Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis

Ewa Deelman[†]
ISI/USC
deelman@isi.edu

Rajive Bagrodia
UCLA
rajive@cs.ucla.edu

Rizos Sakellariou
Manchester University
rizos@cs.man.ac.uk

Vikram Adve
UIUC
vadve@cs.uiuc.edu

Abstract

This paper addresses the issue of efficient and accurate performance prediction of large-scale message-passing applications on high performance architectures using simulation. Such simulators are often based on parallel discrete event simulation, typically using the conservative protocol to synchronize the simulation threads. The paper considers how a compiler can be used to automatically extract information about the lookahead *present in the application, and how this can be used to improve the performance of the null protocol used for synchronization. These techniques are implemented in the MPI-Sim simulator and dHPF compiler, which had previously been extended to work together for optimizing the simulation of local computational components of an application. The results show that the availability of lookahead information improves the runtime of the simulator by factors ranging from 9% up to two orders of magnitude, with 30-60% improvements being typical for the real-world codes. The experiments also show that these improvements are directly correlated with reductions in the number of null messages required by the simulations.*

.

1 Introduction

Direct-execution simulators make use of available system resources to execute directly portions of the application code and simulate architectural features that are of specific interest, or are unavailable. For instance, direct execution simulators can be used to study various architectural components such as the memory subsystem or the interconnection network. The benefits of this direct-execution simulation are obvious: first, one can estimate the value of the new hardware without the expense of manufacturing or 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. However, the constraint of direct execution requires the simulator to use at least as much memory as the target application and constrains the simulator to run at least as long as the application.

To alleviate the cost of direct execution while still maintaining accuracy, in recent work we used compiler support to optimize the simulation of local code [3]. Compiler analysis identifies portions of local code whose *results* do not affect program performance. These sections of the local code are replaced by estimates of their execution time using an analytical model of their performance built by the compiler. Data used only in such computations can also be eliminated. As a result, we observed dramatic savings both in the simulator's runtime as well as its memory requirements [3].

In this paper, we examine how compiler analysis can be used to improve *lookahead* in parallel simulation, and hence further reduce simulation time. When a simulation thread (Logical Process (LP)) knows that before sending the next message, it will process a local code block whose execution time can be predicted, the LP can communicate that information (increased lookahead) to other LPs in the system, possibly allowing those LPs to process events that might not be otherwise processed.

This paper makes three main contributions to parallel simulation of parallel applications:

1. It demonstrates how compiler analysis of a target application program can be used to extract lookahead information useful to a parallel simulation algorithm.
2. It augments an existing compiler-supported simulation system (the MPI-Sim simulator and the Rice dHPF parallelizing compiler) to incorporate this technique in parallel simulation, and
3. It presents a preliminary experimental evaluation demonstrating the potential benefits of this technique.

We present preliminary results evaluating the potential improvements that could be obtained by exploiting lookahead information when using the null message protocol [11] for LP synchronization. Using two synthetic codes and two standard applications (an ASCI benchmark, Sweep3D, and the NAS benchmark, SP), we compare how the simulator performs when it has no lookahead information versus when it can use the compiler-generated lookahead. The synthetic codes allow us to vary the granularity of computation between communications, which has a direct impact on the benefit of lookahead information. The two real codes have deterministic

[†] This work was performed while at UCLA.

communication patterns for which MPI-Sim would not require a synchronization protocol, but by forcing the null protocol we can examine how much similar applications would potentially benefit from the information about lookahead. Our results show that the simulations of the two real codes are 9% to 60% faster when using the lookahead information, and the improvements are higher when the granularity of local computations is higher. The synthetic applications show improvements by up to two orders of magnitude, as the granularity of computations is varied. In all the benchmarks, the improvements in simulation time are directly caused by reductions in the number of null messages required for the simulations. Although these results are preliminary and need to be confirmed by studies with additional applications, they illustrate the large potential benefits that could be achieved via compiler analysis of lookahead in target applications.

2 Related Work

Many of the early program simulators were designed for sequential execution [8, 12, 13]. However, even with the use of direct execution, sequential program simulators tended to be slow with slowdown factors ranging from 2 to 35 for each process in the simulated program [8]. Several efforts have been exploring the use of parallel execution [10, 14, 17, 19, 22] to reduce the model execution times, with varying degrees of success. Many such simulators use sequential or parallel implementations of the quantum protocol. In order to support multiple simulation processes (possibly executing on multiple processors) and maintain accuracy, parallel simulation protocols are used to synchronize the processes. The Quantum protocol 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 [24].

