

# Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures

Vikram S. Adve

*University of Illinois at Urbana-Champaign, Urbana, Illinois 61801*

Rajive Bagrodia

*University of California at Los Angeles, Los Angeles, California 90024*

Ewa Deelman

*Information Sciences Institute, University of Southern California,  
Los Angeles, California 90007*

and

Rizos Sakellariou

*University of Manchester*

Received July 5, 2000; revised February 8, 2001; accepted February 8, 2001

---

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 compiler techniques and a compiler-synthesized static task graph model to identify the subset of the computations whose values have no significant effect on the performance of the program, and to generate symbolic estimates of the execution times of these computations. For programs with regular computation and communication patterns, this information allows us to avoid executing or simulating large portions of the computational code during the simulation. It also allows us to avoid performing some of the message data transfers, while still simulating the message performance in detail. We have used these techniques to integrate the MPI-Sim parallel simulator at UCLA with the Rice dHPF compiler infrastructure. We evaluate the accuracy and benefits of these techniques for three standard message-passing benchmarks on a wide range of problem and system sizes. The optimized simulator has errors of less than 16% 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 regular message-passing programs on systems and problem sizes 10 to 100 times larger than is possible with the original simulator, or other current state-of-the-art simulators. © 2002 Elsevier Science

*Key Words:* parallel simulation; performance modeling; parallelizing compilers.

---

## 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. Considerable research is being done on both analytical and simulation models for performance prediction of complex, scalable systems. Analytical methods typically require custom solutions for each problem and may not be tractable for complex interconnection networks or detailed modeling scenarios; simulation models are likely to be the primary choices for general-purpose performance prediction. As is well known, however, detailed simulations of large systems can be *very* computation-intensive and their long execution times can be a significant deterrent to their widespread use.

The current generation of parallel program simulators uses two techniques to reduce model execution times: direct execution and parallel simulation. In direct execution, the simulator uses the available system resources to directly execute portions of the program. Parallel simulation distributes the computational workload of the simulation among multiple processors, while using appropriate synchronization algorithms to ensure that execution of the model produces the same result as if all events in the model were executed in their causal order. However, the current state of the art is such that even using direct execution and parallel simulations, the simulation of large applications designed for architectures with thousands of processors can run many orders of magnitude slower than their physical counterparts.

In this paper, we propose, implement, and evaluate *practical, automatic optimizations* that exploit compiler support to enable efficient simulation of highly scalable 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 isolate fragments of local computations and message data whose *values* do not affect the performance of the program. (We refer to these as redundant computations.) For example, computations that determine loop bounds, control flow, or message patterns and volumes all have an effect on performance, whereas many array element computations produce values that have no significant effect on performance. These computations can be abstracted away (and replaced by estimates of their performance) while simulating the rest of the program in detail to predict the performance characteristics of the application. Similarly, it is also possible to avoid performing data transfers for many messages whose values do not affect performance, while simulating the performance of the messages in detail. In addition to reducing simulation times, these optimizations can dramatically reduce the memory requirements for the simulation. In particular, if major program arrays are only referenced in redundant computations, they do not have to be allocated at all during the simulation. The memory savings can potentially allow much larger problem sizes and architectures to be studied than would otherwise be feasible.

There are three major aspects to the compiler analysis required to accomplish this optimization: (1) identifying the values within the program that could affect program performance; (2) isolating the computations and communications that

determine these values, and (3) generating symbolic estimates of the execution time of the remaining (i.e., redundant) computations. To perform the first step, we use a compiler-synthesized *static task graph* model [4, 5], an abstract representation that 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. The symbolic expressions in the task graph for control-flow conditions, communication patterns and volumes, and scaling expressions for sequential task execution times directly capture all the values that impact program performance—they are exactly the references that appear in those expressions. The second step uses a compiler technique called *program slicing* [22] to identify the portions of the computation that determine these values. The compiler can then emit *simplified MPI code* that contains exactly the computations that must be actually executed during the simulation (in addition to the communication), while the remaining code fragments are abstracted away. Finally, the compiler estimates the execution time of the abstracted code by using symbolic expressions parameterized by direct measurement. More sophisticated performance estimation for these sequential fragments is possible, but we do not do so here.

In order to demonstrate the impact of these optimizations, we have combined the MPI-Sim parallel simulator [6, 26–28] with the dHPF compiler infrastructure [2] into a program simulation framework that incorporates the new techniques described above. The MPI-Sim simulator simulates unmodified MPI programs and uses both direct execution and parallel simulation to achieve substantial reductions in simulation time. dHPF, in normal usage, compiles an HPF program to MPI and provides extensive parallel program analysis capabilities. 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.<sup>1</sup> In this work, we use the static task graph plus program slicing to perform the simulation optimizations described above. We have also extended MPI-Sim to exploit the information from the compiler 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.

We use several widely used benchmarks to evaluate the utility of the integrated framework: Sweep3D [1], a key ASCI benchmark; SP from the NPB benchmark suite [8], and Tomcatv, a SPEC92 benchmark. All three codes have regular computation and communication patterns. Tomcatv is an HPF benchmark and the optimized simulation is fully automatic using the integrated dHPF + MPI-Sim tool. The other two codes are existing MPI codes and we generated the simplified MPI versions by hand as they would be generated by a compiler and input these to MPI-Sim. The simulation models of each application were validated against measurements over a range of problem sizes and numbers of processors. The validation has been done for the distributed memory IBM SP architecture. The errors in the predicted execution times, compared with direct measurement, were at most 16% in all

<sup>1</sup> We believe that this step can also be done for a wide class of MPI codes directly, although the dHPF compiler does not currently do so.

cases we studied, and often were substantially less. Furthermore, the optimizations had a significant impact on the performance of the simulators: the total memory usage of the simulator using the compiler synthesized model was a *factor of 5 to 2000* less than the original simulator, and the simulation time was typically lower by a factor of 5–10. These dramatic savings allow us to simulate systems or problem sizes that are 10–100 times larger than is possible with the original simulator, without significant reductions in the accuracy of the simulator. For example, we were successful in simulating the execution of a configuration of Sweep3D for a target system with 10,000 processors! In several cases, the simulation time was *faster than the original program*.

One limitation of our work is that the potential benefits could be significantly lower in “irregular” applications where communication patterns or computational costs depend extensively on intermediate results. For such applications, fewer of the intermediate computations can be abstracted away by the compiler. We do not evaluate examples of such applications in this paper, but briefly discuss this issue in Section 6.

The remainder of the paper proceeds as follows. Section 2 first describes the state of the art of parallel program simulation, to set the stage for our work. Section 3 provides a brief overview of MPI-Sim and the static task graph model. Section 4 describes the optimization strategy and the compiler and simulator extensions required to implement the strategy. Section 5 describes our experimental results, and Section 6 presents our main conclusions.

## 2. RELATED WORK

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. To improve the simulators, direct execution has been used [21, 27, 28]. Direct-execution simulators make use of available system resources to directly execute portions of the application code and simulate architectural features that are of specific interest, or are unavailable. For example, simulators can be used to study various architectural components such as the memory subsystem or the interconnection network. Specifically, if one is interested in determining if a faster communication fabric for a network of workstations is of value for a given set of applications, one can run the application on the currently available machines and only simulate the projected network’s behavior. The benefits of this direct-execution simulation are obvious: first, one can estimate the value of the new hardware without the expense of purchasing it; second, one can do the simulation fast—there is no need to simulate the workstation’s behavior (for example, down to the level of memory references) since that part of the hardware is readily available.

Many of the early simulators were designed for sequential execution [9, 13, 14]. However, even with the use of abstract models and direct execution, sequential program simulators tended to be slow with slowdown factors ranging from 2 to 35 for each process in the simulated program [9]. Several recent efforts have been

exploring the use of parallel execution [10, 17, 17, 23, 24, 27, 28] to reduce the model execution times, with varying degrees of success. In order to have multiple simulation processes and maintain accuracy, simulations use protocols to synchronize the processes. One of the widely used protocols is the Quantum protocol, which lets the processes compute for a given quantum before synchronizing them. In general, synchronous simulators that use the quantum protocol must trade-off simulation accuracy with speed (frequent synchronizations slowdown the simulation, but synchronizing less frequently introduces errors, by possibly executing statements out-of-order). Both LAPSE [17, 18] and Parallel Proteus use some form of program analysis to increase the simulation window beyond a fixed quantum. MPI-Sim uses parallel discrete event simulation with the conservative protocol [25, 28]. Supported protocols include the *null message protocol (NMP)* [11], the *conditional event protocol (CEP)* [12], and a new protocol, which is a combination of the two [23]. As discussed in the next section, MPI-Sim exploits the determinism present in the communication pattern of the application to reduce, and in many cases, completely eliminate synchronization overheads.

Although simulation protocol optimizations 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, Dikaiakos et al. developed a tool called FAST that performs abstract simulations, which avoid execution of the computational code entirely [19, 20]. 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 FAST tool 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).

### 3. BACKGROUND AND GOALS

#### 3.1. MPI-Sim: Parallel Simulation of MPI Programs Using Direct Execution

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.

MPI-Sim supports most of the commonly used MPI communication routines, such as point-to-point and collective communications. In the simulator, all collective communication functions are implemented in terms of point-to-point communication functions, and all point-to-point communication functions are implemented using a set of core nonblocking MPI functions.

In general, the host architecture will have fewer processors than the target machine (for sequential simulation, the host machine has only one processor); this requires that the simulator provide the capability for multithreaded execution. Since MPI programs execute as a collection of single threaded processes, it was necessary to provide a capability for multithreaded execution of MPI programs in MPI-Sim. Further, memory and execution time constraints of sequential simulation led to the development of parallel implementations of MPI-Sim. MPI-Sim has been ported to multiple parallel architectures including a distributed memory IBM SP2 as well as a shared-memory SGI Origin 2000.

