For comparison, Column 7 shows the total time for selecting the best program decomposition using exhaustive search. Column 8 shows the relative overhead of the adaptive execution mechanism versus an oracle that would select automatically the best decomposition to run in zero time. This is obtained by dividing the total time spent by the adaptive execution scheme (including the time to execute the best decomposition) with the execution time of the best decomposition alone. Finally, Column 9 shows the relative gain of the dynamic execution mechanism compared to the exhaustive search approach. This is obtained by dividing the total time of the exhaustive search scheme by the total time of the adaptive execution scheme (including the time to execute the best decomposition).

The first point worth noting is the accuracy of the dynamic compilation scheme, which selects the best program configuration in all cases on the Origin2000 and the cluster of SMPs. In the NAS benchmarks, a slice of one outer program iteration captures all the computation and communication within the benchmark and, therefore, accurately reflects the behavior of the whole program. This slicing strategy works well for a large number of scientific numerical applications with strictly iterative structures, like application codes from computational fluid dynamics, molecular chemistry, weather modeling, and crash simulation.

The second point of note is the overhead of dynamic execution. For long-running programs (BT, SP, and LU) the cost of dynamic compilation is negligible, ranging from 3% to 16% on the Origin2000, and from 1% to 4% on the cluster over the execution time of the best program decomposition. This important result implies that probing a small number of candidate program configurations requires almost no additional cost, but is capable of maximizing productivity. For shorter programs (CG, MG, and FT) the cost of dynamic compilation is dominated by the cost of the slowest slice, which is comparable to the cost of executing the whole program (CG, MG, and FT have 15, 4, and 6 slices respectively). On the Origin2000, this poses a problem because the slices that use more than 100 processors execute much slower than the slices that use less than 100 processors.

SGI Origin2000 (times in seconds)								
	128×1	64×2	32×4	Adaptive Execution (AE)		Exhaustive Search (ES) Time	Search overhead (AE Total / Selected)	Gain over exhaustive (ES/AE Total)
				Selected	Total			
CG	157.37	0.91	1.13	64×2	10.66	159.41	11.71	14.95
FT	3.00	1.67	1.93	64×2	2.77	6.60	1.66	2.38
LU	417.28	37.02	53.02	64×2	44.41	507.32	1.20	11.42
MG	19.31	0.59	0.99	64×2	5.81	20.89	9.85	3.60
	121×1	64×2	25×4					
BT	233.33	19.58	33.07	64×2	22.46	285.98	1.15	12.73
SP	1481.84	27.79	54.88	64×2	28.72	1564.50	1.03	54.47
Average							4.43	16.59

Cluster of SMP PCs over Myrinet (times in seconds)								
	16×1	4×4	8×2	Adaptive Execution (AE)		Exhaustive Search (ES) Time	Search overhead (AE Total / Selected)	Gain over exhaustive (ES/AE Total)
				Selected	Total			
BT	885.60	950.33	N/A	16×1	903.98	1835.93	1.02	2.03
CG	14.80	13.08	13.63	4×4	18.70	41.51	1.43	2.22
FT	36.30	41.32	36.69	16×1	89.81	114.31	2.47	1.27
LU	360.85	1002.11	377.94	16×1	375.01	1740.9	1.04	4.64
MG	23.05	25.93	20.36	8×2	58.27	69.34	2.86	1.19
SP	533.38	770.23	N/A	16×1	539.95	1303.61	1.01	2.41
Average							1.64	2.30

Table 1. Performance of the dynamic adaptive execution mechanism on two hierarchical architectures

Nevertheless, dynamic execution is still one or two orders of magnitude faster than exhaustive search. On average, dynamic execution selects the best program decompositions 17 times faster than the exhaustive search approach on the Origin2000. On the cluster, the overhead of the dynamic execution mechanism is affordable even for the short-running benchmarks. Dynamic execution is only 2.3 times as fast as exhaustive search mainly because the mechanism probes a small number of program decompositions (two or three). If more decompositions were probed on the cluster, the exhaustive search time would increase at least linearly to the execution time of the whole program, while the dynamic execution time would only increase by an amount equal to the execution time of the slices.

5. Future Work

This work demonstrates the feasibility of dynamic adaptive execution. The next step is to develop the complete framework that generates slices and automatically selects the correct decomposition to execute. In addition to the logistics for the framework, further investigation of slice generation is planned. For example, slices for non-numeric

codes, which do not have clearly delimited outer iterations may be created via basic block profiling. Improvements in static analysis will aid in generating more accurate slices used in probing.

