Blekinge Institute of Technology
Research Report No 2002:04

Selecting Simulation Models when Predicting Parallel Program Behavior

av

Magnus Broberg, Lars Lundberg,
Håkan Grahn

Department of Software Engineering and Computer Science
Blekinge Institute of Technology

Selecting Simulation Models when Predicting Parallel Program Behaviour

Magnus Broberg, Lars Lundberg, and Håkan Grahn

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@bth.se
Phone: +46-(0)457 38 50 00, Fax: +46-(0)457 271 25

Abstract

The use of multiprocessors is an important way to increase the performance of a supercomputing program. This means that the program has to be parallelized to make use of the multiple processors. The parallelization is unfortunately not an easy task. Development tools supporting parallel programs are important. Further, it is the customer that decides the number of processors in the target machine, and as a result the developer has to make sure that the program runs efficiently on any number of processors.

Many simulation tools support the developer by simulating any number of processors and predict the performance based on a uni-processor execution trace. This popular technique gives reliable results in many cases. Based on our experience from developing such a tool, and studying other (commercial) tools, we have identified three basic simulation models. Due to the flexibility of general purpose programming languages and operating systems, like C/C++ and Sun Solaris, two of the models may cause deadlock in a deadlock-free program. Selecting the appropriate model is difficult, since we in this paper also show that the three models have significantly different accuracy when using real world programs. Based on the findings we present a practical scheme when to use the three models.

Key words: Simulation models, trace-driven simulation, development tool, parallel computing, performance prediction.

1. Introduction

The use of multiprocessors and distributed systems in supercomputing is very common. In order to take advantage of multiprocessors the programs must be parallelized. However, it is often hard to write efficient parallel programs for multiprocessors, since the software development tools for multiprocessors are immature compared to those for sequential programs. As multiprocessors are used more frequently, performance prediction and visualization of parallel programs become more important, since the developer needs support tools to write high performance programs.

The developer must make sure that the program runs efficiently on multiprocessors with different numbers of processors, since it is often the customer who decides the size of the multiprocessor on the basis of the actual performance requirements and the price/performance ratio. In some cases, developers want the program to scale-up beyond the number of processors available in current multiprocessors in order to meet future needs. There is thus no single target environ-

ment, and the development environment may not be the same as the target environment. Often the development environment is the (uni-processor) workstation on the developer's desk.

To predict and visualize the behaviour of parallel programs on the basis of a monitored uni-processor execution is a commonly used technique as shown in, e.g. [1, 2, 3, 4, 7, 9, 12, 13, 14, 15, 19, 20]. The basic idea is to graphically visualize the execution flow of the parallel program on a multiprocessor or distributed target system without actually having access to the target system. In order to do this the parallel execution is simulated based on recorded information collected during the monitored uni-processor execution. This technique has, for instance, been used in a commercial tool for message passing systems [12] and research tools for, e.g. FORTRAN [20], Ada [13], C/C++ [2, 3, 4], pC++ [14, 15, 19], PVM [1], MPI [7], and others [9].

For a number of years we have been working with a performance prediction and visualization tool called VPPB which supports the development of parallel C/C++ programs consisting of a number of Solaris threads and/or processes. We have found that the basic approach, i.e., simulating a multiprocessor execution based on information recorded during a uni-processor execution, is very useful and generally results in very accurate predictions. However, we have also identified an important problem with the approach, viz. that a simulation of a deadlock-free program may result in a deadlock.

We have, based on experiences from real parallel programs, identified three simulation models, called *Direct Model*, *Client-Server Model*, and *Strict Sequence Model*. Measurements on a set of benchmark applications show that Direct Model provides the best predictions on average; the second best predictions are provided by Client-Server Model, whereas Strict Sequence Model has the worst average predictions compared to a real multiprocessor execution. Strict Sequence Model will never result in a deadlock, provided that the parallel program is deadlock-free; deadlocks may occur in the Direct Model and the Client-Server Model. We also present an approach to automatically choose the appropriate simulation model. The evaluation with the benchmark applications shows that there is no significant difference between the simulation times of the three simulation models. The simulation times are significantly shorter than a real execution.