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 [27], which typically work as follows: Each application process in the simulation is modeled by a logical process (LP).<sup>2</sup> Each LP can execute independently, without synchronizing with other LPs, until it executes a *wait* operation (such as an MPI-Recv, MPI-Barrier, etc); a synchronization protocol is used to decide when such an LP can proceed. We briefly describe the default protocol used by MPI-Sim. Each LP in the model computes local quantities called earliest output time (EOT) and earliest input time (EIT) [7]. The EOT represents the earliest future time at which the LP will send a message to any other LP in the model; similarly the EIT represents a lower bound on the receive timestamp of future messages that the LP may receive. Upon executing a *wait* statement, an LP can safely select a matching message (if any) from its input buffer, that has a receive timestamp less than its EIT. Different asynchronous protocols differ only in their method for computing EIT. Our implementation supports a variety of such protocols as mentioned previously. The primary overhead in implementing parallel conservative protocols is due to the communications to compute EIT and the blocking suffered by an LP that has not been able to advance its EIT. We have suggested and implemented a number of optimizations to significantly reduce the frequency and strength of synchronization in the parallel simulator thus reducing unnecessary blocking in its execution [27, 28]. Our optimizations were geared towards exploiting determinism in applications. For

<sup>2</sup> In general, an LP can be used to simulate multiple application processes.

instance, consider an LP that is blocked at a receive statement and its input buffer contains a single message. In general, the LP cannot proceed by removing that message from the buffer as it might be possible that another message destined for this LP is in transit, and that message has a lower timestamp. However, if the receive statement is known by the process to be *deterministic*, it follows that there must exist a unique message that matches the receive statement. As soon as the LP receives this message, it can proceed without the need for any synchronizations with other LPs in the model. In the best case, if every receive statement in the model is known to be deterministic, no synchronization messages will be generated in the model and the parallel simulation can be extremely efficient.

The preceding optimizations have two limitations: first, it works only with communication statements that are *a priori* known to be deterministic. Second, 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.

### 3.2. The Static Task Graph Representation

As will be seen in the next section, the compiler analysis to be performed can be greatly facilitated by exploiting an appropriate abstract representation for the parallel behavior of the program. As part of the POEMS project [3, 16], we have developed an abstract program representation called the *static task graph* (STG) that captures extensive static information about a parallel program [4]. 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*.

To illustrate, an example MPI program and its STG are shown in Fig. 1. A node of the STG represents either some local computation, control-flow (an IF statement or a loop header), or a communication operation (such as a send, receive, or message-wait). Each node in the static graph actually represents a set of possible parallel tasks instantiated at runtime, where each individual runtime task represents a sequential computation with no intervening communication or synchronization operations. The set of tasks for a node is identified by a symbolic set of integer process identifiers, such as the set  $\{ [p]: 0 \leq p \leq P-1 \}$  for the compute node in the figure. 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 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  $p$  ( $1 \leq p \leq P-1$ ) sends to process  $q = p-1$ . (Note that the two IF statements in the code are captured by the mapping and do not need separate IF nodes in the STG.) Each communication node also includes additional symbolic information describing the pattern and volume of communication (not shown).

```

double precision A(NMAX, 1+ceil(NMAX/MINPROC))
double precision mdiag(MMI)
...
call mpi_comm_size(MPI_COMM_WORLD, P, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, myid, ierr)

read(*, N)
b = ceil(N / P)

...
do m = 1,MMI
  if (myid .lt. P) then
    <RECV A(2:N-1, (myid+1)*b+1) from
      processor myid+1>
  endif
  ...
  do mi = MMI,2,-1
    mdiag[mi] = mdiag[mi-1]
    ndiag = ndiag + mdiag[mi]
  enddo
  do I = 2,ndiag-1
    do J = max(2,myid*b+1), min(N, myid*b+b)
      A(I,J) = (A(I,J) + A(I,J+1)) * 0.5
    enddo
  enddo
  if (myid .gt. 0) then
    <SEND A(2:N-1, myid*b+1) to
      processor myid-1>
  endif
enddo

```

a) Original MPI Code

```

double precision mdiag(MMI)
double precision, 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)
...
do m = 1,MMI
  if (myid .lt. P) then
    <RECV dummy_buf(:) from processor myid+1>
  endif
  ...
  do mi = MMI,2,-1
    mdiag[mi] = mdiag[mi-1]
    ndiag = ndiag + mdiag[mi]
  enddo

  call delay((N-2) * (min(N,myid*b+b) -
    max(2,myid*b+1)+1) * w_1)

  if (myid .gt. 0) then
    <SEND dummy_buf(:) to processor myid-1>
  endif
enddo

```

c) Simplified MPI Code

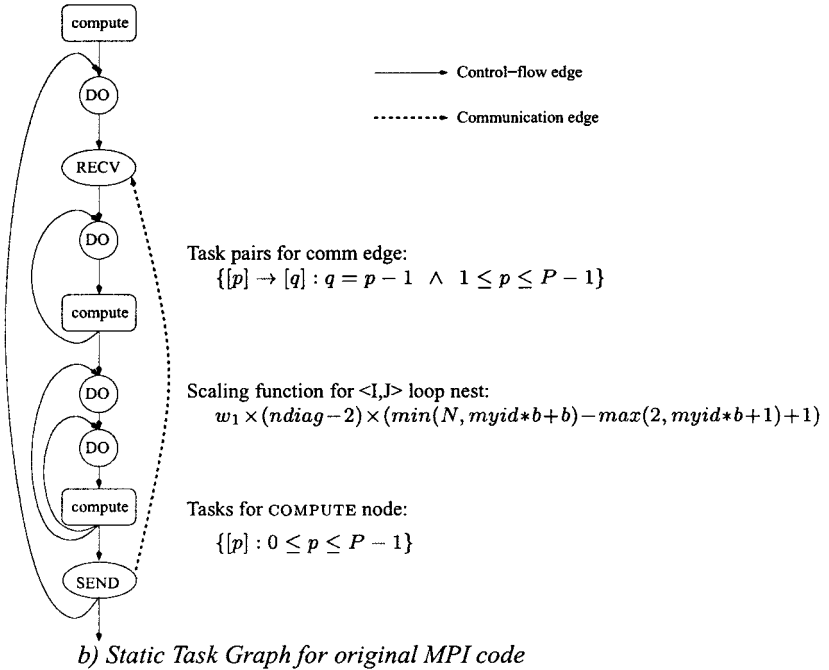


FIG. 1. Example to illustrate (a) a simple MPI program, (b) static task graph for MPI program, and (c) simplified MPI program for efficient simulation.

After constructing the initial task graph, we attempt to compute a *condensed task graph* in which the number of nodes is reduced without losing significant performance

information. In particular, we identify contiguous regions of computational tasks and/or control-flow in the STG that can be collapsed into a single condensed (or collapsed) task. For example, in Fig. 1, the  $I$  and  $J$  loops and their enclosed COMPUTE node can be collapsed into a single node. The first loop nest can be collapsed also. In later analysis, a single collapsed task can be treated essentially the same as a single computational task. The criteria for choosing tasks to collapse 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. As the baseline strategy for this work, we also do not collapse any conditional branches. There are occasionally branches that depend on values held in large arrays, but we have found that these often do not have a significant impact on program performance (because there is relatively little computation in each branch). It would be important to collapse such branches in order to achieve the savings in simulator memory and time we desire. This could be done by using profiling to estimate branching probabilities but we do not do so in our current compiler implementation.

While condensing 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 arbitrary program variables. We introduce variables that represent the number of computational operations of a sequence of statements in a single loop iteration (denoted  $c_i$  for task  $i$ ). The scaling function for the  $\langle I, J \rangle$  loop nest in the example is shown in Fig. 1b, with  $c_1$  denoting the time for each iteration of the inner loop. Each remaining computational node  $j$  that is not collapsed is simply assigned the scaling function  $c_j$ . We use these scaling functions in generating symbolic performance estimates for the abstracted computations, as described in Section 4.4.

Overall, the STG serves as a general, language- and architecture-independent representation of message-passing programs. This enables us to perform many of the analysis steps for optimizing simulation directly on the STG, independent of the code from which the STG was generated. 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 [5]. In the future, we will extract task graphs directly from existing MPI codes. This compiler support is valuable because it enables the techniques developed in this paper to be applied *fully automatically*, i.e., without user intervention, for efficient simulation of parallel programs.

#### 4. COMPILER-SUPPORTED TECHNIQUES FOR EFFICIENT LARGE-SCALE SIMULATION

This section begins by motivating the overall strategy we use to address the key restriction on simulation scalability identified above, namely, the time and memory required for simulating the detailed computations of the target program. We then describe more specifically how this strategy is accomplished.

#### 4.1. Optimization Strategy 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 example MPI code fragment in Fig. 1. This example is taken from the main loop nest of Sweep3D, but has been significantly simplified and abstracted to focus on the details of interest. The example code performs a pipelined (i.e., partially parallel) computation on the array  $A$ , where every processor receives boundary values from its right neighbor, executes two loop nests, and then sends its left boundary values to its left neighbor. The number of iterations of the second loop nest depends on the value of  $ndiag$  computed in the first loop nest. The communication pattern and the number of iterations of the loop nest also 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 all 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 array  $A$ , which are computed in the  $\langle I, J \rangle$  loop nest. We refer to these latter computations as *redundant computations* (from the point of view of performance estimation). If we can estimate the performance of the second computational loop nest analytically, we could avoid simulating the code of this loop nest, while still simulating the communication behavior in detail and estimating the overall execution time of the code.

