Compiler-Supported Simulation of Highly Scalable Parallel Applications

Vikram S. Adve[1] Rajive Bagrodia[2] Ewa Deelman[2] Thomas Phan[2] Rizos Sakellariou[3]

[1] *University of Illinois at Urbana-Champaign*
[2] *University of California at Los Angeles* [3] *Rice University*

Abstract

In this paper, we propose and evaluate practical, automatic techniques that exploit compiler analysis to facilitate simulation of very large message-passing systems. We use a compiler-synthesized static task graph model to identify the control-flow and the subset of the computations that determine the parallelism, communication and synchronization of the code, and to generate symbolic estimates of sequential task execution times. This information allows us to avoid executing or simulating large portions of the computational code during the simulation. We have used these techniques to integrate the MPI-Sim parallel simulator at UCLA with the Rice dHPF compiler infrastructure. The integrated system can simulate unmodified High Performance Fortran (HPF) programs compiled to the Message-Passing Interface standard (MPI) by the dHPF compiler, and we expect to simulate MPI programs as well. We evaluate the accuracy and benefits of these techniques for three standard benchmarks on a wide range of problem and system sizes. Our results show that the optimized simulator has errors of less than 17% compared with direct program measurement in all the cases we studied, and typically much smaller errors. Furthermore, it requires factors of 5 to 2000 less memory and up to a factor of 10 less time to execute than the original simulator. These dramatic savings allow us to simulate systems and problem sizes 10 to 100 times larger than is possible with the original simulator.

1 Introduction

Predicting parallel application performance is an essential step in developing large applications on highly scalable parallel architectures, in sizing the system configurations necessary for large problem sizes, or in analyzing alternative architectures for such systems. Because analytical performance prediction can be intractable for complex applications, program simulations are commonly used for such studies. It is well known that simulations of large systems tend to be slow. Two techniques have been used to alleviate the problem of long execution times: direct execution of sequential code blocks [6, 11, 14, 15] and parallel execution of the simulation models using parallel simulation algorithms [8, 13, 15]. Although these techniques have reduced simulation times, the resulting improvements are still inadequate to simulate the very large problems that are of interest to high-end users. For instance, Sweep3D is a kernel application of the ASCI benchmark suite released by the US Department of Energy. In its largest configuration, it requires computations on a grid with one billion elements. The memory requirements and execution time of such a configuration makes it impractical to simulate, even when running the simulations on high performance computers with hundreds of processors.

To overcome this computational intractability, researchers have used abstract simulations, which avoid execution of the computational code entirely [9, 10]. However, this leads to major limitations that make the approach inapplicable to many real world applications. The main problem with abstracting away all of the code is that the model is essentially independent of program *control flow*, even though the control flow may affect both the communication pattern as well as the sequential task times. Also, the preceding solution requires significant user modifications to the source program (in the form of a special input language) in order to express required information about abstracted sequential tasks and communication patterns. This makes it difficult to apply such a

tool to existing programs written with widely used standards such as Message Passing Interface (MPI) or High Performance Fortran (HPF).

In this paper, we propose and evaluate *practical, automatic techniques* to exploit compiler support for simulation of very large message-passing parallel programs. Our goal is to enable the simulation of target systems with thousands of processors, and realistic problem sizes expected on such large platforms. The key idea underlying our work is to use compiler analysis to derive abstract performance models for parts of the application. The models enable the simulator to accurately predict the performance of program fragments without executing the corresponding code. We use the compiler-synthesized *static task graph* model for this purpose [2, 3]. This is an abstract representation that clearly identifies the sequential computations (tasks), the parallel structure of the program (task scheduling, precedences, and explicit communication), and the control-flow that determines the parallel structure. In order to ensure that control flow is not ignored, we use a compiler technique known as *program slicing* [12] to isolate the portions of the computation that affect the parallel structure of the program and directly execute these computations during the simulation. This allows us to capture the exact effect of these computations on program performance, while abstracting away the remaining code. We also approximate the execution times of the abstracted sequential tasks by using symbolic expressions. The estimates we currently use are fairly simple, and are parameterized by direct measurement. Refining these values and developing analytical approximation techniques less dependent on measurement are an important component of our ongoing work.

We have combined the MPI-Sim parallel simulator [5, 6, 14] with the dHPF compiler infrastructure [1], incorporating the new techniques described above. The original MPI-Sim uses both direct execution and parallel simulation to significantly reduce the simulation time of parallel programs. dHPF, in normal usage, compiles an HPF program to MPI (or to a variety of shared memory systems), and provides extensive parallel program analysis capabilities. The integrated tool can allow us to perform simulation for MPI and HPF programs without requiring any changes to the source code. In previous work, we modified the dHPF compiler to automatically synthesize the static task graph model and symbolic task time estimates for MPI programs compiled from HPF source programs.[1] In this work, we use the static task graph to perform the simulation optimizations described above. We have extended the compiler to emit *simplified MPI code* that captures the required information from the static task graph (STG) and contains exactly the computation and communication code that must be actually executed during simulation. We have also extended MPI-Sim to exploit the compiler-synthesized static task graph model and the sequential task time estimates, and avoid executing significant portions of the computational code. The hypothesis is that this will significantly reduce the memory and time requirements of the simulation and therefore enable us to simulate much larger systems and problem sizes than were previously possible.

The applications used in this paper include Sweep3D [16], a key ASCI benchmark; SP from the NPB benchmark suite [4] and Tomcatv, a SPEC92 benchmark. The simulation models of each application were validated against measurements for a wide range of problem sizes and numbers of processors. The error in the predicted execution times, compared with direct measurement, were at most 17% in all cases we studied, and often substantially less. The validations have been done for the distributed memory IBM SP as well as the shared memory SGI Origin 2000 (note, that MPI-Sim simulates the MPI communication, not the communications via shared memory). Furthermore, the total memory usage of the simulator using the compiler synthesized model is *factors of 5 to 2000* less than the original simulator, and the simulation time is typically 5-10 times less. These dramatic savings allow us to simulate systems or problem sizes that are 1-2 orders of magnitude larger than is possible with the original simulator. For example, we were successful in simulating the execution of a configuration of Sweep3D for a

[1] In the future, we plan to synthesize this information for existing MPI codes as well. The dHPF infrastructure supports very general computation partitioning, communication analysis, and symbolic analysis capabilities that make this feasible for a wide class of MPI programs.

target system with 10,000 processors! In many cases, the simulation time was *faster than the original program*, even though the communication was simulated in detail.

The remainder of the paper proceeds as follows. Section 2 provides a brief overview of MPI-Sim and the static task graph model, and describes our goals and overall strategy. Section 3 describes the process of integrating dHPF and MPI-Sim. Section 4 describes our experimental results, and Section 5 presents our main conclusions.

2 Background and Goals

2.1 Direct Execution Simulation of MPI programs

The starting point for our work is MPI-Sim, a direct-execution parallel simulator for performance prediction of MPI programs. MPI-Sim simulates an MPI application running on a parallel system (referred to as the *target program* and *system* respectively). The machine on which the simulator is executed (the *host machine*) may be either a sequential or a parallel machine. In general, the number of processors in the host machine will be less than the number of processors in the target architecture being simulated, so the simulator must support multi-threading. The simulation kernel on each processor schedules the threads and ensures that events on host processors are executed in their correct timestamp order. A target thread is simulated as follows. The local code is simulated by directly executing it on the host processor. Communication commands are trapped by the simulator, which uses an appropriate model to predict the execution time for the corresponding communication activity on the target architecture.