We go through the Direct, Client-Server, and Strict Sequence models in Sections 2, 3, and 4, respectively. In Section 5 we present our experimental methodology. The experimental results for a number of real world parallel programs predicted by the three models are found in Section 6. In Section 7 we present a simple method for selecting the appropriate model. Discussion of some related work is found in Section 8. Section 9 concludes the paper.

2. Direct Simulation Model

Consider the parallel program in Figure 1 (we assume synchronous synchronization, i.e. a process executing an Activate is blocked until the corresponding Wait_for has been executed). Work(*k*) denotes sequential processing for *k* time units.

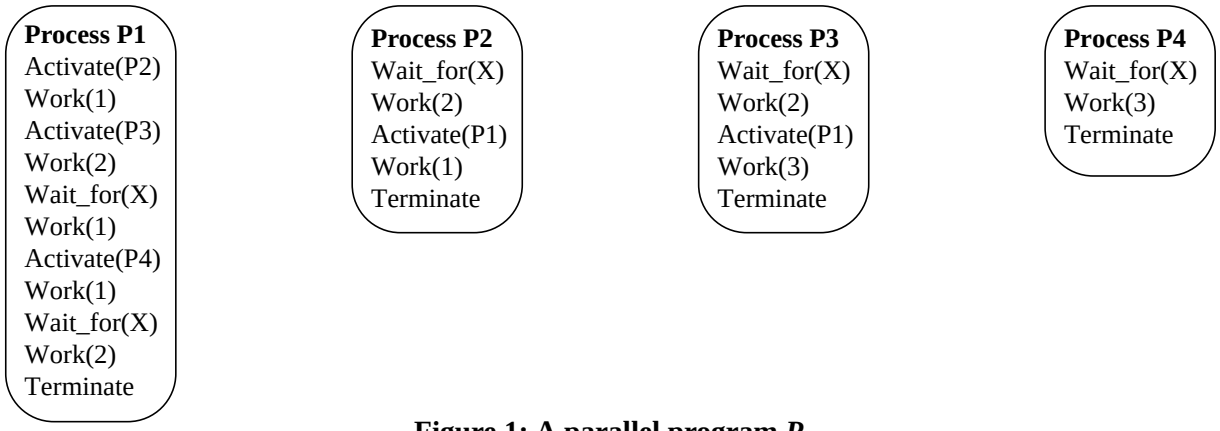

Figure 1: A parallel program *P*.

If this program is executed on a two-processor system where P1 and P2 are mapped to one processor and P3 and P4 are mapped to the other processor, and Prio(P1) > Prio(P2) > Prio(P3) > Prio(P4), we obtain the execution flow shown in Figure 2. Prio(X) is the priority of process X.

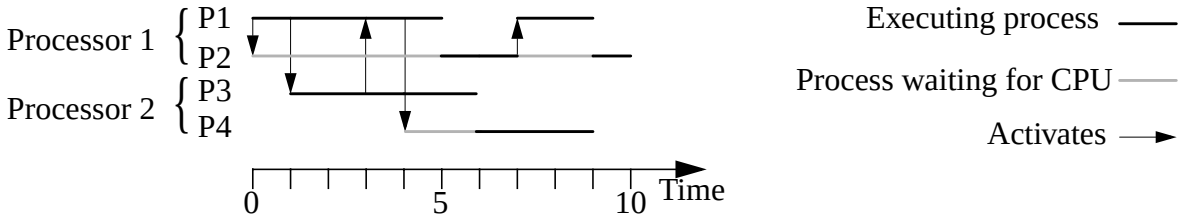

Figure 2: Execution flow of program *P* using two processors.

If we execute program *P* on a uni-processor and record all the synchronization events we get the trace of recorded information shown in Table 1. We have previously shown that it is possible to create the lists (one list per process) in Figure 1 based on the information in Table 1 [2, 3]. Based on the lists in Figure 1 it is trivial to simulate the behaviour shown in Figure 2, i.e. the behaviour using two processors and the priorities and binding of processes to processors discussed above. This means that we will be able to simulate a multiprocessor execution of program *P* based on a recorded uni-processor execution. We have previously validated the accuracy with very good results [2, 3].