We can achieve this optimization by using the compiler to generate the simplified code shown on the right in the figure. In this code, we have replaced the second loop nest with a call to a special simulator-provided delay function. We have extended MPI-Sim to implement such a function, which simply forwards the simulation clock on the local simulation thread by a specified amount. The compiler estimates the cost of the loop nest using the scaling functions computed during task graph construction. This estimated cost is shown as the argument to the delay call. (Section 4.4 describes how this cost is estimated.) Note in the example that the

compiler has avoided allocating the array  $A$  since it is no longer used in the simplified code, significantly reducing the memory required to simulate the program. The array *mdiag* must be retained in this case.

As an additional optimization, if the compiler can prove that the data transferred in the message are also “redundant,” the simulator can also avoid performing an actual data transfer, although it will simulate the message operation in detail. It can also avoid allocating any memory for the message buffer. This message optimization can lead to further savings in simulation time and memory usage. (This is not shown in the figure, but it would imply that the buffer *dummy\_buf* is not needed. The details are explained in Section 4.3 below.)

This paper develops automatic compiler-based techniques to perform the optimizations described above and evaluates the potential benefits of these techniques. In particular, our goal is to use the compiler-generated static task graph (plus additional compiler analysis) to avoid simulating or executing substantial portions of the computational code of the target program and sending unnecessary data. We use the task graph to identify the target references that directly determine the performance of each part of the program, and also to compute the scaling expressions for estimating the delay times. We use additional compiler analysis on the message-passing code to identify the computations that affect the values of target references, i.e., the nonredundant computations.

More specifically, there are four major challenges we must address in achieving the above goals, of which the first three have not been addressed in any previous system known to us:

1. 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.

2. While generating the simplified program, 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, by isolating the redundant computations in the program.

3. We must identify the messages whose contents do not directly affect the computation at the receiver and exploit this information to reduce simulation time and memory usage.

4. 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 4.4. Refining this approach is part of our ongoing work in the POEMS project.

The following subsections describe the techniques we use to address these challenges, and their implementation in dHPF and MPI-Sim. The next three subsections

describe the compiler analysis needed to identify redundant computations, the additional step to identify redundant communication operations, and the approach we use to estimate the performance of the eliminated code. Finally, we describe how we generate the simplified MPI code.

#### 4.2. Program Slicing for Identifying Redundant Computations and Data

The major challenge in performing our optimization correctly and effectively is to identify the redundant computations, i.e., the ones that can be safely eliminated. 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 at that statement [22]. The subset has to be conservative, limited by the precision of static program analysis, and therefore may not be minimal.

The first step in applying program slicing is to identify the variable references that directly affect the execution time of the program, which we call *target references*. The compiler-generated static task graph captures this information directly and precisely, allowing us to avoid a complicated and *ad hoc* analysis of the entire source code. In particular, the values that affect performance are exactly the variable references that appear in the retained control-flow of the condensed graph, in the scaling functions of the sequential tasks and communication events, and in the source and destination expressions of the communication descriptors (or the communication calls themselves). In the example of Fig. 1a, the target references include all the references in the various loop headers and the references to  $N$ ,  $myid$ , and  $b$  in the message calls (since these determine the size and pattern of the message).

Once the target references are identified, program slicing can be used to isolate the computations and data that affect the values of those references. Program slicing is essentially a reachability analysis on the dependence graph of the program, including both data and control dependences. In particular, given a particular target reference, we use a reachability analysis to identify the statements in the program that can affect the value of that reference through some chain of dependences (i.e., through some feasible path in the dependence graph). For example, in the code of Fig. 1a, the reference to  $ndiag$  in the header of the  $I$  loop is a target reference and there is a path of control and data dependences connecting the  $m$  loop header, the  $mi$  loop header, the assignments to  $mdiag[mi]$  and to  $ndiag$ , and the  $I$  loop (in order). Therefore, all these statements have to be retained. But there is no path of dependences starting from the assignment to  $A(I,J)$  and reaching any target reference of interest, so that statement and therefore its enclosing loops are identified as redundant.

Because slicing is a well-known compiler technique, we omit the details here. A state-of-the-art algorithm for program slicing is described in [22] and was used as the basis for our implementation. Applying this technique, however, requires that the target reference be part of the program so that it appears in the program dependence graph computed by the compiler. Some of the expressions of the static

task graph are not directly derived from corresponding expressions in the program, and therefore cannot be used as starting points for program slicing. For such expressions, we introduce dummy procedure call statements at appropriate points in the target program, passing those expressions as arguments, and then rebuild the program dependence graph. Now, these expressions can be used as starting points for slicing. The dummy procedure calls can later be eliminated.

Obtaining the memory and time savings we desire requires full 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 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 are currently adding full interprocedural slicing to the dHPF infrastructure.

The final output of the slicing analysis is the set of computations that must be retained in the simplified MPI code, while the remaining computations of the program (except for I/O statements and communication calls) can be considered redundant. The analysis also identifies the arrays of the program that are only used in the redundant computations, so that they can be eliminated in the generated code.

### 4.3. *Message Optimization for Simulation*

Because some computations are “redundant” (from the point of view of performance), the data transferred in some of the messages may also be redundant. If such cases can be identified, we can avoid performing the data transfers during the simulation, potentially leading to additional time and memory savings. Although this is conceptually similar to redundant computations, we discuss this “message optimization” separately because the mechanism for achieving this optimization is somewhat different, as explained below.

First, the compiler can identify redundant messages as a direct result of the program slicing analysis described above. The slicing analysis will directly identify those message receive calls that receive redundant values. More specifically, the technique described above to account for interprocedural side-effects during slicing identifies a message receive call as “producing” the data values in the destination data buffer, and the slicing algorithm identifies which such data values are redundant. The message send calls corresponding to the message receive are directly identified by the static task graph. The data transferred in these message send calls can also be marked “redundant,” and the slicing algorithm then proceeds to propagate this information further to identify additional redundant computations and communications.

The actual mechanism for performing the message optimization is as follows. We extended MPI-Sim to accept an extra flag on an MPI data-transfer call that identifies whether or not the data transfer is redundant. From the slicing results above,

the compiler flags the MPI calls for which the data transfers are redundant. The compiler also eliminates the allocation of buffers used by these messages, when generating the simplified MPI program.

If a call is not flagged, MPI-Sim simulates the call in detail (by sending the necessary protocol messages and predicting the end-to-end latency for the messages) and sends the data to the receiving simulation thread, so that the actual data are available to the simulated application. However, if the call is flagged by the compiler as redundant, then MPI-Sim still simulates the call in the detail with respect to the MPI communication protocol, but sends only an empty message to the receiving simulation thread. Since redundant receives are also flagged, the receiver does not copy the data in the buffer. The messages need to be present in the simulated application because they provide information about the synchronization in the program. Although this optimization does not reduce the number of messages sent, the size of the messages is reduced, and the memory used by the messages does not need to be allocated. This results in lower latencies incurred by the messages that are sent between processors as well as smaller communication overheads due to copying the data enclosed in the messages into/from the communication buffers. It also results in lower memory usage by the simulator.

#### 4.4. 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 off line simulation of sequential task execution times [16]. 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

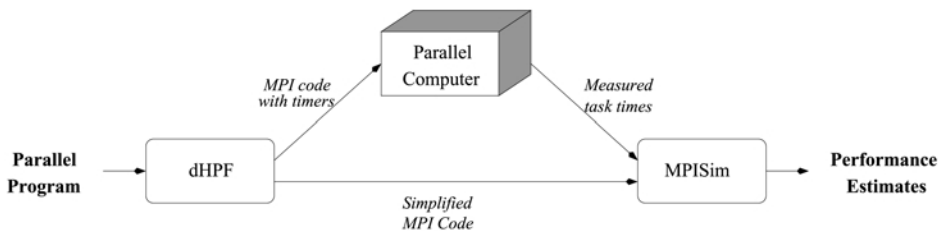


FIG. 2. Compilation, parameter measurement, and simulation for a parallel program.

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 delay 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 Fig. 2.

#### 4.5. Code Generation: Creating the Simplified MPI Program

Based on the condensed task graph, the results of the slicing analysis, and the symbolic performance estimates, we generate the simplified MPI program as follows. We eliminate any control-flow (loops and branches) of the original MPI code that is marked redundant by the slicing algorithm. Second, for each sequential task, the nonredundant computations are retained in the generated program, while the rest of the task is replaced with a single call to the the MPI-Sim delay function, and pass in an argument describing the estimated execution time of the task. For precise performance prediction, the simulator delay calls should not include the time for the retained computations since those will be simulated (and their time accounted for) explicitly. The execution time estimates computed above, however, apply to the entire task. In practice, we have found that the amount of nonredundant code is very small for most tasks and therefore we do not adjust the execution time estimates to account for this retained code. We insert a sequence of calls to a runtime function at the start of the program to read in the values of the  $w_i$  parameters from a file and broadcast them to all processors. Finally, we retain all the communication calls of the original program, adding the flag described above to mark which data transfers are redundant.

We must also eliminate all the storage not required in the simplified program. We directly eliminate program arrays that are marked redundant. If a redundant program array was referenced in some communication call, we replace that array reference with a reference to a single dummy buffer used for all the communication. For messages whose data transfer is redundant, we eliminate the allocation (and packing, if any) of the buffer used in such messages. For all other messages, 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 (and potentially multiple times), depending on when the required message sizes are known.

## 5. 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 information, 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 speedup as compared to sequential model execution, when simulating a large number of target processors.

Results in each of the above categories are presented for both types of the optimizations considered in this paper: elimination of local computations and elimination of data contents from large messages. We begin with a description of our experimental methodology and then describe the results for each of these issues in turn.