Parallel simulators include MPI-Sim [6, 21], described in the next section, the Wisconsin Wind Tunnel (WWT) [18, 22], a shared memory architecture simulation engine and SimOS [24], a complete system simulator (multiple programs plus operating system). SimOS, which simulates the MIPS architecture, takes into account system details such as cache and CPU models as well as device drivers. It is possible to use the emulation mode, which in part uses direct execution to characterize the program execution. In the emulation mode, the simulation is still ten times slower than real time. The main drawback of SimOS is that it does not use any synchronization protocol when running multiple simulation processes on a parallel platform [23], thus reducing the accuracy of the simulations.

Although MPI-SIM is the only simulator that identifies communication patterns and directly exploits them for the purposes of synchronization, other simulators have used techniques to reduce the synchronization overhead. Among them are LAPSE [14] and Parallel Proteus [17]. Both LAPSE and Parallel Proteus use some form of program analysis to increase the simulation window beyond a fixed quantum, without sacrificing accuracy. LAPSE uses a quantum protocol called WHOA (Window-based Halting On Appointments) and runtime analysis to determine the size of the simulation quantum. An appointment is the earliest time the message can be placed in the network. Adding the latency of the network to the appointment time gives the earliest possible arrival for the message. Processes use the minimum of their appointment times (incoming) to determine whether a message can be processed or not. Parallel Proteus reduces the synchronization overhead caused by frequent barriers of the quantum protocol by using *predictive barriers* and *local barriers*. The predictive barriers method uses runtime and compile time analysis to determine, at the beginning of a simulation quantum, the earliest simulation time at which any process will send a message to any other process.

In previous work [21], we designed a novel approach to synchronization in which the blocking time at the *receive* statement is reduced by analyzing the communication patterns in the program. Specifically, each simulation process uses this analysis to locally identify whether an incoming application message is safe to process right away or whether synchronizations with other processes are necessary. In some cases, the optimization resulted in simulations where no synchronization was necessary.

3 Background

3.1 Simulation of Large-Scale Applications with MPI-Sim

The starting point for our work is MPI-Sim [6, 21], 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. In the compiler-enhanced version of MPI-Sim, portions of the local code are modeled by an analytical performance model, while the remaining local code is directly executed. 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 non-blocking MPI functions. The simulator has been validated against several MPI implementations including those on the IBM SP and SGI Origin 2000 [6].

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 [20], which typically work as follows: Each application process in the simulation is modeled by a Logical Process (LP). 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) [5]. 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. However, in this paper, we concentrate on the Null Message protocol [11], where the EOT is communicated between the LPs via *null messages*. In our model, when an LP is blocked at a receive statement and cannot find a matching message, the LP requests null messages from all LPs in the system (or a subset of LPs with which it communicates) and recomputes its EIT whenever a null message arrives. An LP can get a null message request at any time, at which point it returns its EOT. Later, we show how we use the compiler derived analytical models to extract the lookahead present in the application and thus improve an LP's estimate of its EOT.

3.2 Compiler Analysis

In previous work [3], we implemented and evaluated compiler techniques to improve the performance of parallel simulation of very large message-passing parallel programs. The key idea underlying this work was to apply compiler analysis to locate fragments of local computation whose resulting values do not affect performance, and to avoid simulating those fragments in detail by replacing them with (symbolic) analytical performance estimates. For example, computations of values that determine loop bounds, branches, message patterns, and message sizes all have a direct impact on performance. In contrast, the results of other computations do not affect performance, and only their execution times are required for performance prediction. The latter, which we term as '*redundant*', do not need to be simulated in detail and can be abstracted away and replaced by an analytic performance estimate of their execution time, while simulating the rest of the program in detail. During simulation, the simulator can use the analytical estimate to advance the clock accordingly. As a corollary, it is also possible to avoid performing data transfers[1] for many messages whose values do not affect performance, while simulating the *performance* of the messages in detail.

The compiler analysis for accomplishing the above has three major aspects: (1) identifying the values in the program that do not affect performance (a *value* is a pair <*variable, statement*> representing the data stored in that variable at that statement); (2) identifying computations that only affect these values and therefore can be abstracted away; and (3) generating symbolic estimates for the execution time of these computations.

For the first step, we use a compiler-synthesized *static task graph* model [2, 4], an abstract program representation that identifies the sequential computations (tasks), the parallel structure of the program (task precedences, 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 capture all these program variables that have a direct impact on program performance.

For the second step, we use a compiler technique called *program slicing* [16] to identify those portions of the computation that determine the values of those variables; these are exactly the computations that must be retained. (Given a particular value in a program as defined above, program slicing uses data and control dependence information to identify those portions of the computations that may directly or indirectly affect that value in some execution of the program. This analysis must be performed interprocedurally, and can be performed for an entire set of values at once.) The compiler then generates *simplified MPI code* that contains those computations plus the communication. The remaining code fragments are replaced by a call to a function that will be interpreted by the simulator as a command to advance its clock by a specified value; this value should correspond to the execution time of the abstracted computation.