Table 1: Information recorded during a monitored uni-processor execution of program *P*.

Time	Process	Event
0	P1	Create P2
1	P1	Create P3
3	P1	Wait for event X
5	P2	Send event X to P1
6	P1	Create P4
7	P1	Wait for event X
8	P2	Terminate
10	P3	Send event X to P1
12	P1	Terminate
15	P3	Terminate
18	P4	Terminate

The example above indicates no problems, and we have successfully used this Direct Model on a number of parallel programs. However, there are unfortunately situations where our Direct Model fails, e.g. the program *Q* in Figure 3.

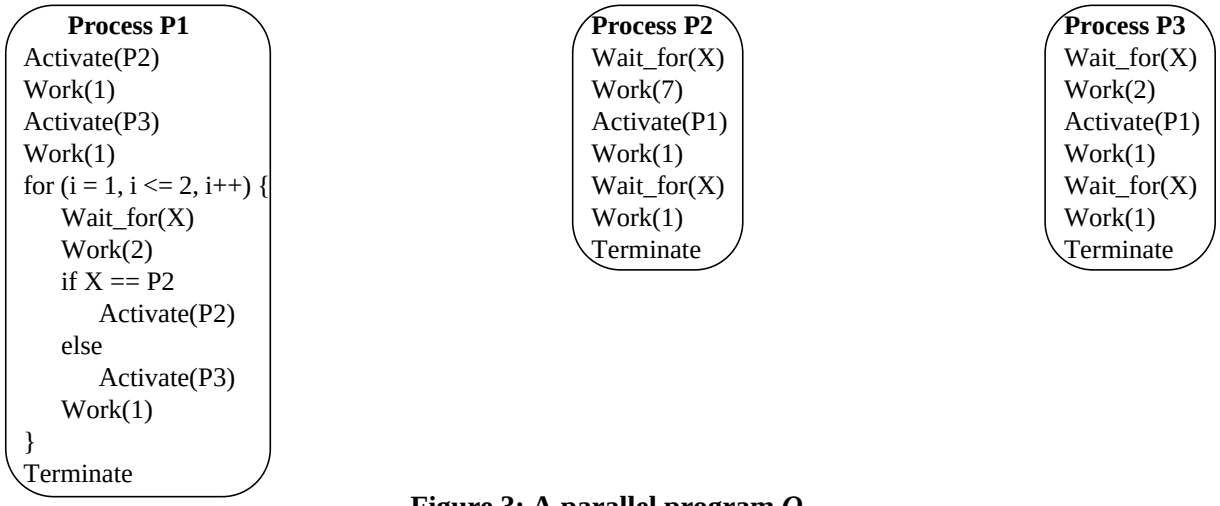

Figure 3: A parallel program *Q*.

Table 2 shows the information recorded during a uni-processor execution. Since we only record the synchronization events, we do not have any knowledge about program structures such as loops and if-statements. Consequently, if we, based on the recorded information in Table 2, try to obtain sequential lists (like the ones in Figure 1) we will end up with a program *Q'* shown in Figure 4.

Table 2: Information recorded during a monitored uni-processor execution of program *Q* given that Prio(P1) > Prio(P2) > Prio(P3).

Time	Process	Event
0	P1	Create P2
1	P1	Create P3
2	P1	Wait for event X
9	P2	Send event X to P1
11	P1	Send event X to P2
12	P2	Wait for event X
13	P1	Wait for event X
14	P2	Terminate
16	P3	Send event X to P1
18	P1	Send event X to P3
19	P3	Wait for event X
20	P1	Terminate
21	P3	Terminate