### 5.1. 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. The key arrays of this code are distributed across the processors in contiguous blocks in the second dimension, i.e., using the HPF distribution (**\*,BLOCK**).<sup>3</sup> Sweep3D, a Department of Energy ASCI benchmark [1], and SP, a NAS Parallel Benchmark from the NPB2.3b2 benchmark suite [8], 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; (b) the simplified MPI code with the calls to the MPI-Sim delay function, using the

<sup>3</sup> Like many benchmarks, Tomcatv executes for a fixed number of iterations whereas a real version would execute until convergence is reached. The convergence test alone would make many of the intermediate computations appear essential for performance estimation. In practice, such iterative codes would have to be evaluated for a predetermined number of iterations in order to benefit most from our optimizations.

techniques described in Section 4; and (c) the normal MPI code with timer calls inserted to measure the task time parameters, as described in Section 4.4. 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 evaluate the compiler techniques we developed for a wide range of regular message-passing codes.

For each application except NAS SP, we measured the task times (values of  $w_i$ ) for each problem size on 16 processors. These measured values were then used in experiments with the same problem size on different numbers of processors. For NAS SP, we measured the tasks only for a single problem size (on 16 processors), and used the same task times for experiments with all problem sizes and numbers of processors. 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).

## 5.2. Validation

The original MPI-Sim was successfully validated on a number of benchmarks and architectures [6, 27, 28]. The new techniques described in Section 4, 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 compiler-supported simulation based on these techniques.

For each application, the optimized simulator (henceforth denoted as MPI-SIM-TG) was validated against direct measurements of the application execution time and also compared with the predictions from the original simulator<sup>4</sup>. 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 Fig. 2. The size of Tomcatv used for the validation was  $2048 \times 2048$ . Figure 3 shows the results from 4 to 64 processors. MPI-Sim with the analytical model (MPI-SIM-TG) has slightly higher errors than MPI-Sim with direct execution (MPI-SIM-DE) for this benchmark. Nevertheless, the error in the performance predicted by MPI-SIM-TG was below 16% with an average error of 11.3% against the measured performance.

<sup>4</sup> The message optimizations further introduced do not modify the underlying communication model or introduce any other approximations, and thus do not affect simulator accuracy.

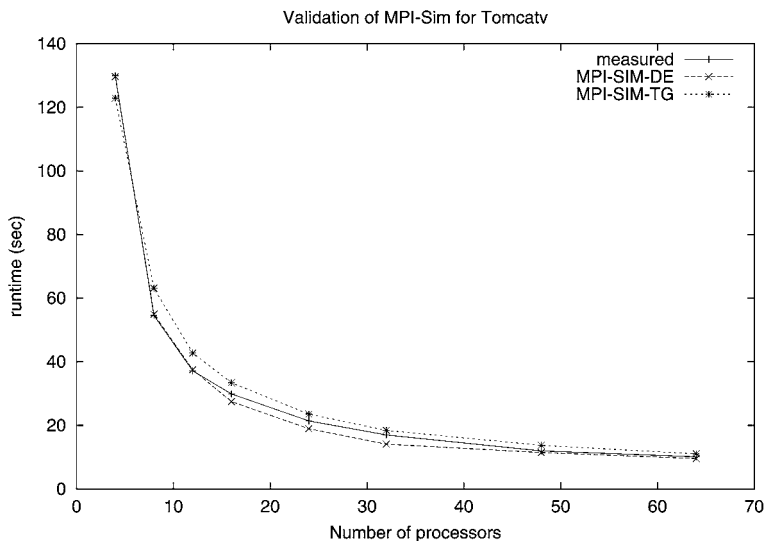


FIG. 3. Validation of MPI-Sim for  $(2048 \times 2048)$  Tomcatv (on the IBM SP).

Figure 4 shows the execution time of the model for Sweep3D with a total problem size of  $150 \times 150 \times 150$  grid cells as predicted using MPI-SIM-TG, 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 9.8%. On average, MPI-SIM-DE differed from the measured value by 3.7% and MPI-SIM-TG by 7.2%.

Finally, we validated MPI-SIM-TG 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,

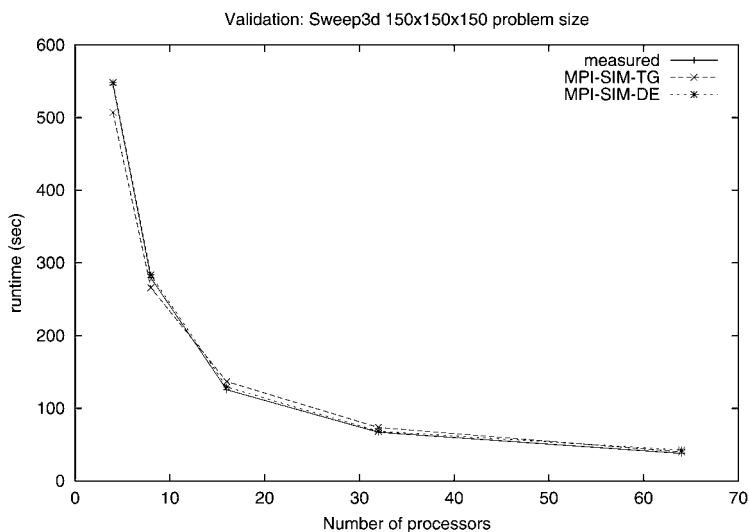


FIG. 4. Validation of Sweep3D on the IBM SP, fixed total problem size.

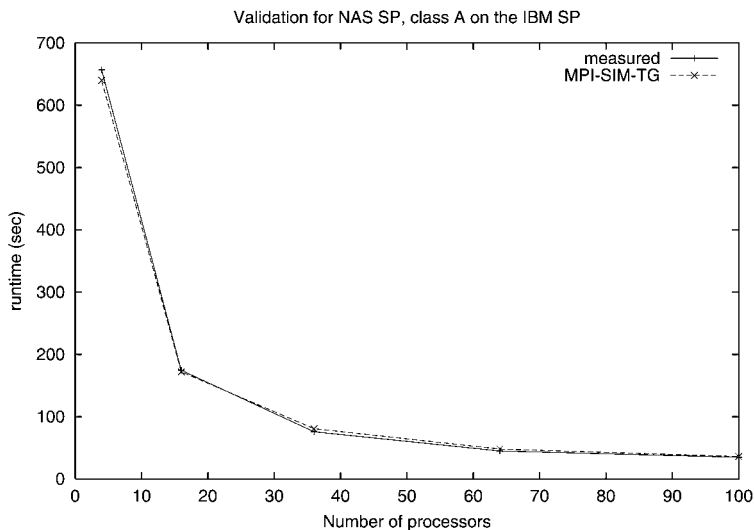


FIG. 5. Validation for NAS SP, class A on the IBM SP.

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 tasktimes were obtained from class A. This result is particularly interesting because, for programs of the same size, class C on average runs 16.6 times longer than class A. This 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 7 summarizes the errors that MPI-SIM-TG incurred when simulating the three applications. All the errors are within 16%, and all but a few are within about

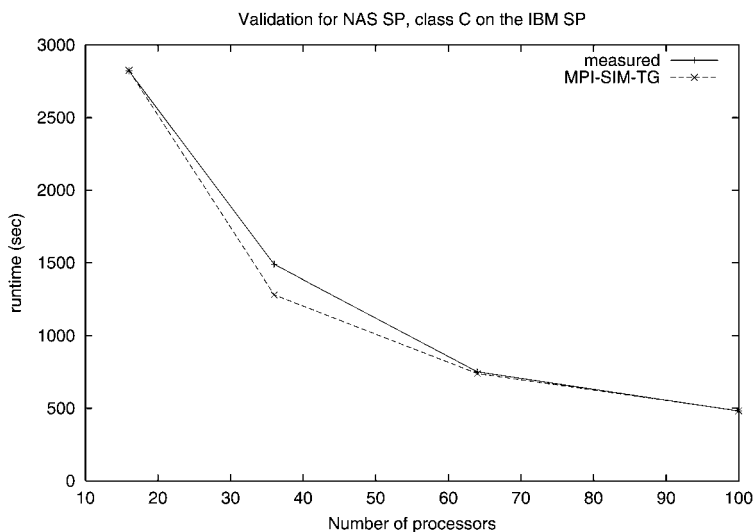


FIG. 6. Validation for NAS SP, class C on the IBM SP.

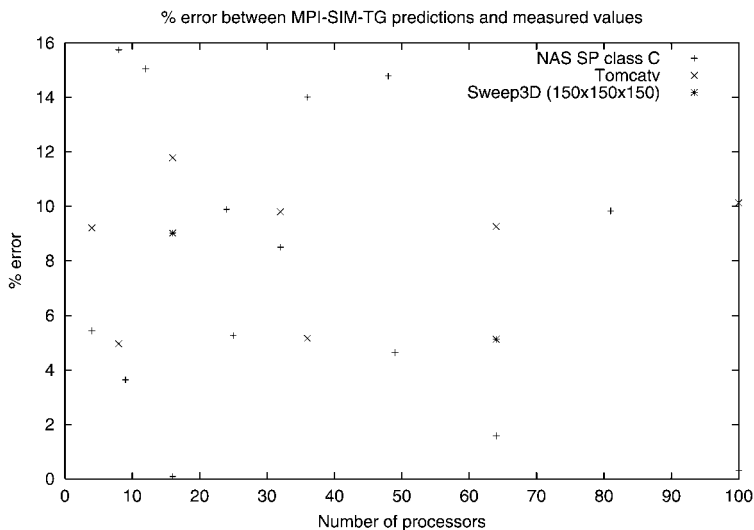


FIG. 7. Percent error incurred by MPI-SIM-TG when predicting application performance.