[1] This saves simulation time because performing the data transfer may require significant overhead, e.g., if the source and destination threads of the message are mapped to different host processors in the simulation.

Finally, in order to estimate the execution time of the abstracted code, the compiler generates simple symbolic expressions parameterized by direct measurement.

The above techniques have been implemented in the Rice dHPF compiler [1]. In [3], we evaluated the above techniques for three benchmarks: Sweep3D [25], a key ASCI benchmark; NAS SP from the NAS benchmark suite [7] and Tomcatv, a SPEC92 benchmark. Over a wide range of problem sizes and numbers of processors (on the distributed memory IBM SP), the errors in the predicted execution times, compared with direct measurement, were at most 17% in all cases we studied, and often were substantially less (the direct execution MPI-Sim had errors of about 7%). Moreover, in each application, the compiler techniques led to a significant reduction in simulator memory usage (up to 2000 times) and simulation time (up to 10 times), thus allowing us to simulate problem sizes *up to 100 times larger* than what was possible with state-of-the-art simulation tools before.

This paper extends the preceding work in two directions. First, we assume that the receive statements are not deterministic and an LP must use some synchronization algorithm to identify safe messages (we use a conservative null message algorithm for this). Second, we use the compiler-derived representation to extract the lookahead present in the application to improve performance of the null message protocol.

4 Lookahead Extraction

Lookahead plays an important role in improving the performance of conservative simulation protocols. In the context of the application, we focus on portions of the code where the simulation thread is blocked (such as in blocking receives and sends). When a logical process (LP) executes a receive statement, it checks if its input message queue contains any *safe* messages, i.e., any message with a timestamp less than the EIT of the LP. If so, the safe message(s) can be processed; otherwise the LP is blocked until its EIT is advanced using the underlying null message based protocol.

In general, for program simulations using direct execution, the lower bound on the EOT of an LP is its current simulation time (T) plus L, the minimum latency of any message that can be sent. However, if the LP can compute an accurate lower bound on the execution time of a local code block that precedes any message transmissions, perhaps via compiler analysis, it can compute a more accurate EOT. At some point in its execution, let T_{LC} represent the execution time of a code block of an LP, then its EOT becomes $T+L+T_{LC}$, and thus enables the blocked LPs to have a better estimate of the EIT. In previous work, researchers have estimated this execution time using pre-simulation [15]. In this paper, we show how this can be computed using compiler analysis, and used to improve the efficiency of the resulting model.

Example 1:

```
MPI_Recv(……)
for (j=1; j<N; j++) {
      mdiag[j] = mdiag[j-1];
      ndiag = ndiag + mdiag[j];  }
MPI_Recv(&a, … …);
for (k=0; k<ndiag; k++ )
      a[k] = a[k-1] + …
MPI_Send(&a, … …);                    (a)
```

```
for (j=1; j<N; j++) {
      mdiag[j] = mdiag[j-1];
      ndiag = ndiag + mdiag[j]; }
MPI_Recv(&a, … …);
advance_clock(ndiag * w_3);
MPI_Send(&a, … …);                    (b)
```

```
for (j=1; j<N; j++) {
      mdiag[j] = mdiag[j-1];
      ndiag = ndiag + mdiag[j]; }
set_lookahead(ndiag* w_3);
MPI_Recv(&a, … …);
reset_lookahead();
advance_clock(ndiag*w_3);
MPI_Send(&a, … …);                    (c)
```

Example 1a shows a portion of a code where boundary conditions of a loop are calculated, a receive statement is posted[2], values of an array "a" are calculated, and finally the computed data is sent to the next processor. Such a code structure is common in many scientific applications, including applications discussed in this paper. The compiler can estimate that the amount of time the second loop is executed is the number of times the second loop is executed (ndiag) times the average duration of a single iteration of the loop (w_3)[3]. The compiler also determines, based on the task graph analysis described in the previous section, that in order to predict the performance of the code, the actual values computed in array a are not necessary. Hence, it replaces that portion of the code with a call to advance the simulation clock by the estimated execution time of the loop (Example 1b) (For details about how the compiler calculates the analytic estimates, please see [3].) The first loop cannot be abstracted away since the value of ndiag computed by the loop body is needed to estimate the performance of the second loop. The compiler can also notice that a communication primitive precedes

[2] In real applications, the receive would be posted before the loop boundary calculation. We use this code here only to illustrate better the example.

[3] This is clearly a simplistic estimate, but it can be improved using existing compiler-driven modeling techniques for sequential code (e.g., [9]). The specific choice of this model is orthogonal to the optimizations we have proposed.

the second loop, and assumes that simulation process synchronization might occur during the communication call. The compiler then provides the simulator with lookahead information before the communication call is made (`set_lookahead(ndiag*w_3)`) and then resets the lookahead to 0 after the communication call. Note that this use of lookahead information introduces no *additional* approximations in the simulation, beyond the compiler-enhanced simulation described in our previous work.