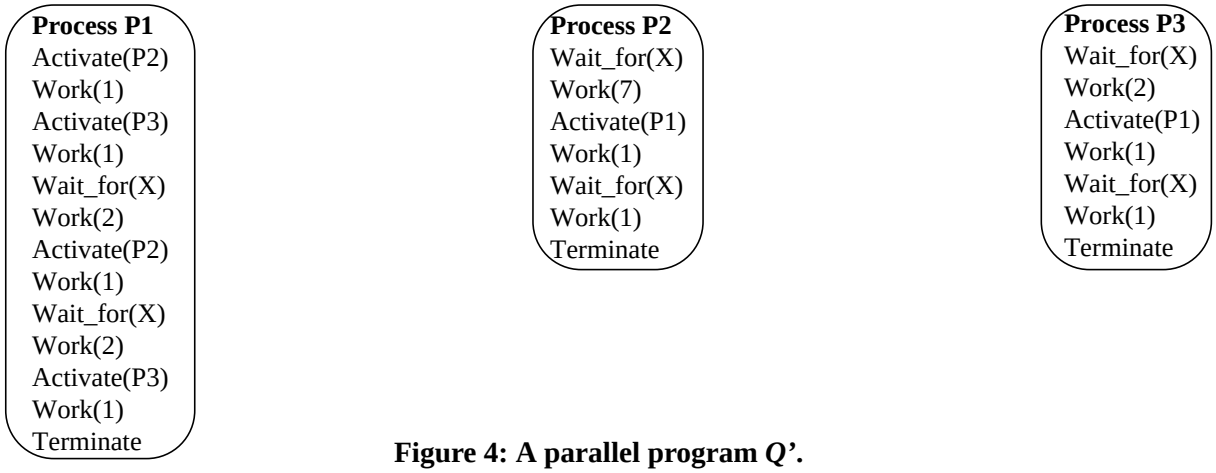

Figure 4: A parallel program *Q'*.

The recording in Table 2 is all the information we have about program *Q*. Based on this recording we obtain a program *Q'*, i.e., we (erroneously) assume that *Q'* is the program that generated the information in Table 2 (N.B. for program *P* we get the same program *P'* based on the trace in Table 1). If we simulate *Q'* for a three-processor system where each process is mapped to its own processor, we produce the execution flow shown in Figure 5.

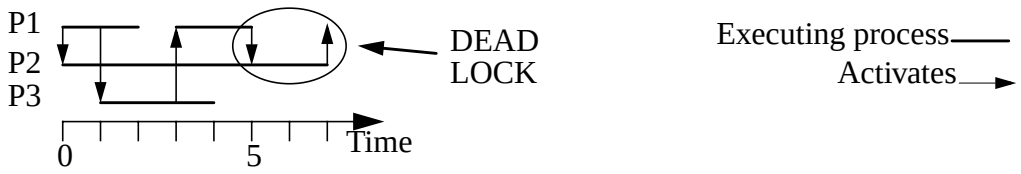

Figure 5: Execution flow of program *Q'* when using one processor per process.

As can be seen, there is a deadlock situation when P1 at time 5 (erroneously) tries to activate P2 instead of P3; P1 will block since P2 is not ready to receive a message. At time 7, P2 will send to P1, but P1 is already blocked, and thus P2 will block. This results in a deadlock, i.e., our simulation has produced a deadlock from a deadlock-free program (program *Q* will not deadlock on three processors). Consequently, we cannot use the Direct Model for program *Q*.

3. Client-Server Simulation Model

Program *Q* above made it obvious that we need to consider also other models than the Direct Model. Looking at program *Q* we see that in some cases we need to model also program loops; at least the loops that contain synchronizations.

By looking at a number of real parallel programs we have defined a simulation model called Client-Server Model (because it mimics the behaviour of a client-server application). A similar model has been used in [8, 9]. The difference between the Direct Model and the Client-Server Model is that each process is represented as a set of (short) lists in the Client-Server Model, whereas each process was represented as one list in the Direct Model. Each Wait_for(X) is also replaced by a Wait_for(PY), where PY can be P1, P2 or P3. The rule for replacing X with PY is simply that each Wait_for statement will now wait exclusively for the process that issued the *corresponding* Activate during the monitored uni-processor execution.