10%. 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.

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. SAMPLE is structured as a program where each process performs a given amount of computation and then communicates the data to other processes in the system. The amount of computation performed, the amount of communication and the communication patterns can be varied.

SAMPLE was validated on the Origin 2000. Two common communication patterns were selected: wavefront and nearest neighbor. For each pattern, the

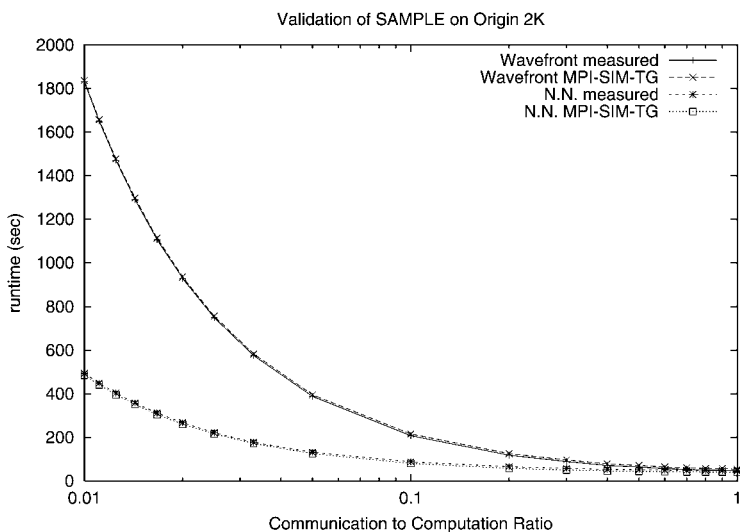


FIG. 8. Validation of SAMPLE on the origin 2000.

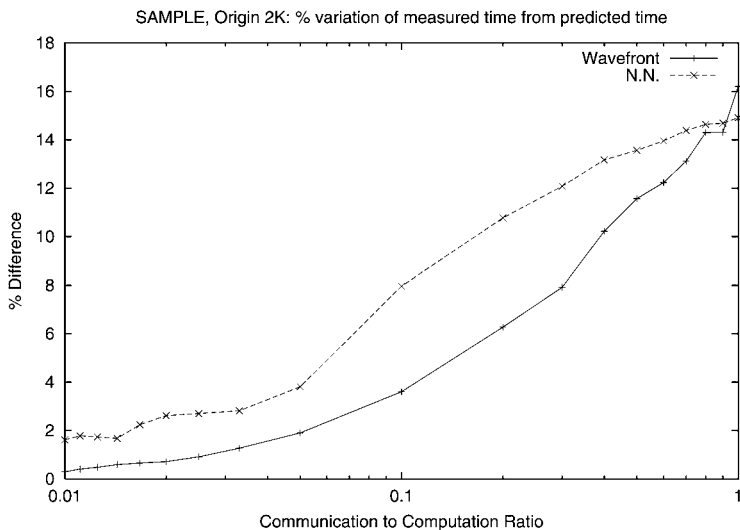


FIG. 9. Effect of communication to computation ratio on predictions.

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-TG prediction. In order to demonstrate better the impact of computation granularity on the validation, Fig. 9 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 computation granularity in the program decreases, the simulator incurs larger errors. This can be expected because both measurement errors and task time estimation errors can become relatively more significant. Nevertheless, the graph shows that the predicted values differ by at most 15% from the measured values, even for small computation to communication ratios.

TABLE 1

Memory Usage in MPI-SIM-DE and MPI-SIM-TG for the Benchmarks

	Number of processors	MPI-SIM-DE total memory use	MPI-SIM-TG total memory use	Memory reduction factor
Sweep3D, $4 \times 4 \times 255$ per proc. problem size	1	589KB	6KB	98
	4900	2884MB	30MB	96
Sweep3D, $6 \times 6 \times 1000$ per proc. problem size	1	33.599MB	19KB	1768
	6400	215GB	122MB	1762
NAS SP, Class A	4	104MB	7.36MB	14
NAS SP, Class C	16	1596.16MB	310.08MB	5
Tomcatv, $2048 \times 2048$	4	236MB	118.4KB	1993

The accuracy of MPI-SIM-TG for large computation to communication ratio (below 5% error) indicates that the slightly higher errors we observed for Tomcatv, Sweep3D, and NAS SP must be due to the presence of small computation to communication ratios.

### 5.3. 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.

Table 1 shows the total amount of memory needed by MPI-Sim when using the analytical (MPI-SIM-TG) 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  $4 \times 4 \times 255$  per processor problem size. Similarly, for the  $6 \times 6 \times 1000$  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 the 20 million cell total problem size, which can be divided into  $4 \times 4 \times 255$ ,  $7 \times 7 \times 255$ , and  $28 \times 28 \times 255$  per processor problem sizes which need to run on 4900, 1600, and 100 processors, respectively.

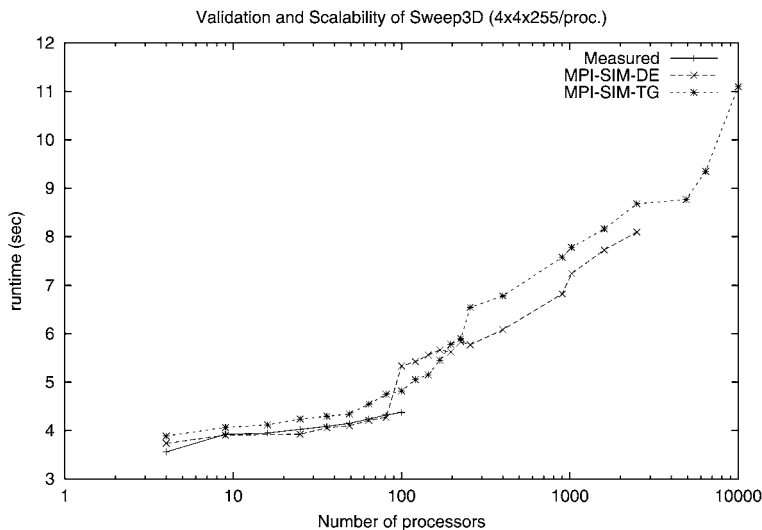


FIG. 10. Scalability of Sweep3D for the  $4 \times 4 \times 255$  per processor size (IBM SP).

The scalability of the simulator for the  $4 \times 4 \times 255$  problem size can be seen in Fig. 10. The 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. Since the application's predicted runtime for 10,000 processors is 11.0955 s and the runtime of the simulator for that configuration is 148.118 s, the simulator's slowdown is only 13.35! 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.

#### 5.4. 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 four ways:

1. performance gains when using the message optimization (MPI-SIM-TGMO) and MPI-SIM-TG as compared to MPI-SIM-DE,
2. absolute performance (i.e., total simulation time) of MPI-SIM-TG vs MPI-SIM-DE and vs the application,
3. parallel performance of MPI-SIM-TG, in terms of both absolute and relative speedups, and
4. performance of MPI-SIM-TG when simulating large systems on a given parallel host system.

*Effect of optimizations on simulator's performance.* To illustrate the performance improvements between MPI-SIM-DE, MPI-SIM-TG, which takes advantage of only the local optimizations and MPI-SIM-TGMO, which additionally optimizes

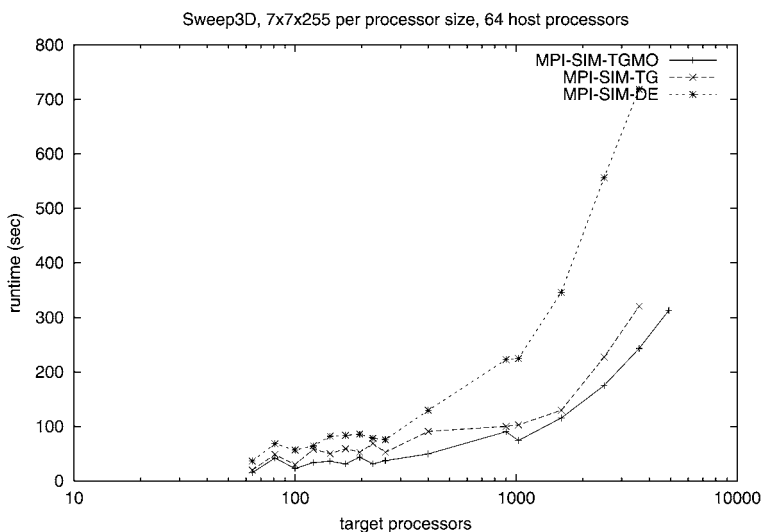


FIG. 11. Sweep3D,  $7 \times 7 \times 255$  per processor size, (MPI-SIM-TGMO is MPI-SIM-TG with message optimization).

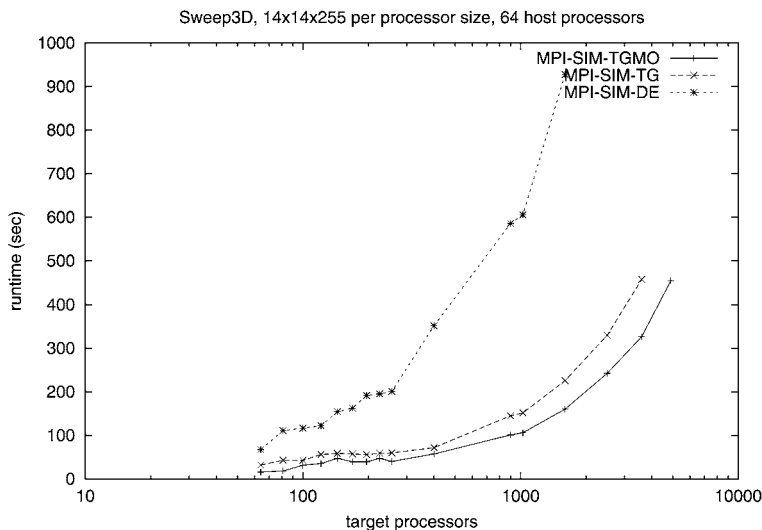


FIG. 12. Sweep3D,  $14 \times 14 \times 255$  per processor size.

the messages being sent, we conducted experiments on the three benchmarks. In case of Sweep3D we compared the performance of the three versions of the simulator when each had a given number of host processors available. The problem size per processor was fixed, and the number of target processors in the experiment was increased. This study demonstrates the ability of each simulator to efficiently simulate large problem sizes.