The simulation kernel provides support for sequential and parallel execution of the simulator. Parallel execution is supported via a set of conservative parallel simulation protocols [6] and has been shown to be effective in reducing the execution time of the simulation models [5]. However, the use of direct execution in the simulator implies that the memory and computation requirements of the simulator are at least as large as that of the target application, which restricts the target systems and application problem sizes that can be studied even using parallel host machines. The compiler-directed optimizations discussed in the next section are primarily aimed at alleviating these restrictions.

2.2 Optimization Strategies and Challenges

Parallel program simulators used for performance evaluation execute or simulate the actual computations of the target program for two purposes: (a) to determine the execution time of the computations, and (b) to determine the impact of computational *results* on the performance of the program, due to artifacts like communication patterns, loop bounds, and control-flow. For many parallel programs, however, a sophisticated compiler can extract extensive information from the target program statically. In particular, we identify two types of relevant information often available at compile-time:

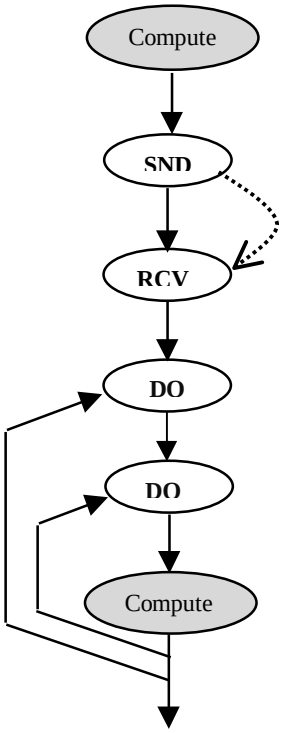
1. The parallel structure of the program, including the sequential portions of the computation (*tasks*), the mapping of tasks to threads, and the communication and synchronization patterns between threads.
2. Symbolic estimates for the execution time of isolated sequential portions of the computation.

If this information can be provided to the simulator directly, it may be possible to avoid executing substantial portions of the computational code during simulation, and therefore reduce the execution time and memory requirements of the simulation.

To illustrate this goal, consider the simple example MPI code fragment in Figure 1. The code performs a "shift" communication operation on the array D, where every processor sends its boundary values to its left neighbor, and then the code executes a simple computational loop nest. In this simple example, the communication pattern and

```
double precision A(NMAX, 1 + ceil(NMAX/MINPROC))
double precision D(NMAX, 1 + ceil(NMAX/MINPROC))
…
call mpi_comm_size(MPI_COMM_WORLD, P, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, myid, ierr)
read(*, N)
b = ceil(N / P)
…
if (myid .gt. 0) then
  <SEND D(2:N-1, myid*b+1) to processor myid-1>
endif
if (myid .lt. P) then
  <RECV D(2:N-1, (myid+1)*b+1) from processor myid+1>
endif
…
do J = max(2,myid*b+1), min(N, myid*b+b)
 do I = 2,N-1
    A(I,j) = (D(I,j) + D(I,j-1)) * 0.5
  enddo
enddo
```

(a) Original MPI Code



Task Pairs: $\{[p] \rightarrow [q]: q = p-1 \wedge p \geq 1\}$

Control-flow edge

Communication edge

Tasks: $\{[p] : 0 \leq p \leq P-1\}$

(b) Task Graph for Original MPI Code

```
integer, allocatable :: dummy_buf
…
call mpi_comm_size(MPI_COMM_WORLD, P, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, myid, ierr)
call read_and_broadcast(w_1)
read(*, N)
b = ceil(N / P)
allocate dummy_buf((N-2)*2)
…
if (myid .gt. 0) then
  <SEND dummy_buf(:)  to processor myid-1>
endif
if (myid .lt. P) then
  <RECV dummy_buf(:) from processor myid+1>
endif
…
call delay((N-2) * (min(N,myid*b+b) -
                    max(2,myid*b+1)) * w_1)
```

(c) Simplified MPI Code

Figure 1: Example to illustrate (a) a simple MPI program, (b) task-graph for MPI program, and (c) simplified MPI program for efficient simulation.

the number of iterations of the loop nest depend on the values of the block size per processor (b), the array size (N), the number of processors (P), and the local processor identifier (myid). Therefore, the computation of these values *must* be executed or simulated during the simulation. However, the communication pattern and loop iteration counts do *not* depend on the *values* stored in the arrays A and D, which are computed and used in the computational loop nest (or earlier). Therefore, if we can estimate the performance of the computational loop nest analytically, we could avoid simulating the code of this loop nest, while still simulating the communication behavior in detail. We could achieve this by generating the simplified code shown on the right in the figure, where we have replaced the loop nest with a call to a special simulator-provided delay function. We have extended MPI-Sim to provide such a function which simply forwards the simulation clock on the simulation thread by a specified amount. The compiler estimates the cost of the loop nest in the form of a simple scaling function shown as the argument to the delay call. This function describes how the computational cost varies with the retained variables (b, N, P and myid), plus a parameter w_1 representing the cost of a single loop iteration. We currently obtain the value of w_1 by direct measurement for one or a few selected problem sizes and number of processors, and use the scaling function to compute the required delay value for other problem sizes and number of processors.

As part of the POEMS project [7], we have developed an abstract program representation called *the static task graph* (STG) that captures extensive static information about a parallel program, including all the information mentioned above. The STG is designed to be computed automatically by a parallelizing compiler. It is a compact, symbolic representation of the parallel structure of a program, *independent of specific program input values or the number of processors*. Each node of the STG represents a set of possible parallel tasks, typically one per process, identified by a symbolic set of integer process identifiers. To illustrate, the STG for the example MPI program is shown in Figure 1. The compute node for the loop nest represents a set of tasks, one per process, denoted by the symbolic set of process ids $\{[p]: 0 \le p \le P-1\}$. Each node also includes markers describing the corresponding region of source code of the original program (for now, each node must represent a contiguous region of code). Each edge of the graph represents a set of parallel edges connecting pairs of parallel tasks described by a symbolic integer mapping. For example, the communication edge in the figure is labeled with a mapping indicating that each process p $(1 \le p \le P-1)$ sends to process $q = p$-1. STG nodes fall into one of three categories: control-flow, computation and communication. Each computational node includes a symbolic scaling function that captures how the number of loop iterations in the task scales as a function of arbitrary program variables. Each communication node includes additional symbolic information describing the pattern and volume of communication. Overall, the STG serves as a general, language- and architecture-independent representation of message-passing programs (and can be extended to shared-memory programs as well). In previous work, we extended the dHPF compiler to synthesize static (and dynamic) task graphs for MPI programs generated by the dHPF compiler from HPF source programs [3]. In the future, we will extract task graphs directly from existing MPI codes.

This paper develops techniques that use the static task graph model (plus additional compiler analysis) to enhance the efficiency and scalability of parallel direct-execution simulation, and evaluates the potential benefits of these techniques. In particular, our goal is to exploit the information in the static task graph to avoid simulating or executing substantial portions of the computational code of the target program. We use the task graph to identify the computational tasks, decide which computational regions can be collapsed into delay functions, and compute the scaling expressions for those delay functions. In the example of Figure 1, we decide based on the task graph that the entire loop nest could be collapsed into a single "condensed" task and replaced with a single delay function. The scaling expression for this delay function as computed by the compiler is shown in part (c) of the figure. We can also use a delay function for the compute task at the top of the graph, but we need to retain the

code that computes the values of *N*, *myid*, *b*, and *P*, as explained above. We use the compiler analysis described in Section 3.2 to expose this code.