Example 2:

```
for (j=1; j<N; j++) {
       mdiag[j] = mdiag[j-1];
       ndiag = ndiag + mdiag[j];}
MPI_Recv(&a, … …);
for (k=0; k<ndiag; k++ ) {
       a[k] = a[k-1] + …}
MPI_Send(&a, … …);
```

(a)

```
Set_lookahead(N*w_2);
MPI_Recv(… …)
reset_lookahead();
for (j=1; j<N; j++) {
       mdiag[j] = mdiag[j-1];
       ndiag = ndiag + mdiag[j]; }
set_lookahead(ndiag*w_3);
MPI_Recv(&a, … …);
reset_lookahead();
advance_clock(ndiag*w_3)
MPI_Send(&a, … …);
```

(b)

Even though we used analytic performance estimates only for code blocks that were abstracted away, for the purpose of lookahead we can also use compiler-generated performance estimates for portions of the code that need to be directly executed (such as loop boundary calculations). The key requirement is that these estimates must be lower-bounds for the actual execution time, so that the simulator does not violate causality. Consider Example 2. In 2a, there is a receive before the loop boundary calculation. Although we need to calculate the value of `ndiag` in the loop body, the compiler can let the simulator know that, when it is blocked in communications, it will not send a message with a timestamp smaller than the current simulation time plus the minimum message latency (L) plus the lookahead (`N*w_2`) (Example 2b). After the communication call is completed, the lookahead is reset to 0. This extension is not included in this paper because developing lower-bound performance estimates via compiler analysis requires substantial new research and is a subject for future work.

5 Results

MPI-Sim and the compiler optimized MPI-Sim have been previously validated [3, 6, 21] on a variety of applications such as NAS, ASCI and SPEC92 benchmarks on two hardware platforms: the IBM SP and the SGI Origin 2000. The original MPI-Sim predicted the performance of the applications within 7% of the measured system. The compiler-enhanced simulator, which used analytical models for portions of the computation, validated to within 17% of the measured system. The use of lookahead information in this work does not introduce any additional approximations over the latter. Therefore, we focus here on the improvement in performance of the null message protocol achieved by using the compiler-extracted lookahead information. All the following experiments were run on the IBM SP-2 at Lawrence Livermore, and used up to 128 processors on the machine.

5.1 Benchmarks

We use two synthetic benchmarks and two real world applications in our experiments. In the first synthetic code, the processes of the application are logically arranged in a ring topology. The processes execute several computation and communication iterations. First, the even processors perform a given amount of computation and then decide whether to send the results to the "right" or to the "left". The odd processes then enter the computation and communication phase. In each iteration, the receiving process does not know where the next message is coming from and therefore may need to request null messages from other simulation processes (assuming that a demand driven null message algorithm is used) to decide whether a given message is safe to process. The second application increases the dimension of the process topology to two. Again the processes are divided into two communicating groups. The first group computes its values and decides whether to send the values first horizontally and then vertically, or the other way around. Once again, the receiver needs to use null messages to identify safe messages. In both applications, the computation is abstracted away by the compiler and replaced with compiler generated analytical models. We will refer to the two synthetic benchmarks as 1D and 2D, respectively.

We also use two standard benchmarks, the ASCI Sweep3D code [25], a key benchmark used in the DOE ASCI program, and NAS SP, a fluid dynamics code from the NAS benchmark suite [7]. The compiler abstracted away most of the computation present in the codes. The most aggressive version of MPI-Sim [21] detects from the parameters to MPI calls that the null message protocol is not necessary for these two codes. We force the simulator to use the null message protocol in order to characterize the value of the lookahead in these codes and examine the potential benefits of the optimization. Many other applications such as NAS LU (which solves the same problem as NAS SP using a different algorithm) do require the null protocol, but are currently not supported by our dHPF compiler extensions for simulation.

In the following experiments, we compare the absolute performance improvement between MPI-Sim using no lookahead information (NOL) and MPI-Sim using the

lookaheads calculated by the compiler (LO). In both versions, the previous compiler optimization of abstracting away redundant computations is included so that we use a sophisticated and efficient simulation system as a baseline, and so that the two versions have identical accuracy.

5.2 Impact of Lookahead for Synthetic Benchmarks

The amount of computation in the 1D synthetic benchmark is related to the minimum message latency in the system (L=54µsec). The amount of computation (which is abstracted away) is taken from a normal distribution with three different means. Experiments are conducted for means of L/4, L/8 and L/20 and a standard deviation of 10% of the mean. In the first experiments, the number of host processors is the same as the number of target processors.