The number of lists used for representing a process is determined by the number of Wait_for statements that the process has executed during the monitored uni-processor execution. Each Wait_for statement defines the start of a new list. In the case with lists that start with a Wait_for for the same process PY the Wait_for statement is selected that corresponds to the Activate that during the monitored uni-processor execution triggered the Wait_for. The termination of a process is done when all the process's lists have been executed. Figure 6 shows how the trace in Table 2 would be transformed into a program *Q"* using the Client-Server Model.

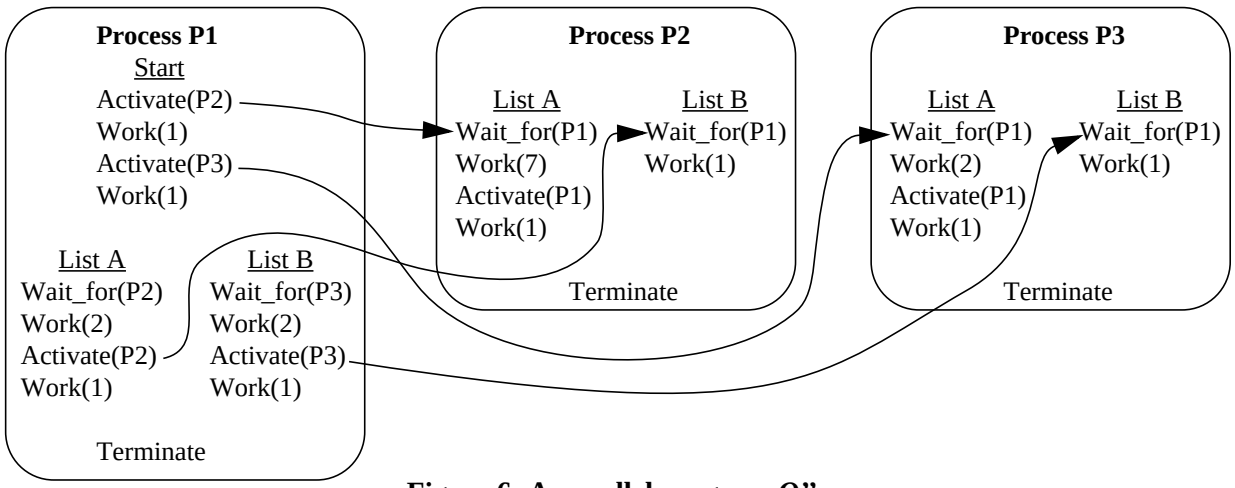

Figure 6: A parallel program *Q".*

Figure 6 shows that processes P1, P2 and P3 consist of two lists each and that process P1 is the start of the program and thus in addition has a start list that is executed first. Process P2 has two lists that both wait for process P1 to activate. In order to determine which list to start when processor P1 activates P2 we use the monitored uni-processor execution to find which instance of P1's Activate(P2) is related to which Wait_for(P1) in P2. The same reasoning is applied to the two Wait_for in P3. The relations are shown in Figure 6 with the arrows.

The rules for simulating are such that the lists may be executed out of order and a process that reaches a Wait_for statement may continue with a Wait_for statement in another list. However, each Wait_for is now waiting exclusively for one particular Activate.

When simulating program *Q"* using one processor for each process we get the execution flow shown in Figure 7. This is a correct simulation of program *Q*. One could note that a Client-Server Model of program *P* would have resulted in a completion time of 11, i.e. an erroneously pessimistic prediction.

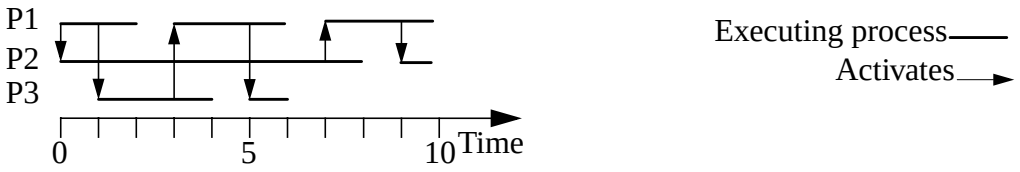

Figure 7: Execution flow of program *Q"* when using one processor per process.