There are three major challenges we must address in achieving the above goal, of which the first two have not been addressed in any previous system known to us:

a) We must transform the original parallel program into a simplified but legal MPI program that can be simulated by MPI-Sim. The simplified program must include only the computation and communication code that needs to be executed by the simulator. It must yield the same performance estimates as the original program for total execution time (for each individual process), total communication and computation times, as well as more detailed metrics of the communication behavior.

b) We must be able to abstract away as much of the local computation within each task as feasible , and eliminate as many data structures of the original program as possible. The major challenge here is that some computational results may affect the parallel performance of the program, as described earlier.

c) We must estimate the execution times of the abstracted computational tasks for a given program size and number of processors. Accurate performance prediction for sequential code is a challenging problem that has been widely studied in the literature. We use a fairly straightforward approach described in Section 3.3. Refining this approach is part of our ongoing work in the POEMS project.

The next section describes the techniques we use to address these challenges, and their implementation in dHPF and MPI-Sim.

3 Compiler-Supported Techniques for Efficient Large-Scale Simulation

Our overall goal is to translate a given MPI program into a simplified, complete MPI program that can then be directly input to MPI-SIM. The static task graph serves as the parallel program representation that we use to perform this translation. We first describe the basic process of using the task graph to generate the simplified MPI program, then describe the compiler analysis needed to retain portions of the computational code and data, and finally discuss the approach we use to estimate the performance of the eliminated code.

3.1 Translating the static task graph into a simplified MPI program

The STG directly identifies the local (sequential) computational tasks, control flow, and communication tasks and patterns of the parallel program. The next stage is to identify contiguous regions of computational tasks and /or control-flow in the STG that can be collapsed into a single condensed (or collapsed) task, such as the loop nest of Figure 1. Note that this is simply a transformation of the STG for simplifying further analysis and does not directly imply any changes to the parallel program itself. We refer to the task graph resulting from this transformation as the *condensed task graph.* In later analysis, we can consider only a single computational task or a single collapsed task at a time for deciding how to simplify the code (we refer to either as a single sequential task).

The criteria for collapsing tasks depend on the goals of the performance study. First, as a general rule, a collapsed region must not include any branches that exit the region, i.e., there should be only a single exit at the end of the region. Second, for the current work, a collapsed region must contain no communication tasks because we aim to simulate communication precisely. Finally, deciding whether to collapse conditional branches involves a difficult tradeoff: it is important to eliminate control-flow that references large arrays in order to achieve the savings in memory and time we desire, but it is difficult to estimate the performance of code containing such control-flow. We have found, however, that there are typically few branches that involve large arrays that do have a significant

impact on program performance. For example, one minor conditional branch in a loop nest of Sweep3D depends on intermediate values of large 3D arrays. The impact of this branch on execution time is relatively negligible , but detecting this fact in general can be difficult within the compiler because it may depend on expected problem sizes and computation times. Therefore, there are two possible approaches we can take. The more precise approach is to allow the user to specify through directives that specific branches can be eliminated and treated analytically for program simulation. A simpler but more approximate approach is to eliminate any conditional branches inside a collapsible loop nest, and rely on the statistical average execution time of each iteration to provide a good basis for estimating total execution time of the loop nest. With either approach, we can use profiling to estimate the branching probabilities of eliminated branches We have currently taken the second approach, but the first one is not difficult to implement and could provide more precise performance estimates.

While collapsing the task graph, we also compute a scaling expression for each collapsed task that describes how the number of computational operations scales as a function of program variables. We introduce time variables that represent the execution time of a sequence of statements in a single loop iteration (denoted w_i for task i). The approach we use to estimate the overall execution time of each sequential task is described in Section 3.3.

Based on the condensed task graph (and assuming for now that the compiler analysis of Section 3.2 is not needed), we generate the simplified MPI program as follows. We retain any control-flow (loops and branches) of the original MPI code that is retained in the condensed task graph, i.e., the control-flow that is not collapsed. Second, we retain the communication code of the original program, in particular only the calls to the underlying message-passing library. If a program array that is otherwise unused is referenced in any communication call, we replace that array reference with a reference to a single dummy buffer used for all the communication. We use a buffer size that is the maximum of the message sizes of all communication calls in the program and allocate the buffer statically or dynamically (potentially multiple times), depending on when the required message sizes are known. Third, we replace the code sequence for each sequential task of the task graph by a call to the MPI-Sim delay function, and pass in an argument describing the execution time of the task. We insert a sequence of calls to a runtime function, one per w_i parameter, at the start of the program to read in the value of the parameter from a file and broadcast it to all processors. Finally, we eliminate all the data variables not referenced in the simplified program.

3.2 Program slicing to retain subsets of the computational code and data

A major challenge that was mentioned earlier is that intermediate computational results can affect the program execution time in a number of ways. The solution we propose is to use *program slicing* to retain those parts of the computational code (and the associated data structures) that affect the program execution time. Given a variable referenced in some statement, program slicing finds and isolates a subset of the program computation and data that can affect the value of that variable [12]. The subset has to be conservative, limited by the precision of static program analysis, and therefore may not be minimal.

We begin by finding the variables whose values affect relevant execution time metrics, determined by the goals and desired accuracy of the performance study. For this work, these variables are exactly the variables that appear in the retained control-flow of the condensed graph, in the scaling functions of the sequential tasks, and in the calls to the communication library. Program slicing would then isolate the computations that affect those variable values.

Obtaining the memory and time savings we desire requires interprocedural program slicing, so that we completely eliminate the uses of as many large arrays as possible. General interprocedural slicing is a challenging but feasible compiler technique that is not currently available in the dHPF infrastructure. For now, we take limited interprocedural side-effects into account, in order to correctly handle calls to runtime library routines (including communication calls and runtime routines of the dHPF compiler's runtime library). In particular, we assume that

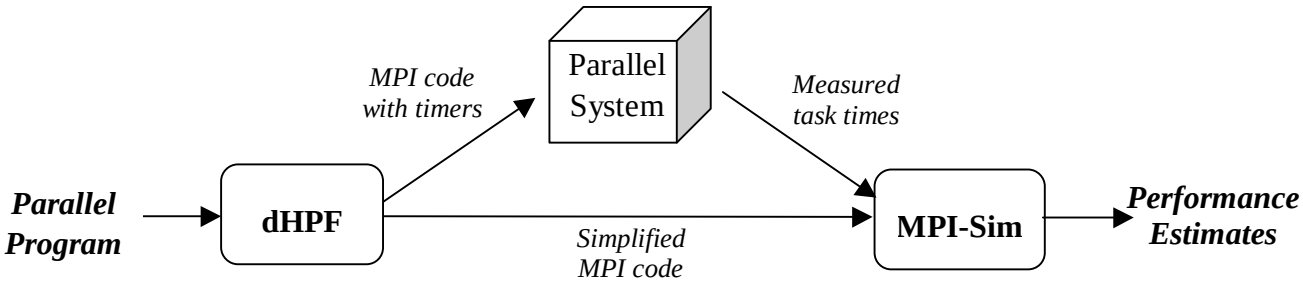


Figure 2: Compilation, parameter measurement and simulation for a parallel program.

these routines can modify any arguments passed by reference but cannot modify any global (i.e., common block) variables of the MPI program. This is necessary and sufficient to support single-procedure benchmarks. We expect to incorporate full interprocedural slicing in the near future, to support continuing work in POEMS.

3.3 Estimating task execution times

The main approximation in our approach is to estimate sequential task execution times without direct execution. Analytical prediction of sequential execution times is an extremely challenging problem, particularly with modern superscalar processors and cache hierarchies. There are a variety of possible approaches with different tradeoffs between cost, complexity, and accuracy.