For NAS SP, since the problem size of the application is given (here class C), we fixed the number of target processors and varied the number of host processors available to the simulator. This study illustrates not only the relative performance of the simulators, but also their ability to use computational resources. Figures 11,

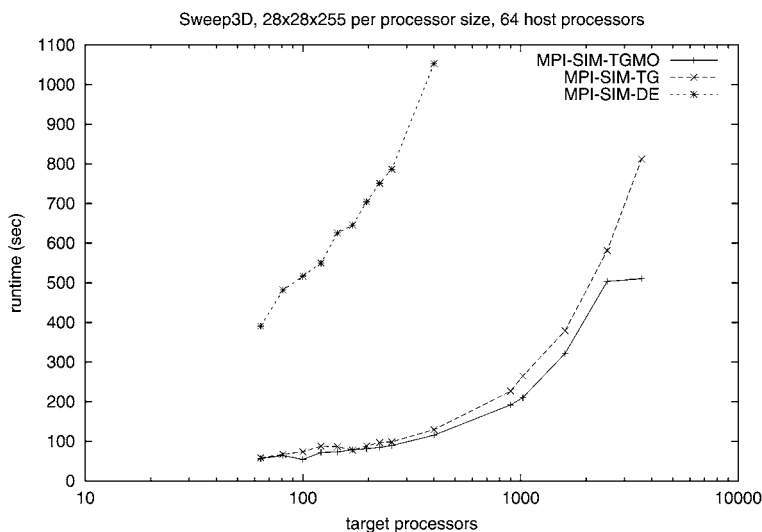


FIG. 13. Sweep3D,  $28 \times 28 \times 255$  per processor size.

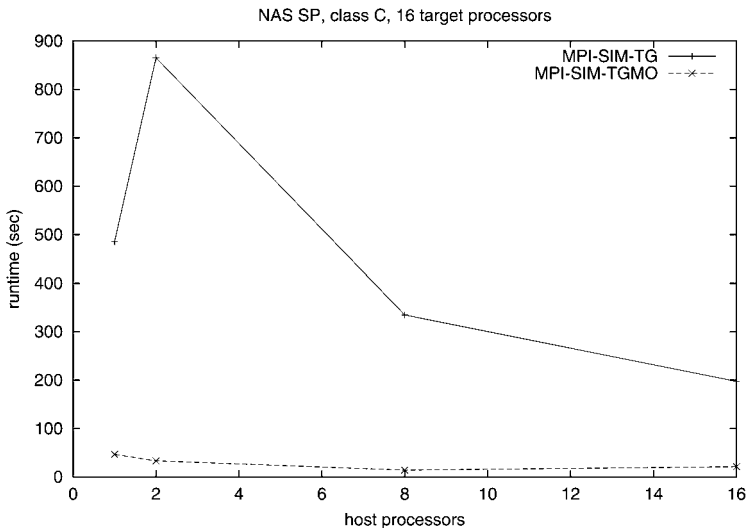


FIG. 14. A 16 target processor simulation of NAS SP, class C running on various numbers of host processors.

12, and 13 show the performance of MPI-SIM-TGMO, MPI-SIM-TG, and MPI-SIM-DE when simulating Sweep3D for three sizes per processor sizes:  $7 \times 7 \times 255$ ,  $14 \times 14 \times 255$ , and  $28 \times 28 \times 255$ . All simulators use 64 host processors to simulate up to 4900 target processors. The improvements in performance between MPI-SIM-DE and MPI-SIM-TG for the above sizes are on the average 39.7, 67.28, and 88.07%, respectively. As the problem size per processor grows larger, the amount of computation per processor increases thus the amount of computation abstracted away increases resulting in runtime savings.

Although the biggest performance gain is in the computation optimization, reducing the size of the messages sent, where possible, is beneficial. The simulation,

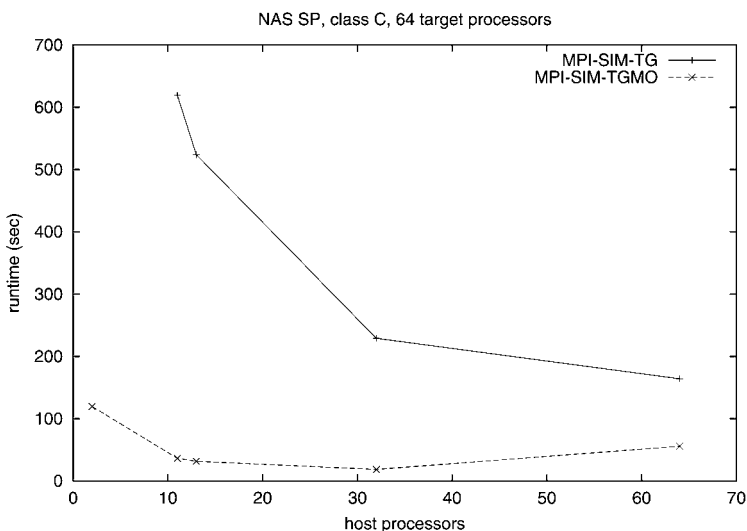


FIG. 15. A 64 target processor simulation of NAS SP, class C running on various number of host processors.

MPI-SIM-TGMO, runs faster than the simulation, which just optimizes the computation (MPI-SIM-TG). The improvements for the sizes  $7 \times 7 \times 255$ ,  $14 \times 14 \times 255$ , and  $28 \times 28 \times 255$  are 28.04, 31.23, and 13.9%, respectively. The benefits of the message optimizations are limited for the Sweep3D application, because it uses a large number of barrier synchronizations as well as collective operations such as (MPI\_Allreduce). These operations either take no data or only single data items.

We also observed great performance improvements for the NAS SP benchmark, class C, the largest size available in the suite. Figures 14 and 15 show the performance of MPI-SIM-TG and MPI-SIM-TGMO for two target processor configurations: 16 and 64. The simulations were run on a variety of host processors from 1 to 64. First, both MPI-SIM-TG and MPI-SIM-TGMO ran faster than the actual application. The measured runtime of the application executing on 16 processors is 2623.38 s, whereas running on 64 processors it is 790.67 s.

Additionally, Figs. 14 and 15 illustrate that the simulation can run an order of magnitude faster than MPI-SIM-TG when the message optimization is used. In Fig. 14, the jump in runtime for MPI-SIM-TG (from 1 to 2 host processors) is due to the large communication costs. The size of the messages sent between processors is 605,161 doubles. Therefore the cost of sending these messages increases considerably when more than one processor is used. When only 2 host processors are used this increased cost is not compensated by the increased computational power. However, as the number of host processors increases, better performance is achieved. Since the size of these large messages can be reduced to 0 in the MPI-SIM-TGMO simulation, this communication overhead is significantly reduced and the simulator performs substantially better than MPI-SIM-TG. As the number of target processors increases (to 64 in Fig. 15), the size of the messages in the simulation is reduced (to 370,441 for the 64 target processor code.) Still, using the message optimization results in an order of magnitude decrease in the simulator's runtime.

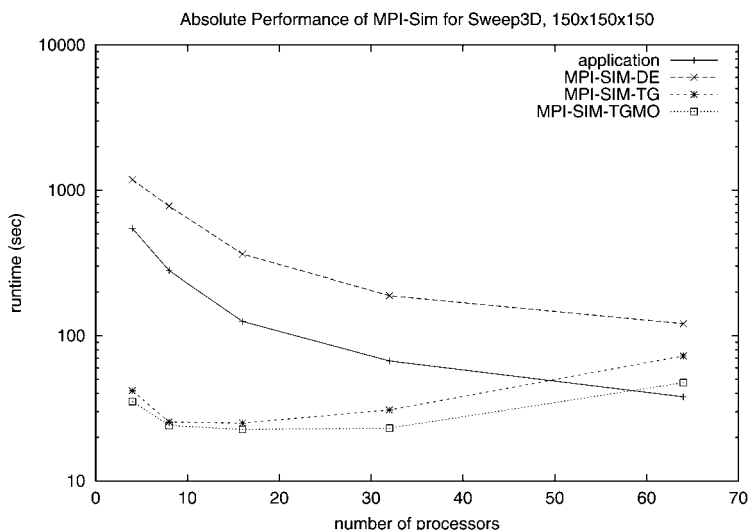


FIG. 16. Absolute performance of MPI-Sim for fixed total problem size Sweep3D.

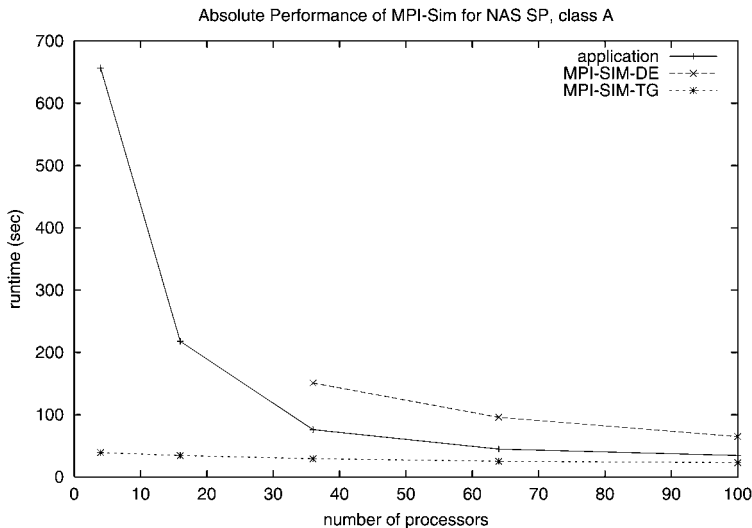


FIG. 17. Absolute performance of MPI-Sim for the NAS SP benchmark, class A.