Consequently, it seems that one should use the Client-Server Model when the Direct Model does not work. However, our problems do not stop here since there are situations where the Client-Server Model also results in deadlock for a deadlock-free program as in program *R* in Figure 8. In this figure the notion of X_i is used to represent an element in an array *X* at index *i*.

Process P1
Activate(P2)
Work(1)
Activate(P3)
Work(1)
Activate(P4)
Work(1)
for (i = 1, i <= 2, i++) {
  Wait_for(X_i)
  Work(2)
}
Activate(X_1)
Work(1)
Activate(X_2)
Work(1)
Wait_for(X_3)
Work(2)
Activate(X_3)
Work(1)
Terminate

Process P2
Wait_for(X)
Work(12)
Activate(P1)
Work(1)
Wait_for(X)
Work(1)
Terminate

Process P3
Wait_for(X)
Work(7)
Activate(P1)
Work(1)
Wait_for(X)
Work(1)
Terminate

Process P4
Wait_for(X)
Work(2)
Activate(P1)
Work(1)
Wait_for(X)
Work(1)
Terminate

Figure 8: A parallel program *R*.

Table 3 shows the information recorded during a uni-processor execution. In order not to make this presentation too detailed we skip the program *R'* (compare with programs *P'* and *Q'*) that would be produced in the Direct Model. If the reader does this exercise he/she will find that a multiprocessor simulation of *R'* using one process per processor will result in a deadlock. However, Figure 8 shows a deadlock-free program. Based on our experiences from the *Q* program above we will now try the Client-Server Model, thus obtaining a program *R"* (compare to *Q"*). Figure 9 shows program *R"*.

Table 3: Information recorded during a monitored uni-processor execution of Client-Server program *R* given that Prio(P1) > Prio(P2) > Prio(P3) > Prio(P4).

Time	Process	Event
0	P1	Create P2
1	P1	Create P3
2	P1	Create P4
3	P1	Wait for event X
15	P2	Send event X to P1
17	P1	Wait for event X
18	P2	Wait for event X
25	P3	Send event X to P1
27	P1	Send event X to P2
28	P1	Send event X to P3
29	P2	Terminate
30	P3	Wait for event X
31	P1	Wait for event X
32	P3	Terminate
34	P4	Send event X to P1
36	P1	Send event X to P4
37	P4	Wait for event X
38	P1	Terminate
39	P4	Terminate