The simplest approach, and the one we use in this paper, is to measure task times (specifically, the w_i) for one or a few selected problem sizes and number of processors, and then use the symbolic scaling functions derived by the compiler to estimate the delay values for other problem sizes and number of processors. Our current scaling functions are symbolic functions of the number of loop iterations, and do not incorporate any dependence of cache working sets on problem sizes. We believe extensions to the scaling function approach that capture the non-linear behavior caused by the memory hierarchy are possible.

Two alternatives to direct measurement of the task time parameters are (a) to use compiler support for estimating sequential task execution times analytically, and (b) to use separate offline simulation of sequential task execution times [7]. In both cases, the need for scaling functions remains, including the issues mentioned above, because it is important to amortize the cost estimating these parameters over many prediction experiments.

The scaling functions for the tasks can depend on intermediate computational results, in addition to program inputs. Even if this is not the case, *they may appear to do so to the compiler*. For example, in the NAS benchmark SP, the grid sizes for each processor are computed and stored in an array, which is then used in most loop bounds. The use of an array makes forward propagation of the symbolic expressions infeasible, and therefore completely obscures the relationship between the loop bounds and program input variables. We simply retain the executable symbolic scaling expressions, including references to such arrays, in the simplified code and evaluate them at execution time.

We have been able to automate fully the modeling process for a given HPF application compiled to MPI. The modified dHPF compiler automatically generates two versions of the MPI program. One is the simplified MPI code with delays calls described previously. The second is the full MPI code with timer calls inserted to perform the measurements of the w_i parameters. The output of the timer version can be directly provided as input to the delay version of the code. This complete process is illustrated in Figure 2.

4 Results

We performed a detailed experimental evaluation of the compiler-based simulation approach. We studied three issues in these experiments:

1. The accuracy of the optimized simulator that uses the compiler-generated analytical model, compared with both the original simulator and direct measurements of the target program.
2. The reduction in memory usage achieved by the optimized simulator compared with the original and the resulting improvements in the overall scalability of the simulator in terms of system sizes and problem sizes that can be simulated.
3. The performance of the optimized simulator compared with the original, in terms of both absolute simulation times and in terms of relative speedups when simulating a large number of target processors.

We begin with a description of our experimental methodology and then describe the results for each of these issues in turn.

4.1 Experimental Methodology

We used three real-world benchmarks (Tomcatv, Sweep3D and NAS SP) and one synthetic communication kernel (SAMPLE) in this study. Tomcatv is a SPEC92 floating-point benchmark, and we studied an HPF version of this benchmark compiled to MPI by the dHPF compiler. Sweep3D, a Department of Energy ASCI benchmark [16], and SP, a NAS Parallel Benchmark from the NPB2.3b2 benchmark suite [4], are MPI benchmarks written in Fortran 77. Finally, we designed the synthetic kernel benchmark, SAMPLE, to evaluate the impact of the compiler-directed optimizations on programs with varying computation granularity and message communication patterns that are commonly used in parallel applications.

For Tomcatv, the dHPF compiler automatically generates three versions of the output MPI code: (a) the normal MPI code generated by dHPF for this benchmark, where the key arrays of the HPF code are distributed across the processors in contiguous blocks in the second dimension (i.e., using the HPF distribution (*,BLOCK)); (b) the simplified MPI code with the calls to the MPI-Sim delay function, making full use of the techniques described in Section 3; and (c) the normal MPI code with timer calls inserted to measure the task time parameters, as described in Section 3.3. Since dHPF only parses and emits Fortran and MPI-Sim only supports C, we use f2C to translate each version of the generated code to C and run it on MPI-Sim. For the other two benchmarks, Sweep3D and NAS SP, we manually modified the existing MPI code to generate the simplified MPI and the MPI code with timers for each case (since the task graph synthesis for MPI codes is not implemented yet). These codes serve to show that the compiler techniques we developed can be applied to a large range of codes with good results.

For each application, we measured the task times (values of w_i) on 16 processors. These measured values were then used in experiments with the same problem size on different numbers of processors. The only exception was NAS SP, where we measured the task only for a single problem size (on 16 processors), and used the same task times for other problem sizes as well. Recall that the scaling functions we use currently do not account for cache working sets and cache performance. Changing either the problem size or the number of processors affects the

working set size per process and, therefore, the cache performance of the application. Nevertheless, the above measurement approach provided very accurate predictions from the optimized simulator, as shown in the next subsection.

All benchmarks, except SAMPLE, were evaluated for the distributed memory IBM SP (with up to 128 processors); the SAMPLE experiments were conducted on the shared memory SGI Origin 2000 (with up to 8 processors).

4.2 Validation

The original MPI-Sim was successfully validated on a number of benchmarks and architectures [5, 6]. The new techniques described in Section 3, however, introduce additional approximations in the modeling process. The key new approximation is in estimating the sequential execution times of portions of the computational code (tasks) that have been abstracted away. Our aim in this section is to evaluate the accuracy of MPI-Sim when applying these techniques.

For each application, the optimized simulator was validated against direct measurements of the application execution time and also compared with the predictions from the original simulator. We studied multiple configurations (problem size and number of processors) for each application.

We begin with Tomcatv, which is handled fully automatically through the steps of compilation, task measurements, and simulation shown in Figure 2. The size of Tomcatv used for the validation was 2048×2048. Figure 3 shows the results from 4 to 64 processors. Even though MPI-Sim with the analytical model (MPI-SIM-AM) is not as accurate as MPI-Sim with direct execution (MPI-SIM-DE), the error in the performance predicted by MPI-SIM-AM was below 16% with an average error of 11.3%.

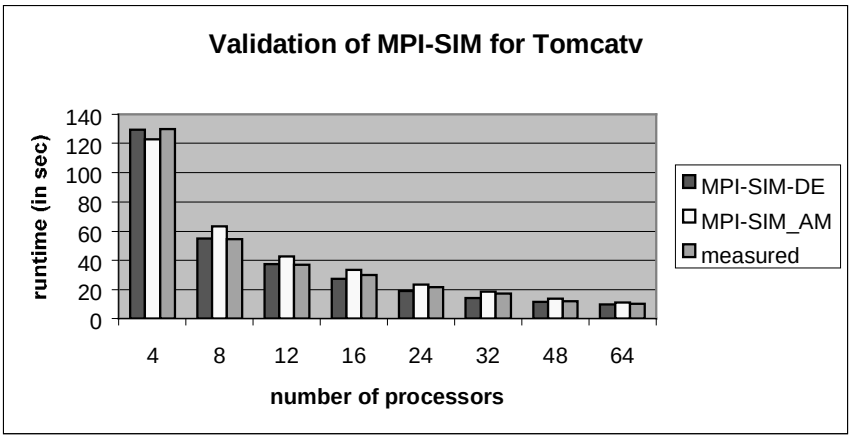


Figure 3: Validation of MPI-Sim for (2048×2048) Tomcatv (on the IBM SP).

Figure 4 shows the execution time of the model for Sweep3D with a total problem size of 150×150×150 grid cells as predicted using MPI-SIM-AM, MPI-SIM-DE, as well as the measured values, all for up to 64 processors. The predicted and measured values are again very close and differ by at most 7%.