*Absolute performance, local code optimization only.* 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 16 shows the absolute performance for Sweep3D with a total problem size of  $150^3$ . MPI-SIM-DE is on the average 2.8 times slower than the actual application (measured in the figure). However, MPI-SIM-TG is initially faster than the measured application starting at 13 times faster when running on 4 processors, gradually becoming only 2.2 times faster for 32 processors and finally being twice as slow as the application running on 64 processors. Message optimizations present in MPI-SIM-TGMO further decrease the simulators' runtime by on the average 18%

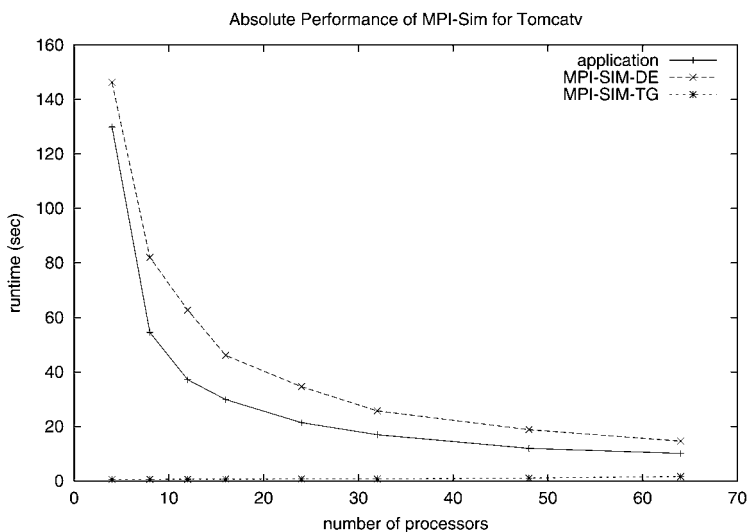


FIG. 18. Absolute performance of MPI-Sim for Tomcatv (2048 x 2048).

as compared to MPI-SIM-TG. Both MPI-SIM-TG and MPI-SIM-TGMO are always faster (on the average 16 and 18.5 times faster, respectively) than MPI-SIM-DE, showing the clear benefits of compiler optimizations. However, as the number of processors increases the amount of communication relative to the computation increases thus exposing the overhead of simulating the communications and making MPI-SIM-TG and MPI-SIM-TGMO slower than the application.

Figure 17 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-TG is able to run much faster than the application, even though detailed simulation of the communication is still performed. For 36 processors, it runs 2.5 times faster while, for 100 processors, it runs 1.5 times faster. The relative performance of MPI-SIM-TG 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.

Even more dramatic results were obtained with Tomcatv, where the runtime of MPI-SIM-TG does not exceed 2 seconds for all processor configurations as compared to the runtime of the application which ranges from 130 to 10 seconds (Fig. 18). This is due to the ability of the compiler to abstract away most of the computation. All that the simulator needs to directly execute is the skeleton code that controls the flow of the computation and communication patterns.

*Parallel performance.* To evaluate the parallel performance of the simulator, we study how well can it take advantage of increasing system resources (here processors) to solve a given problem (fixed total problem size). Figures 14 and 15 indirectly demonstrate the performance of the simulator; to illustrate the performance better, the speedup achieved for the 16 target configuration is depicted in Fig. 19. Although MPI-SIM-TGMO has a smaller runtime than MPI-SIM-TG, it scales well for only up to 8 host processors. This is because, as the number of host

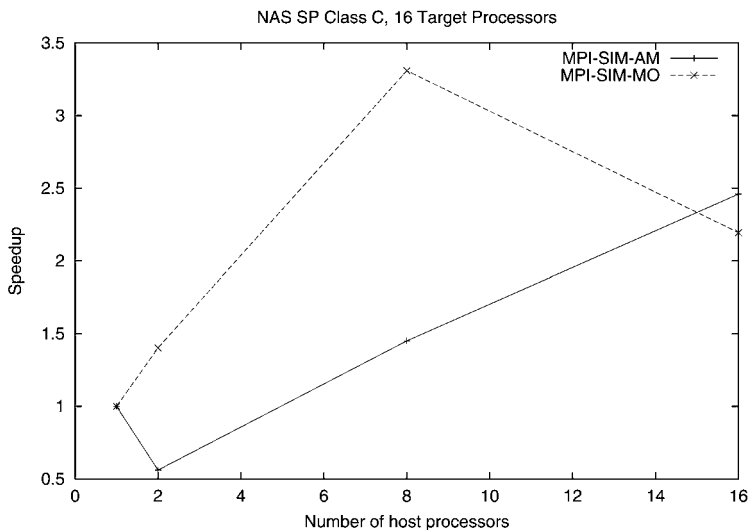


FIG. 19. Speedup of MPI-Sim for NAS SP.

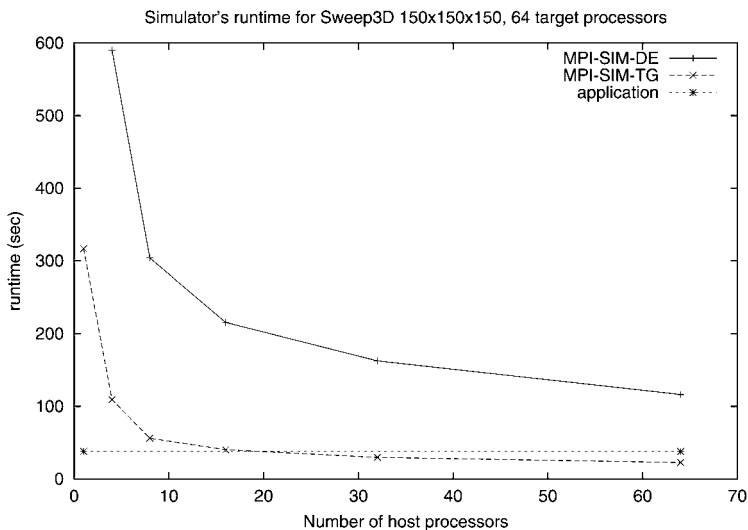


FIG. 20. Parallel performance of MPI-Sim.

processors increases, the communication overhead between the host begins to dominate the runtime. On the other hand, MPI-SIM-TG, which had to send large messages, suffers most when more than one host is used, but then is able to distribute that overhead among more processors.

Clearly, the performance of the simulator is better when larger systems are simulated. For the 64-target processor case (Fig. 15), the runtime decreases steadily as the number of processors is increased. However, using more than 32 host processors actually increases the simulator's runtime. (The 64 Target Class C could not be run on a single processor due to memory constraints, so direct speedup comparisons are not possible.) Better scalability is seen for the Sweep3D application. Figure 12 shows the performance of MPI-SIM-TG and MPI-SIM-DE simulating

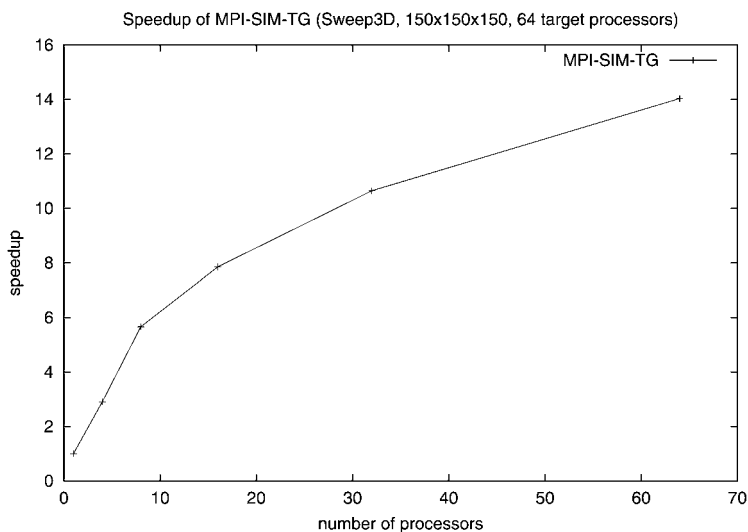


FIG. 21. Speedup of MPI-SIM-TG for Sweep3D.

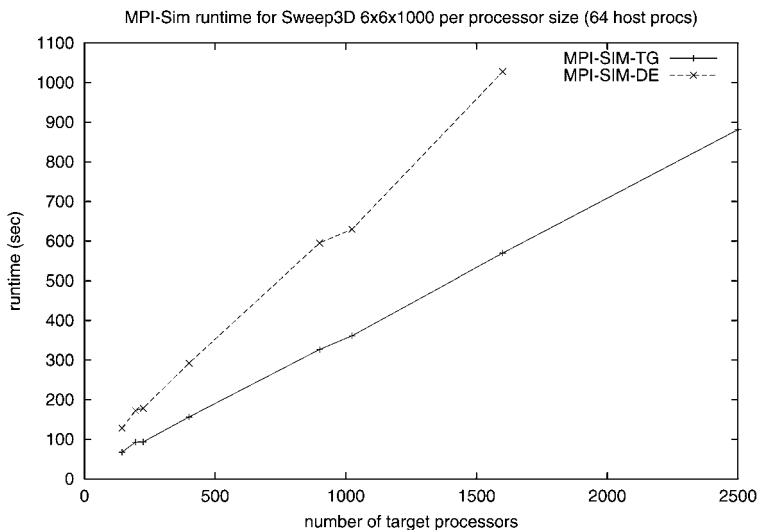


FIG. 22. Performance of MPI-SIM when simulating Sweep3D on large systems.

the  $150^3$  Sweep3D running on 64 target processors when the number of host processors varies from 1 to 64. The data for the single processor MPI-SIM-DE simulation are not available because the simulation exceeds the available memory. Clearly, both MPI-SIM-DE and MPI-SIM-TG scale well. The speedup of MPI-SIM-TG is also shown in Fig. 21. 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 impressive, 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-TG is on the average 5.4 times faster than that of MPI-SIM-DE.