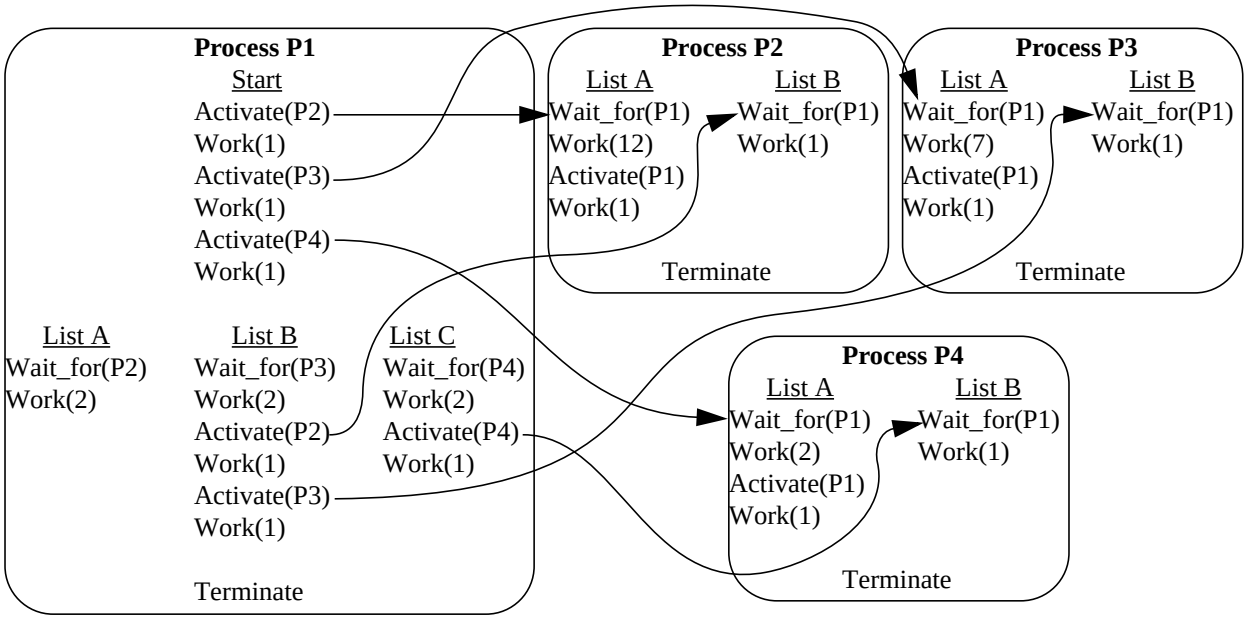

Figure 9: A parallel program *R”*.

If we simulate *R”* for a four-processor system where each process is mapped to a processor of its own we achieve the execution flow shown in Figure 10. As can be seen, the result is a deadlock situation. Process P1 sends to P2, which is not waiting for an event; P1 will thus block. P2 sends to P1, which is blocked; P2 will thus also block.

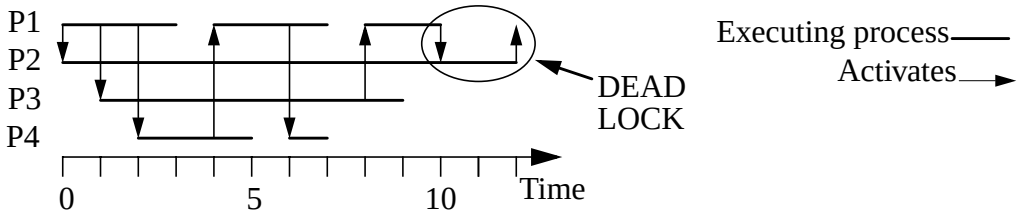

Figure 10: Execution flow of *R”*.

4. Strict Sequence Simulation Model

In order to handle programs that result in deadlock for the Direct Model and the Client-Server Model we have to resort to a pessimistic approach that we call Strict Sequence Model. The disadvantage with this model is that it can overestimate the multiprocessor completion time of some parallel programs. The advantage is that this model will never result in a deadlock, provided that the parallel program is deadlock-free. This model has been used in e.g. [12].

The difference between the Client-Server Model and Strict Sequence Model is that the (small) lists representing a process in the Client-Server Model are merged into one list (as in the Direct Model). The difference between the Strict Sequence Model and the Direct Model is how the Wait_for is handled. In the Strict Sequence Model the Wait_for waits for the Activate from the

process that during the monitored uni-processor execution activated the Wait_for, whereas it will accept an Activate from any process (on the same event of course) in the Direct Model. In the Strict Sequence Model the trace shown in Table 3 would thus result in the lists shown in Figure 11. We call the program obtained in the Strict Sequence Model for program *R'''*.