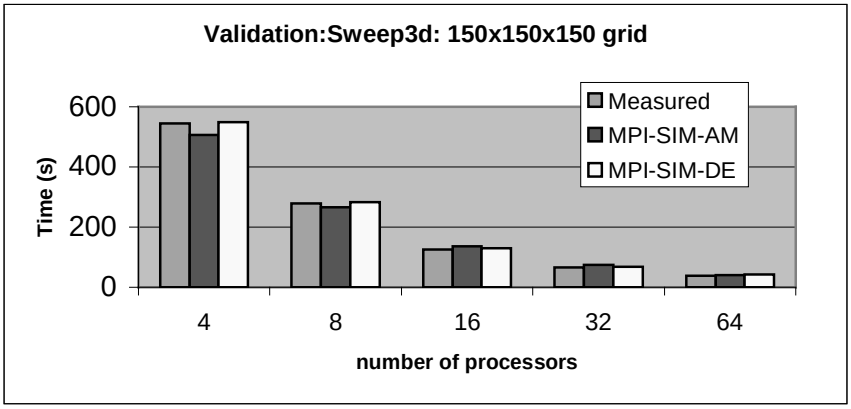


Figure 4: Validation of Sweep3D on the IBM SP, Fixed total Problem Size.

Finally, we validated MPI-SIM-AM on the NAS SP benchmark. The task times were obtained from the 16 processor run of the class A, the smallest of the three built-in sizes (A, B and C) of the benchmark, and used for experiments with all problem sizes. Figures 5 and 6 show the validation for class A and the largest size, class C. The validation for class A is good (the errors are less than 7%). The validation for class C is also good with an average error of 4%, even though the task times were obtained from class A. This result is particularly interesting because class C on average runs 16.6 times longer than class A, over and above the scaling factor for different numbers of processors. It demonstrates that the compiler-optimized simulator is capable of accurate projections across a wide range of scaling factors. Furthermore, cache effects do not appear to play a great role in this code or the other two applications we have examined. This is illustrated by the fact that the errors do not increase noticeably when the task times obtained on a small number of processors were used for a larger number of processors.

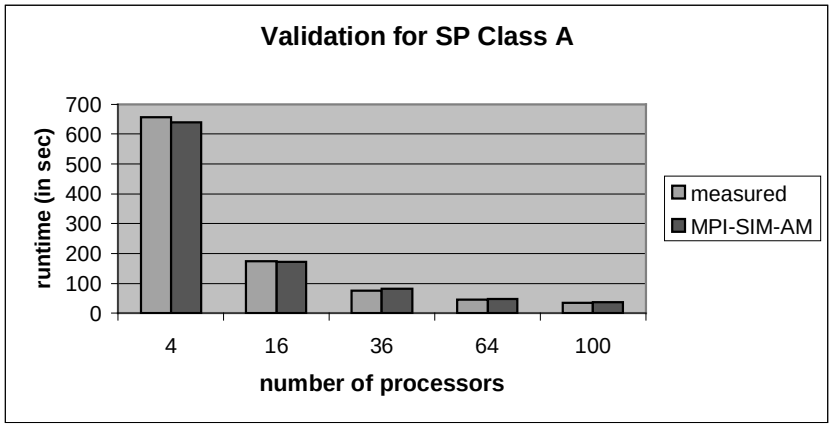


Figure 5: Validation for NAS SP, class A on the IBM SP.

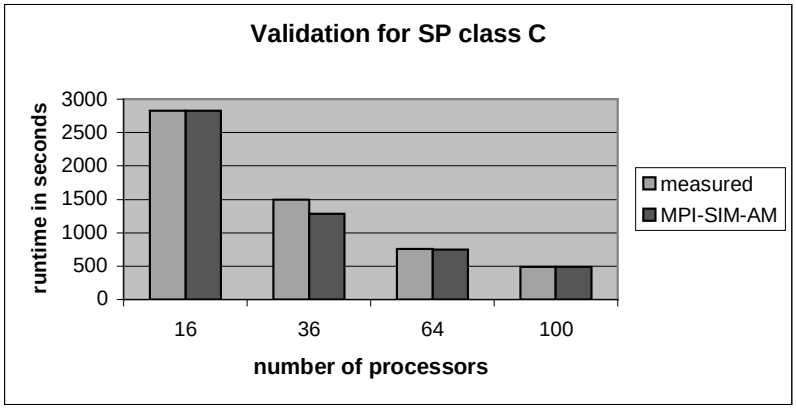


Figure 6: Validation for NAS SP, class C on the IBM SP.

Figure 7 summarizes the errors that MPI-Sim with analytical models incurred when simulating the three applications. All the errors are within 16%. The figure emphasizes that the compiler-supported approach combining analytical model and simulation is very accurate for a range of benchmarks, system sizes, and problem sizes.

It is hard to explore these errors further without detailed analysis of each application. Therefore, to better quantify what errors can be expected from the optimized simulator, we used our SAMPLE benchmark, which allows us to vary the computation to communication ratio as well as the communication patterns.

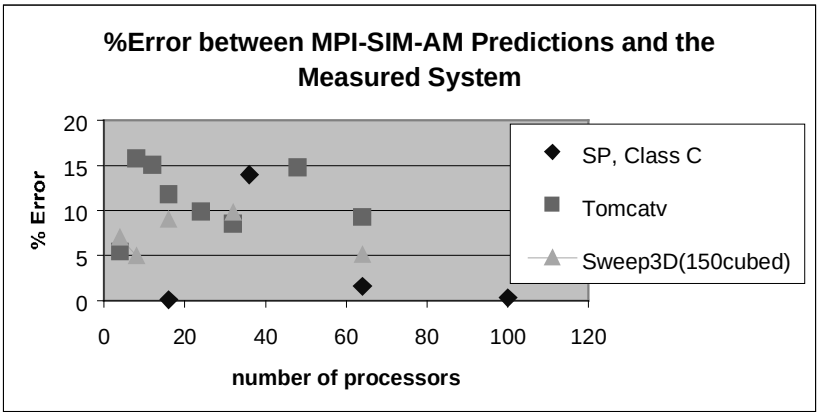


Figure 7: Percent Error Incurred by MPI-SIM-AM when Predicting Application Performance.

SAMPLE was validated on the Origin 2000. Two common communication patterns were selected: wavefront and nearest neighbor. For each pattern, the communication to computation ratio was varied from 1 to 100 to a ratio of 1 to 1. Figure 8 plots the total execution time for the program and MPI-SIM-AM prediction. In order to demonstrate better the impact of computation granularity on the validation, Figure 8 plots the percentage variation in the predicted time as compared with the measured values. As can be seen from the figure, the predictions are very accurate when the ratio of computation to communication is large, which is typical of many real-world

applications. As the amount of communication in the program increased, the simulator incurs larger errors with the predicted values differing by at most 15% from the measured values.

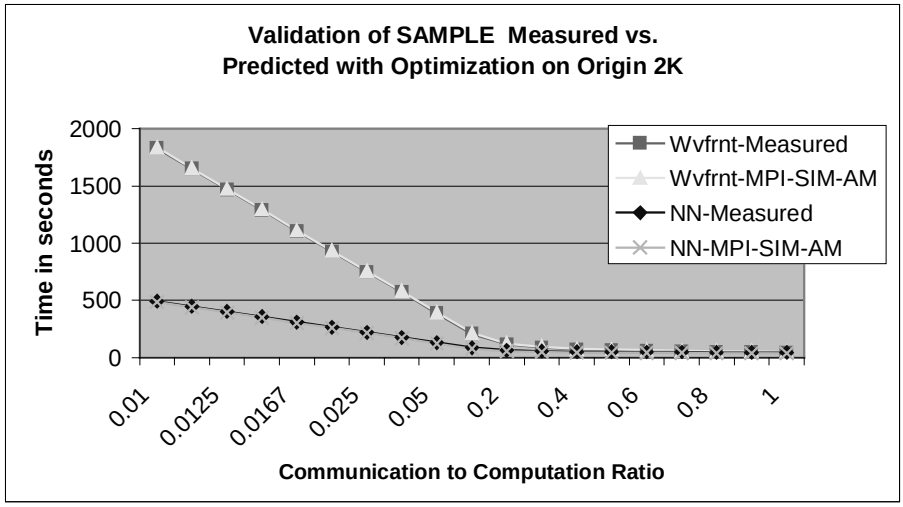


Figure 8: Validation of SAMPLE on the Origin 2000.