*Performance for large systems.* To quantify further the performance improvement for MPI-SIM-TG, we have compared the running time of the two versions of the simulator (MPI-SIM-TG versus the original MPI-SIM-DE) when predicting the performance of a large system; in this case we choose to simulate a billion-cell problem for Sweep3D. The application’s developers envision this problem to utilize 20000 processors, which corresponds to a  $6 \times 6 \times 1000$  per processor problem size. Figure 13 shows the running time of the two versions as a function of the number of target processors when 64 host processors are used. The problem size is fixed per processor, 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. However, even with the optimizations, the memory requirements are still too large to be able to simulate the desired target system.

## 6. 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 is based on using compiler analysis

to identify portions of the computation whose *results* do not have a significant impact on program performance, and therefore do not have to be simulated in detail. The compiler builds an intermediate static task graph representation of the program which enables it to identify program values that have an impact on performance, and also enables it to derive scaling functions for computational tasks. The compiler then uses program slicing to determine what portions of the computations are not needed in determining performance. Finally, the compiler abstracts away those parts of the computational code (and corresponding data structures), replacing them with simple, analytical performance estimates. It also flags messages for which the data transfer does not have to be performed within the simulation. All of the communication code is retained by the compiler and is simulated in detail by MPI-Sim.

Our experimental evaluation shows that this approach introduces relatively small errors into the prediction of program execution times. 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 regular message passing programs for systems and problem sizes that are 10–100 times larger than is possible with current state-of-the-art simulation techniques.

One direction in which we are building on this work is to explore other strategies for exploiting compiler information to make parallel simulation more efficient. One promising possibility is to use the compiler estimates of computation times to increase the “lookahead” in conservative synchronization algorithms for parallel simulation [15]. By predicting how far ahead one simulation thread will run without performing communication events, other simulation threads can be notified to also simulate further ahead, thus increasing the effective parallelism and therefore the speedup of parallel simulation. A second interesting possibility is that the compiler optimizations may make optimistic parallel simulation more competitive. A major overhead in optimistic simulation is due to the costs of checkpointing and rollback of simulation state. Since our compiler-based techniques have achieved large reductions in simulator memory usage, these costs should be much smaller.

Perhaps the most significant limitation of our work is that the benefits could be much smaller for applications where the parallelism and communication patterns depend extensively on intermediate results of the computations. In particular, so-called “irregular” applications may have this property. The benefits for such applications will depend heavily on the ratio of the number of computations that affect future computation costs and communication patterns to the number of those that do not (e.g., computations that update statistics, test for convergence, check errors, etc., as well as any subsets of such applications that have a regular structure). Evaluating the benefits for such applications requires further research, and probably a refinement of the techniques developed here.

A different direction would be to explore whether the techniques described here can be extended to other types of distributed applications (i.e., nonscientific applications) that use network communication intensively. If very fast simulation techniques could be developed for such applications, they could prove valuable for predicting performance of network-intensive codes (e.g., distributed multimedia

codes). They could also prove valuable in controlling runtime optimization decisions such as object migration, load balancing, or adaptation for quality-of-service requirements, which are critical decisions for many distributed applications.

## ACKNOWLEDGMENTS

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." See also the project's web page at <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 F30602-96-1-0159. We thank all the members of the POEMS project for their valuable contributions. We also thank the Lawrence Livermore National Laboratory for the use of their IBM SP. This work was performed while Adve and Sakellariou were with the Computer Science Department at Rice University and Deelman was with the Computer Science Department at UCLA.

## REFERENCES

1. The ASCI Sweep3D Benchmark Code, [http://www.llnl.gov/asci\\_benchmarks/](http://www.llnl.gov/asci_benchmarks/).
2. V. Adve and J. Mellor-Crummey, Using integer sets for data-parallel program analysis and optimization, in "Proceedings of ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Quebec, Canada, 1998."
3. V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. J. Teller, and M. K. Vernon, POEMS: End-to-end performance design of large parallel adaptive computational systems, *IEEE Trans. Software Engrg.* **26** (November 2000), 1027–1049.
4. V. S. Adve and R. Sakellariou, Application representations for multiparadigm performance modeling of large-scale parallel scientific codes, *Internat. J. High-Performance Comput. Appl.* **14** (2000), 304–316.
5. V. S. Adve and R. Sakellariou, Compiler synthesis of task graphs for parallel program performance prediction, in "Proceedings of 13th Workshop on Languages and Compilers for Parallel Computing (LCPC'00), Yorktown Heights, NY, 2000."
6. R. Bagrodia, E. Deelman, S. Docy, and T. Phan, Performance prediction of large parallel applications using parallel simulations, in "Proceedings of 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Atlanta, GA, 1999."
7. R. Bagrodia, R. Meyer, M. Takai, C. Yu-An, Z. Xiang, J. Martin, and S. Ha Yoon, Parsec: A parallel simulation environment for complex systems, *Computer* **31** (1998).
8. D. Bailey, T. Harris, W. Shaphir, R. v. d. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-090, NASA Ames Research Center, 1995.
9. E. A. Brewer, A. Colbrook, C. N. Dellarocas, and W. E. Weihl, PROTEUS: A high-performance parallel-architecture simulator, in "Proceedings of 1992 ACM Sigmetrics and Performance, Newport, RI, 1992."
10. S. Chandrasekaran and M. D. Hill, Optimistic simulation of parallel architectures using program executables, in "Proceedings of 10th Workshop on Parallel and Distributed Simulation (PADS'96), 1996."
11. K. M. Chandy and J. Misra, Distributed simulation: A case study in design and verification of distributed programs, *IEEE Trans. Software Engrg.* **5** (1979), 440–452.
12. K. M. Chandy and R. Sherman, The conditional event approach to distributed simulation, in "Proceedings of Distributed Simulation Conference, 1989."

13. R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, The rice parallel processing testbed, in "Proceedings of 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Fe, NM, 1988."
14. H. Davis, S. R. Goldschmidt, and J. Hennessey, Multiprocessor simulation and tracing using tango, in "Proceedings of ICPP'91, 1991," pp. 99–107.
15. E. Deelman, R. Bagrodia, R. Sakellariou, and V. Adve, Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis, in "Proc. 15th Workshop on Parallel and Distributed Simulation (PADS 01), Lake Arrowhead, CA, USA, May 2001."
16. E. Deelman, A. Dube, A. Hoisie, Y. Luo, D. Sundaram-Stukel, H. Wasserman and 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, in "Proceedings of First International Workshop on Software and Performance (WOSP), Santa Fe, NM, 1998."
17. P. Dickens, P. Heidelberger, and D. Nicol, Distributed memory lapse: Parallel simulation of message-passing programs, in "Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), 1994."
18. P. M. Dickens, P. Heidelberger, and D. M. Nicol, Parallelized direct execution simulation of message-passing parallel programs, *IEEE Trans. Parallel Distrib. Systems* 7 (1996), 1090–1105.
19. M. D. Dikaiakos, A. Rogers, and K. Steiglitz, Fast: A functional algorithm simulation testbed, in "Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Durham, NC, 1994."
20. M. D. Dikaiakos, A. Rogers, and K. Steiglitz, Functional algorithm simulation of the fast multipole method: Architectural implications, *Parallel Process. Lett.* 5 (1996), 55–66.
21. M. Durbhakula, V. S. Pai, and S. V. Adve, Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ilp processors, in "Proceedings of 5th International Symposium on High Performance Computer Architecture (HPCA), 1999."
22. S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Trans. Programm. Languages Systems* 12 (1990), 26–60.
23. V. Jha and R. L. Bagrodia, Transparent implementation of conservative algorithms in parallel simulation languages, in "Proceedings of 1993 Winter Simulation Conference (WSC'93), Los Angeles, CA, 1993."
24. U. Legedza and W. E. Wehl, Reducing synchronization overhead in parallel simulation, in "Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96), 1996," pp. 86–95.
25. S. Prakash and R. Bagrodia, An adaptive synchronization method for unpredictable communication patterns in dataparallel programs, in "Proceedings of 9th International Parallel Processing Symposium, Santa Barbara, CA, 1995."
26. S. Prakash and R. Bagrodia, Parallel simulation of data parallel programs, in "Proceedings of 8th Workshop on Languages and Compilers for Parallel Computing (LCPC'95), Columbus, OH, 1995."
27. S. Prakash and R. L. Bagrodia, Mpi-sim: using parallel simulation to evaluate mpi programs, in "Proceedings of IEEE Winter Simulation Conference, Washington, DC, 1998."
28. S. Prakash, E. Deelman, and R. Bagrodia, Asynchronous parallel simulation of parallel programs, *IEEE Trans. Software Engrg.* 26 (May 2000), 385–400.
29. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, The wisconsin wind tunnel: Virtual prototyping of parallel computers, in "Proceedings of the 1993 ACM Sigmetrics Conference, 1993," pp. 48–60.