Table 1 shows the results for the 1D benchmark with the 3 different means. The rows are divided into three groups, representing the three means 2.7µsec, 6.75µsec and 13.5µsec. The number of target processors is varied from 4 to 100. The shaded areas represent a simulator's runtime greater than 2hrs (7,200sec), the maximum readily available machine time. Clearly, the simulator's ability to extract lookahead results in better performance. As the amount of lookahead increases (when the mean of the distribution for the abstracted computation is increased), the performance difference between NOL and LO also increases.

Mean = L/20= 2.7µsec	**Runtime in seconds**	
Procs	**NOL**	**LO**
4	862.428	26.9109
16	4262.14	127.972
64	>7200	500.037
100	>7200	853.927
Mean = L/8= 6.75µsec		
4	2565.98	63.2986
16	>7200	325.567
64	>7200	1344.2
100	>7200	2413.35
Mean = L/4= 13.5µsec		
4	5901.44	149.262
16	>7200	859.935
64	>7200	4873.38
100	>7200	>7200

Table 1: Runtime for MPI-Sim with and without lookahead, 1D benchmark with various means.

Mean = L/20= 2.7µsec	**Number of Protocol Messages**	
Procs	**NOL**	**LO**
4	4,445,148	111,258
16	22,226,220	556,800
64		2,351,223
100		3,713,391
Mean = L/8= 6.75µsec		
4	13,333,479	333,483
16		1,673,325
64		7,017,570
100		11,065,032
Mean = L/4= 13.5µsec		
4	30,613,356	740,817
16		4,558,590
64		24,971,247
100		

Table 2 Null message performance for the 1D synthetic benchmark.

The great differences in the runtimes of the simulators are directly related to the number of protocol messages needed to perform the simulation, as can be seen from Table 2.

Similarly, for the 2D benchmark (with a fixed amount of lookahead), MPI-Sim is able to use the lookahead to improve the simulator's performance by as much as two orders of magnitude for up to 49 target processors (host processors = target processors), as seen in Figure 1. For more than 49 processors, the simulator which had no lookahead information did not complete the simulation in the available time (7,200sec). The improvement in performance is directly related to the reduction in the number of necessary null messages, as shown in Figure 2.

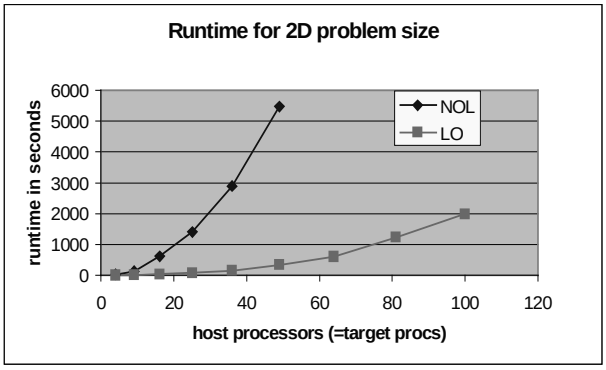

Figure 1: 2D Synthetic Problem, the number of host processors equals the number of target processors.

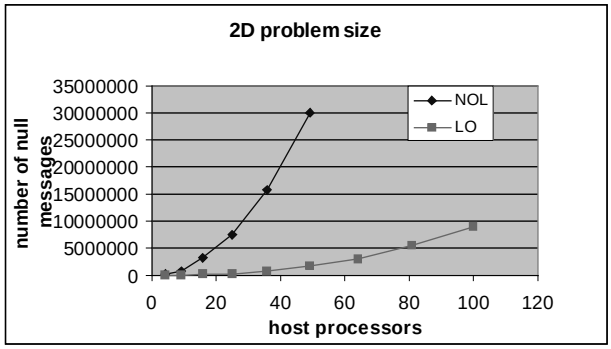

Figure 2: Null message performance for the 2D synthetic benchmark.

5.3 Impact of Lookahead for Sweep3D and SP

The previous results characterized the simulator's performance when the number of available host processors was equal to the number of simulated (target) processors. However, it is more often the case that the number of host processors is smaller than the number of target processors. Table 3 shows how the simulator performs when four host processors are used to simulate a system of 4, 16, 64 and 100 processors. The benchmark is the 1D code and the results are shown for normal distributions with means of 2.7µsec, 6.75µsec and 13.5µsec. The shaded areas represent again a simulator's runtime greater than 7,200sec. The NOL version is on the average 36 times slower than the LO version.

Mean	Target procs	NOL	LO
2.7µsec	4	850.096	24.3271
	16	891.166	27.26
	64	1123.99	29.1321
	100	1494.79	38.8056
6.75µsec	4	2577.2	66.2126
	16	2704.15	67.6685
	64	3363.05	84.3307
	100	4476.27	114.975
13.3µsec	4	5952.25	145.72
	16	6148.76	260.634
	64	>7200	457.22
	100	>7200	489.049

Table 3: Simulator's runtime for the 1D problem running on 4 host processors.