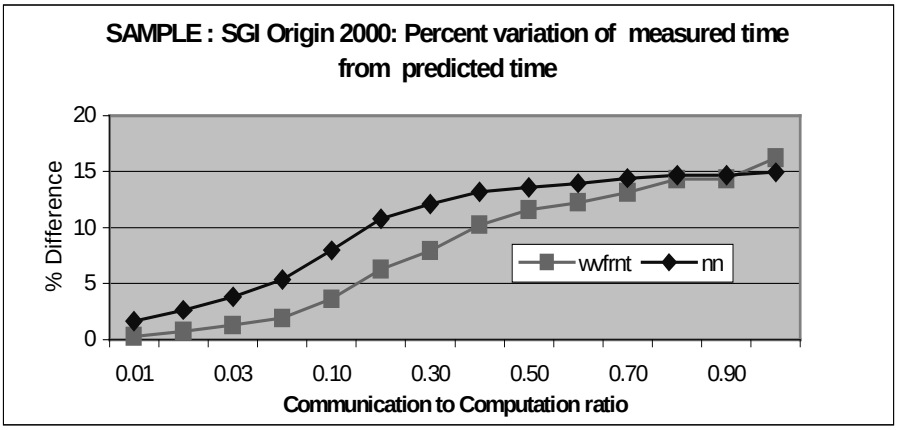


Figure 9: Effect of Communication to Computation Ratio on Predictions.

The accuracy of MPI-SIM-AM for small communication to computation ratio (below 5% error) indicates that the slightly higher errors we observed for the Tomcatv, Sweep3D and NAS SP benchmarks must come from the errors in task time estimation. Those errors might be a result of small computational granularity, where the timers used to measure the task times can inflate the time obtained for a code segment.

4.3 Expanding the simulator to larger systems and problem sizes.

The main benefit of using the compiler-generated code is that we can decrease the memory requirements of the simplified application code. Since the simulator uses at least as much memory as the application, decreasing the amount of memory for the application decreases the simulator's memory requirements, thus allowing us to simulate large problem sizes and systems.

	Number of processors	MPI-SIM-DE total memory use	MPI-SIM-AM total memory use	Memory Reduction Factor
Sweep 3D, 4×4×255	1	589KB	6KB	98
per Proc. Problem Size	4900	2884MB	30MB	96
Sweep 3D, 6×6×1000	1	33.599MB	19KB	1768
per Proc. Problem Size	6400	215GB	122MB	1762
SP, Class A	4	104MB	7.36MB	14
SP, Class C	16	1596.16MB	310.08MB	5
Tomcatv, 2048×2048	4	236MB	118.4KB	1993

Table 1: Memory Usage in MPI-SIM-DE and MPI-SIM-AM for the benchmarks.

Table 1 shows the total amount of memory needed by MPI-Sim when using the analytical (MPI-SIM-AM) and direct execution (MPI-SIM-DE) models. For Sweep3D, with 4900 target processors, the analytical models reduce memory requirements by *two orders of magnitude* for the 4×4×255 per processor problem size. Similarly, for the 6×6×1000 problem size, the memory requirements for the target configuration with 6400 processors are reduced by *three orders of magnitude*! Three orders of magnitude reduction is also achieved for Tomcatv, while smaller reductions are achieved for SP. This dramatic reduction in the memory requirements of the model allows us to (a) simulate much larger target architectures, and (b) show significant improvements in execution time of the simulator.

To illustrate the improved scalability achieved in the simulator with the compiler-derived analytical models, we consider Sweep3D. In this paper, we study a small subset of problems that are of interest to application developers. They are represented by two problem sizes: 4×4×255 and 6×6×1000 cells per processor, which correspond to a 20 million and 1 billion total problem size and need to run on 4,900 and 20,000 processors, respectively.

The scalability of the simulator can be seen in Figures 10 and 11. For the 4×4×255 problem size, memory requirements of the direct execution model restricted the largest target architecture that could be simulated to 2500 processors. With the analytical model, it was possible to simulate a target architecture with 10,000 processors. For the 6×6×1000 problem size, direct execution could not be used with more than 400 processors, whereas the analytical model scaled up to 6400 processors. Note that instead of scaling the system size, we could scale the problem size instead (for the same increase in memory requirements per process), in order to simulate much larger problems.

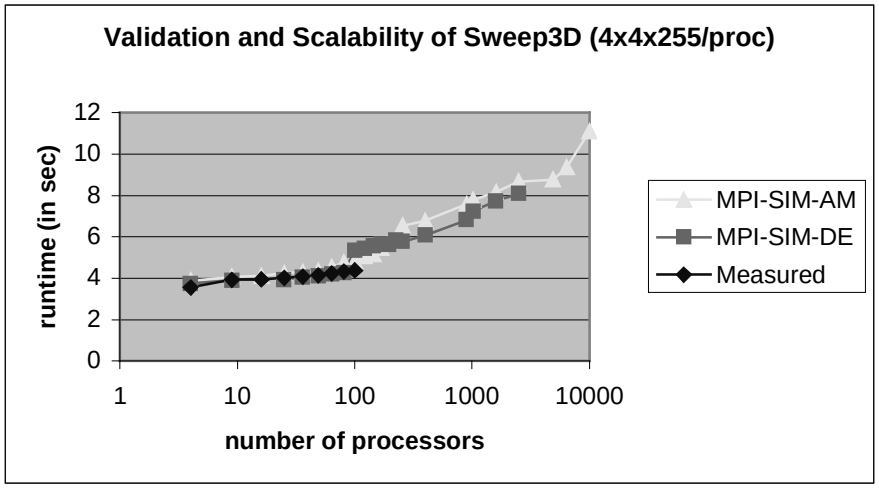


Figure 10: Scalability of Sweep3D for the 4×4×255 per Processor Size (IBM SP).

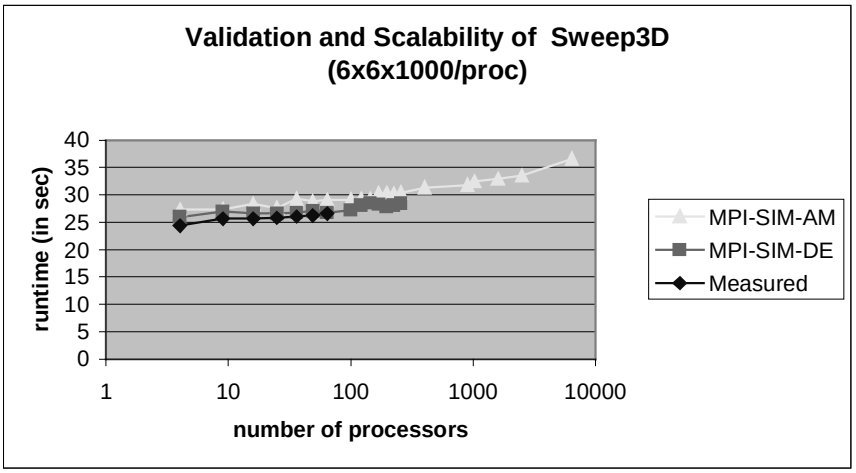


Figure 11: Scalability of Sweep3D for the 6×6×1000 per Processor Size (IBM SP).