Further investigation of the factors that contribute to the performance of multilevel systems is also needed. This is useful not only for creating compact program slices from all types of codes, but also for improving other static analysis and optimization algorithms for multigrain parallelism. For example, one important observation obtained from the hybrid parallelization of the NAS benchmarks is the importance of the granularity of parallelism when performing a hybrid decomposition. What is needed are the means to quantify program parallelism and pair it to the available hardware resources. Other factors to consider include characteristics of the memory hierarchy and latency of the communication network. Because each model of parallelism has different strengths and weaknesses, these must be considered when partitioning a program for multilevel parallelism. A better understanding of these factors will result in better analysis tools. The desired result is a comprehensive description for static analysis to aid both slice generation and static scheduling of multilevel parallelism.

6. Conclusion

The emergence of hierarchical systems provides the opportunity to utilize multiple levels of program parallelism at the hardware level. This shows promise for using multilevel parallelism to improve performance. Unfortunately, neither formal methods for decomposing parallelism at multiple levels nor the issues involved in performing such decompositions are well understood.

This paper explores the potentials of multilevel parallelism and introduces a means for determining a good decomposition. First, the results of a hybrid parallelization of the NAS parallel benchmarks provide motivation for the problem. Multilevel parallelization is shown to be capable of increasing performance. However, the many factors to consider complicate the selection of an efficient parallelization strategy. In addition to operating system and program characteristics, the characteristics of the hardware parallelism (e.g., shared memory versus network usage and operation granularity) require consideration. Next, the benefits of a dynamic compilation and adaptive execution scheme are highlighted. Dynamic compilation provides the ability to evaluate at runtime parallelization tradeoffs that are unavailable during compilation. The components of a program slice and the means of utilizing the slice at runtime are discussed. Finally, the results of an initial study of dynamic compilation and adaptation are presented. The dynamic adaptation scheme is shown to select the best parallel decomposition in all cases—often with minimal overhead.

References

- [1] J. Berthou et al. Defining the best parallelization strategy for a diphasic compressible fluid dynamics code. In *Proc. of the Second European Workshop on OpenMP (EWOMP '00)*, Edinburgh, Scotland, October 2000.
- [2] T. Boku, S. Yoshikawa, and M. Sato. Implementation and performance evaluation of SPAM particle code with MPI-OpenMP hybrid programming. In *Proc. of the Third European Workshop on OpenMP (EWOMP '01)*, Barcelona, Spain, September 2001.
- [3] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proc. of Supercomputing 2000, High Performance Networking and Computing Conference (SC '00)*, Dallas, TX, November 2000.
- [4] F. Cappello and O. Richard. Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model. In *Proc. of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, Paris, France, October 1999.
- [5] F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Sixth International Symposium on High-Performance Computer Architecture (HPCA '00)*, Toulouse, France, January 2000.
- [6] S. M. Carroll, W. Ko, M. Yankelevsky, and C. D. Polychronopoulos. Compiler design for extensibility and modularity. In *Proc. of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC '01)*, Cumberland Falls, Kentucky, August 2001.
- [7] K. L. Cartwright and J. D. Blahovec. Adding OpenMP to an existing MPI code: Will it be beneficial? In *Proc. of Supercomputing 2000, High Performance Networking and Computing Conference (SC '00)*, Dallas, TX, November 2000.
- [8] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proc. of the 2nd SUIF Compiler Workshop (SUIF '97)*, Stanford, CA, August 1997.
- [9] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proc. of Supercomputing 2000, High Performance Networking and Computing Conference (SC '00)*, Dallas, TX, November 2000.
- [10] F. Mathey, P. Kloos, and P. Blaise. OpenMP optimization of a parallel MPI CFD code. In *Proceedings of the Second European Workshop on OpenMP (EWOMP '00)*, Edinburgh, Scotland, October 2000.
- [11] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. Is data distribution necessary in OpenMP? In *Proc. of Supercomputing 2000, High Performance Networking and Computing Conference (SC '00)*, Dallas, TX, November 2000.
- [12] J. Saltz, R. Mirchandaney, and D. Baxter. Runtime parallelization and scheduling of loops. In *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM*, pages 303–312, June 1989.
- [13] L. Smith and P. Kent. Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code. *Concurrency: Practice and Experience*, 12(12):1121–1129, 2000.
- [14] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '01)*, pages 93–102, Snowbird, UT, June 2001.
- [15] M. Yankelevsky, W. Ko, D. S. Nikolopoulos, and C. D. Polychronopoulos. Using machine descriptors to select parallelization models and strategies on hierarchical systems. In *Poster Session of SC2001: High Performance Networking and Computing (SC '01)*, Denver, CO, November 2001.