Although MPI-Sim with lookahead performed well for synthetic benchmarks, it is important to evaluate its performance on standard codes. We first look at the NAS SP benchmark, size A (the smallest size in the suite). Figure 3 shows the runtime of both NOL and LO when simulating NAS SP. In this case, the number of host processors is equal to the number of target processors. For the NAS SP benchmark, the lookahead we were able to extract allowed MPI-Sim to execute on the average 58.4% faster then the original simulator. Again, the improvement in performance is consistent with the decreased number of messages. The simulator without lookahead needs on average 72.5% more protocol messages (Figure 4).

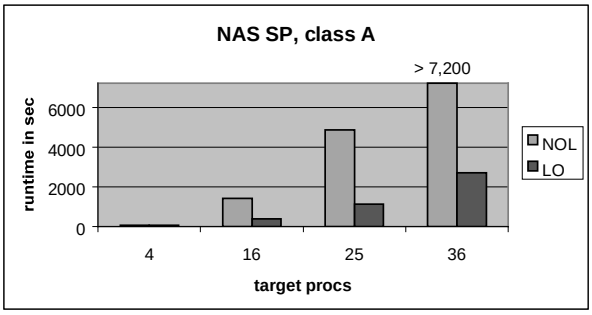

Figure 3: Runtime of MPI-Sim when simulating NAS SP class A.

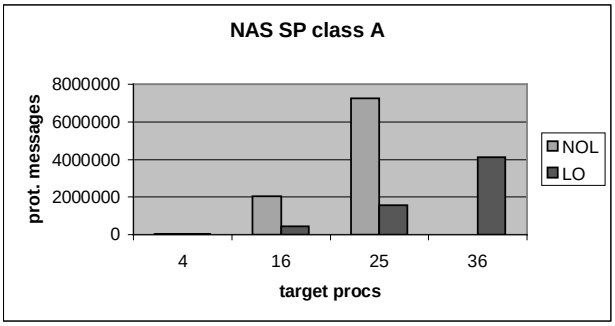

Figure 4: The number of protocol messages needed for synchronization in NAS SP.

The last set of experiments is for the Sweep3D benchmark. Figures 5 and 6 show the performance of MPI-Sim with a per processor fixed problem size of 4×4×255. The host system uses 16 processors to simulate up to 64 target processors. For this configuration, LO runs on the average 29.83% faster than the version without lookahead (Figure 5), which corresponds to the 25% reduction in the number of null messages (Figure 6).

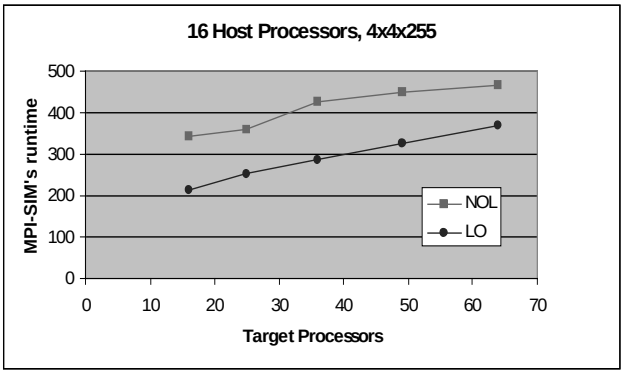

Figure 5: Runtime of MPI-Sim simulating Sweep3D, 4×4×255 per processor size, using 16 processors.

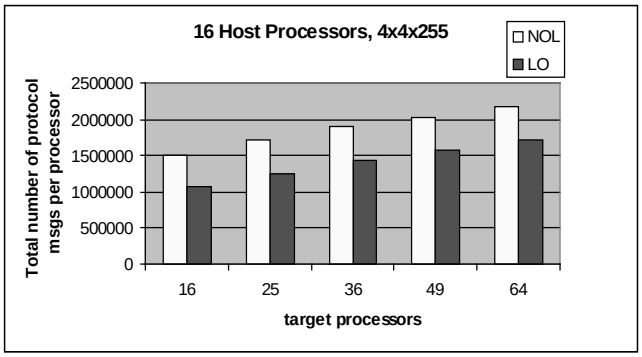

Figure 6: Number of null messages used by MPI-Sim simulating Sweep3D, 4×4×255 per processor problem size.