4.4 Performance of MPI-Sim

The benefits of compiler-optimized simulation are not only evident in memory reduction but also in improved performance. We characterize the performance of the simulator in three ways:

1. absolute performance (i.e., total simulation time) of MPI-SIM-AM vs. MPI-SIM-DE and vs. the application,
2. parallel performance of MPI-SIM-AM, in terms of both absolute and relative speedups, and
3. performance of MPI-SIM-AM when simulating large systems on a given parallel host system.

Absolute Performance

To compare the absolute performance of MPI-Sim, we gave the simulator as many processors as were available to the application (#host processors = # target processors). Figure 12 shows the runtime of the application and the measured runtime of the two versions of the simulator running NAS SP class A. We observe that MPI-SIM-DE is running about twice slower than the application it is predicting. However, MPI-SIM-AM is able to run much faster than the application, even though detailed simulation of the communication is still performed. In the best case (for 36 processors), it runs 2.5 times faster. For 100 processors, it runs 1.5 times faster. The relative performance of MPI-SIM-AM decreases as the number of processors increases because the amount of computation in the application decreases with increased number of processors and thus the savings from abstracting the computation are decreased.

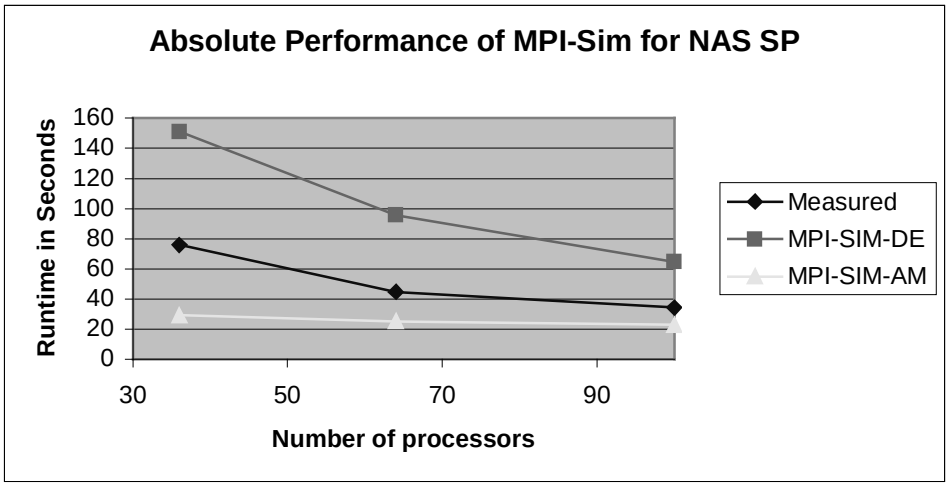


Figure 12: Absolute Performance of MPI-Sim for the NAS SP Benchmark, class A.

Even more dramatic results were obtained with Tomcatv, where the runtime of MPI-SIM-AM does not exceed 2 seconds for all processor configurations as compared to the runtime of the application which ranges from 130 to 10 seconds (Figure 13).

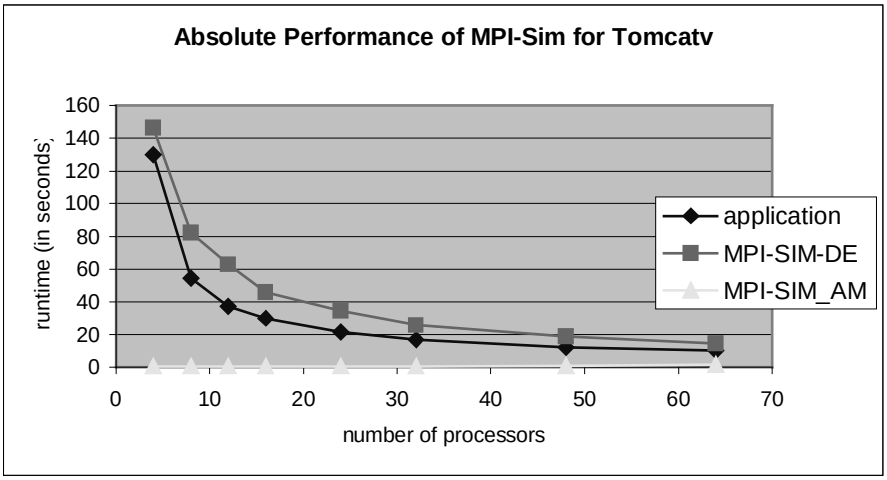


Figure 13: Absolute Performance of MPI-Sim for Tomcatv (2048x2048).

Parallel Performance

So far we have analyzed the performance of MPI-SIM-AM when the simulator had access to the same computational resources as the application. Obviously, when simulating large systems, the number of host processors will be less than the number of target processors, yet the simulator must provide results in a reasonable amount of time. Figure 14 shows the performance of both versions of MPI-Sim simulating the 150^3 Sweep3D running on 64 target processors when the number of host processors is varied from 1 to 64. The data for the single processor MPI-SIM-DE simulation is not available because the simulation exceeds the available memory. Clearly, both MPI-SIM-DE and MPI-SIM-AM scale well. The speedup of MPI-SIM-AM is also shown in Figure 15. The steep slope of the curve for up to 8 processors indicates good parallel efficiency. For more than 8 processors the speedup is not as good, reaching about 15 for 64 processors. This is due to the decreased computation to communication ratio in the application. Still, the runtime of MPI-SIM-AM is on the average 5.4 times faster than that of MPI-SIM-DE.

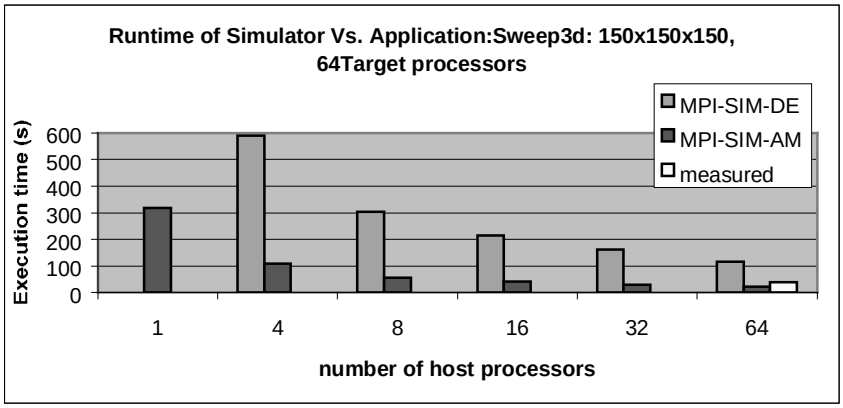


Figure 14: Parallel Performance of MPI-Sim.

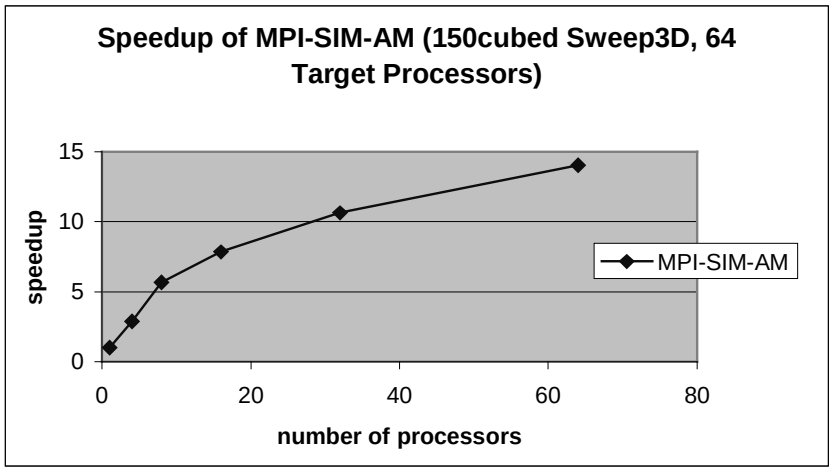


Figure 15: Speedup of MPI-SIM-AM for Sweep3D.