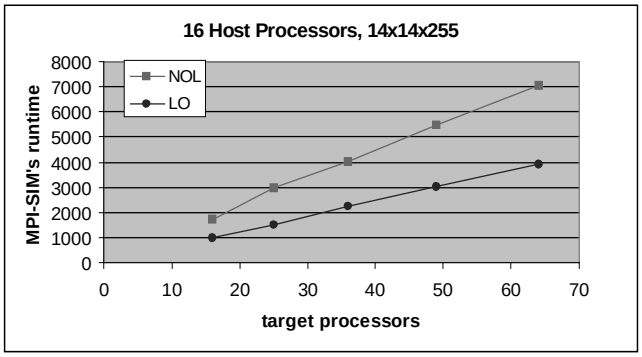

Figure 7: Runtime of MPI-Sim simulating Sweep3D, 14×14×255 per processor size, using 16 processors.

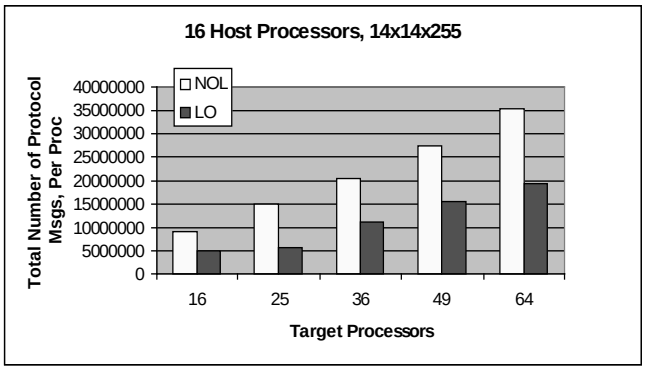

Figure 8: Number of null messages used by MPI-Sim simulating Sweep3D, 14×14×255 per processor size.

When the per-processor problem size is larger (14×14×255 per processor size), the granularity of the computation is greater (the lookahead is greater) and thus the benefit from lookahead is increased. Figure 7 shows the runtime of NOL and LO when using 16 host processors and simulating target systems from 16 to 64 processors. The performance improvement in the LO version is on the average 45% faster and the decrease of null messages is on the average 48%.

Our final two figures study the impact of lookahead information on the speedup of the simulator. We simulate Sweep3D for a fixed total problem size of 100^3 cells running on a fixed target system of 128 processors, and vary the number of host processors. The LO version of MPI-Sim performs on the average only 9.2% better than the NOL version, mainly because the granularity of computation per target processor is quite low for this case (Figure 9).

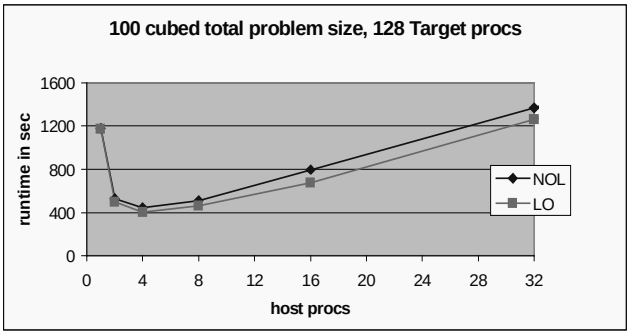

Figure 9: Runtime of MPI-Sim predicting the performance of 100^3 total problem size and a 128 processor target system.

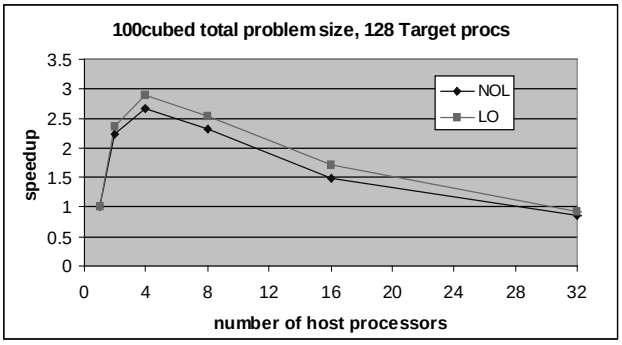

Figure 10: Speedup of MPI-Sim predicting the performance of 100^3 total problem size and a 128 processor target system.

Figure 10 shows that both versions achieve a maximum speedup of about 3 when using 4 host processors, and their speedup degrades beyond that point because the computation granularity *per host processor* is decreasing, thus decreasing the available lookahead. Since only a relatively small lookahead is available, the efficiency of the parallel simulation is relatively poor and the difference in performance of the LO and NOL versions is not as great as in the cases where computation granularity is greater.

6 Conclusions

In this paper, we have considered how compiler analysis can extract lookahead information to improve the performance of parallel simulation of message-passing parallel applications. Our prior system used compiler analysis to abstract away portions of the computational code and replace them with analytical performance estimates, yielding large benefits in simulator efficiency

(those benefits are obtained for either sequential or parallel simulation). In this paper, we showed that the compiler estimates can be used to provide lookahead information to the simulator, which can reduce the synchronization messages required for the synchronization protocol used in parallel simulation. We presented preliminary experiments using two synthetic applications and two widely used real world codes, which showed that using lookahead information may potentially lead to large reductions in the running time of the simulator.