Performance for Large Systems

To further quantify the performance improvement for MPI-SIM-AM we have compared the running time of the simulators when predicting the performance of a large system. Figure 16 shows the running time of the simulators as a function of the number of target processors, when 64 host processors are used. The problem size is fixed per processor (6×6×1000), so the problem size increases with the increased number of processors. The figure clearly shows the benefits of the optimizations. In the best case, when the performance of 1600 processors is simulated (corresponding to the 57.6 million problem size) the runtime of the optimized simulator is nearly half the runtime of the original simulator.

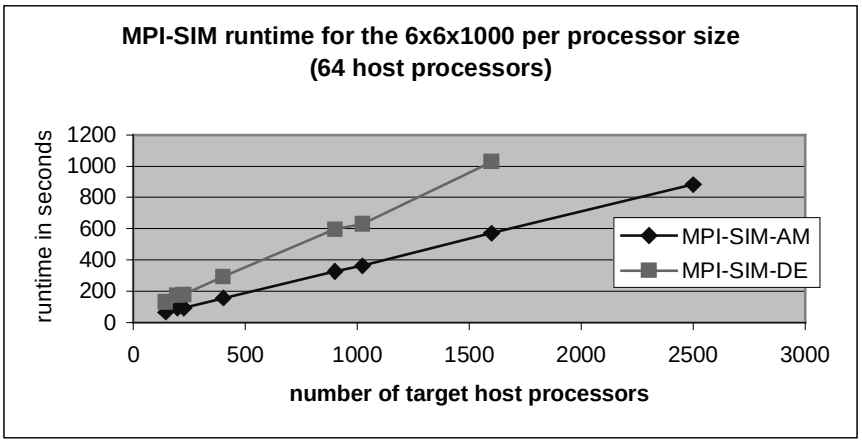


Figure 16: Performance of MPI-SIM when Simulating Sweep3D on Large Systems.

5 Conclusions

This work has developed a scalable approach to detailed performance evaluation of communication behavior in Message Passing Interface (MPI) and High Performance Fortran (HPF) programs. Our approach, which is based on integrating a compiler-derived analytical model with detailed parallel simulation, introduces relatively small errors into the prediction of program execution times. The compiler was used to build an intermediate static task graph representation of the program which enables the identification of computational tasks and control-flow that can be collapsed, and allows the derivation of the scaling functions for these collapsed regions of code. Program slicing was used to determine what portions of the computation in the collapsed tasks can be eliminated. The compiler was then used to abstract away parts of the computational code and corresponding data structures, replacing them with analytical performance estimates. The communication was retained by the compiler and was simulated in detail by MPI-Sim. The benefit we achieve is significantly reduced simulation times (typically more than a factor of 2) and greatly reduced memory usage (by two to three orders of magnitude). This gives us the ability to accurately simulate detailed performance behavior of much larger systems and much larger problem sizes than was possible earlier. In our current work, we are exploring a number of alternative combinations of modeling techniques. In this paper, we used an analytical model for sequential computations and detailed simulation for communication. An obvious alternative is to extend the MPI-Sim simulator to take as input an abstract model of the communication (based on message size, message destination, etc.) and use it to predict communication performance. Conversely, we can use detailed simulation for the sequential tasks, instead of

analytical modeling and measurement. Within POEMS, we aim to support any combination of analytical modeling, simulation modeling and measurement for the sequential tasks and the communication code. The static task graph provides a convenient program representation to support such a flexible modeling environment [2].

Acknowledgements

This work was performed while Adve was at Rice University. This work was supported by DARPA/ITO under Contract N66001-97-C-8533, "End-to-End Performance Modeling of Large Heterogeneous Adaptive Parallel/Distributed Computer/Communication Systems," (http://www.cs.utexas.edu/users/poems/). The work was also supported in part by the ASCI ASAP program under DOE/LLNL Subcontract B347884, and by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. We wish to thank all the members of the POEMS project for their valuable contributions. We would also like to thank the Lawrence Livermore National Laboratory for the use of their IBM SP.

References

1. V. S. Adve and J. Mellor-Crummey. “Using Integer Sets for Data-Parallel Program Analysis and Optimization,” *Proc. ACM SIGPLAN Symp. On Prog. Lang. Design and Implementation*, June 1998.

2. V. S. Adve and R. Sakellariou. “Application Representations for a Multi-Paradigm Performance Modeling Environment for Parallel Systems,” *International Journal of High-Performance and Scientific Applications* (to appear).

3. V. S. Adve, and R. Sakellariou. “Compiler Synthesis of Task Graphs for a Parallel System Performance Modeling Environment,” Technical Report CS-TR-98-332, Computer Science Dept., Rice University, December 1998.

4. D. Bailey, T. Harris, W. Shaphir, R. van der Wijngaart, A. Woo, and M. Yarrow. “The NAS Parallel Benchmarks 2.0,” Report NAS-95-090, NASA Ames Research Center, 1995.

5. R. Bagrodia, E. Deelman, S. Docy, and T. Phan. “Performance Prediction of Large Parallel Applications using Parallel Simulations,” *Proc of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Atlanta, May 1999.

6. R. Bagrodia, and S. Prakash. “Using Parallel Simulation to Evaluate MPI Programs,” *Proc. 1998 Winter Simulation Conference* (WSC’98), Washington D.C., December 1998.

7. E. Deelman, A. Dube, A. Hoisie, Y. Luo, D. Sundaram-Stukel, H. Wasserman, V. S. Adve, R. Bagrodia, J. C. Browne, E. Houstis, O. M. Lubeck, R. Oliver, J. Rice, P. J. Teller and M. K. Vernon. “POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems,” *Proc. First International Workshop on Software and Performance (WOSP ’98)*, October 1998.

8. P.M. Dickens, P. Heidelberger, and D.M. Nicol, “Parallel Direct Execution Simulation of Message-Passing Parallel Programs”, *IEEE Transactions on Parallel and Distributed System 1996.*

9. M. D. Dikaiakos, A. Rogers, and K. Steiglitz. “FAST: A Functional Algorithm Simulation Testbed,” *Proc. MASCOTS’94*, February 1994.

10. M. D. Dikaiakos, A. Rogers, and K. Steiglitz. “Functional Algorithm Simulation of the Fast Multipole Method: Architectural Implications,” *Parallel Processing Letters 6(1)*, March 1996, pp. 55-66.

11. M. Durbhakula, V. S. Pai, and S. V. Adve. “Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors,” *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA)*, January 1999, pp. 23-32.

12. S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems 12(1)*, January 1990, pp. 26-60.

13. S.S. Mukherjee, S.K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M.D. Hill, J.R. Larus, and D.A. Wood. “Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator”, *Workshop on Performance Analysis and Its Impact on Design (PAID), 1997.*

14. S. Prakash and R. Bagrodia. “Parallel Simulation of Data Parallel Programs,” *Proc. of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, Ohio, August 1995.

15. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. “The Wisconsin Wind Tunnel,” *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 48-60.

16. “The ASCI Sweep3d Benchmark Code,” http://www.llnl.gov/asci_benchmarks/.