We identify two key issues for future work. First, we must examine additional applications to evaluate to what extent these applications benefit from techniques to improve lookahead in parallel simulation. Second, and perhaps most exciting, we aim to explore how lookahead estimation techniques could be used for arbitrary computations, not just those whose results do not affect performance. This is important because such a technique could lead to significant additional improvements for a broad range of codes, especially irregular codes. The key challenge in this work would be to develop compiler techniques for reliable lower-bound performance estimates for computational fragments.

7 Acknowledgments

This work was supported by DARPA/ITO under Contract N66001-97-C-8533, by the NSF Next Generation Software program under contract number EIA-9975024, and by the NSF Operating Systems and Compilers program under grant number CCR-9988482. Thanks to Lawrence Livermore Laboratory for providing extensive computer time on the IBM SP/2.

References

[1] V. S. Adve and J. Mellor-Crummey, "Using Integer Sets for Data-Parallel Program Analysis and Optimization," *ACM SIGPLAN PLDI'98*, 1998.

[2] V. S. Adve and R. Sakellariou, "Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction," *LCPC'00*, Springer-Verlag LNCS, vol. 2017, 2001.

[3] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou, "Compiler-supported simulation of highly scalable parallel applications," *Journal of Parallel and Distributed Computing, Special Issue on Parallel Simulation (to appear)*. A preliminary version appeared at *ACM/IEEE SC99*, 1999.

[4] V. S. Adve and R. Sakellariou, "Application Representations for a Multi-Paradigm Performance Modeling Environment for Parallel Systems," *International Journal of High-Performance Computing Applications*, vol. 14, pp. 304-316, 2000.

[5] 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*, vol. 31, 1998.

[6] R. Bagrodia, E. Deelman, S. Docy, and T. Phan, "Performance Prediction of Large Parallel Applications using Parallel Simulations," *ACM SIGPLAN PPoPP*, 1999.

[7] D. Bailey, T. Harris, W. Shaphir, R. v. d. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA Ames Research Center NAS-95-090, 1995.

[8] E. A. Brewer, A. Colbrook, C. N. Dellarocas, and W. E. Weihl, "PROTEUS: a high-performance parallel-architecture simulator," *ACM Sigmetrics*, 1992.

[9] C. Cascaval, L. DeRose, D. Padua, and D. Reed, "Compile-Time Based Performance Prediction," *LCPC'99*, Springer-Verlag LNCS, vol. 1863, 2000.

[10] S. Chandrasekaran and M. D. Hill, "Optimistic simulation of parallel architectures using program executables," *PADS*, 1996.

[11] K. M. Chandy and J. Misra, "Distributed simulation: a case study in design and verification of distributed programs," *IEEE TSE*, vol. 5, pp. 440-52, 1979.

[12] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice parallel processing testbed," *ACM Sigmetrics*, 1988.

[13] H. Davis, S. R. Goldschmidt, and J. Hennessy, "Multiprocessor Simulation and Tracing using Tango," *Proceedings of ICPP'91*, pp. 99-107, 1991.

[14] P. Dickens, P. Heidelberger, and D. Nicol, "A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs," *PADS*, 1994.

[15] P. M. Dickens, P. Heidelberger, and D. M. Nicol, "Parallelized direct execution simulation of message-passing parallel programs," *IEEE TPDS*, vol. 7, pp. 1090-1105, 1996.

[16] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM TOPLAS*, vol. 12, pp. 26-60, 1990.

[17] U. Legedza and W. E. Weihl, "Reducing Synchronization Overhead in Parallel Simulation," *PADS'96*, pp. 86-95, 1996.

[18] 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," *Workhop on Performance Analysis and its Impact on Design (PAID)*, 1997.

[19] S. Prakash and R. Bagrodia, "An adaptive synchronization method for unpredictable communication patterns in data parallel programs," *IPPS*, 1995.

[20] S. Prakash and R. L. Bagrodia, "MPI-SIM: using parallel simulation to evaluate MPI programs," *IEEE WSC*, 1998.

[21] S. Prakash, E. Deelman, and R. Bagrodia, "Asynchronous Parallel Simulation of Parallel Programs," *IEEE TSE*, vol. 26, 2000.

[22] 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," *ACM Sigmetrics*, 1993.

[23] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, vol. 3, pp. 34-43, 1995.

[24] M. Rosenblum, E. Begnion, S. Devine, and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM TOMACS*, vol. 7, pp. 78-103, 1997

[25] "The ASCI Sweep3D Benchmark Code". http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html