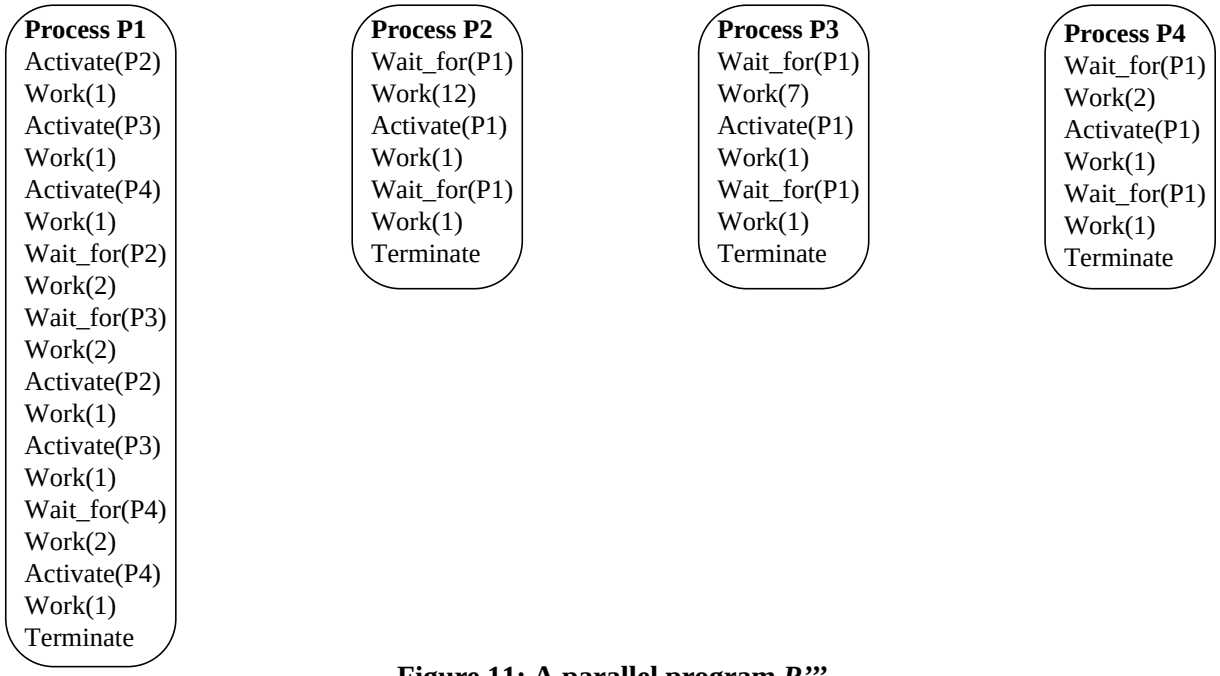

Figure 11: A parallel program *R'''*.

Figure 12 shows the execution flow for program *R'''*. There are obviously no deadlock problems using the Strict Sequence Model. The predicted execution time (21 time units) is, however, pessimistic compared to the real completion time for *R* using one processor per process (15 time units).

One could note that a Strict Sequence Model of program *P* would have resulted in a completion time of 12 and a Strict Sequence Model simulation of program *Q* would have resulted in 13 time units, i.e. too pessimistic predictions (the correct predictions can be seen in Figure 2 and Figure 7, respectively).

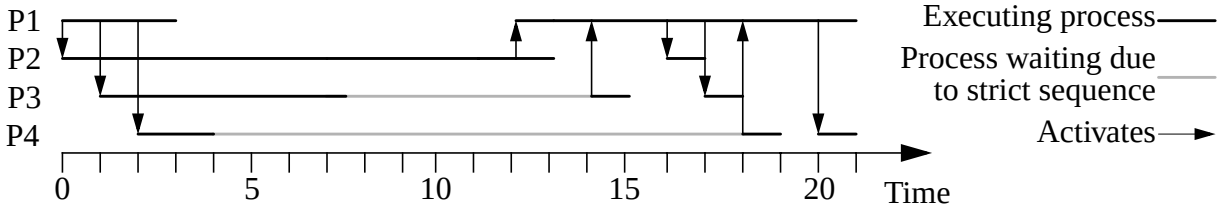

Figure 12: Execution flow of program *R'''*.

5. Experimental Methodology

The small examples in Section 2, 3, and 4 indicate that the predictions become more and more pessimistic as we go from Direct Model to Client-Server Model, and then to Strict Sequence Model. We would now like to verify this observation by applying the different models on a benchmark suite of supercomputing applications. We first present the tool, VPPB [2, 3, 4], that we have used in our experiments, and then we present the benchmark applications.

5.1. Overview of the VPPB Tool

The VPPB tool enables the developer to monitor the execution of a parallel program on a uni-processor workstation, and then predicts and visualizes the program behaviour on a simulated multiprocessor with an arbitrary number of processors. This is a previously used technique, e.g. in [12].

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB tool is shown in Figure 13. The developer writes the multi-threaded program (a) in Figure 13, compiles it and an executable binary file is obtained. After that, the program is executed on a uni-processor. When starting the monitored execution (b), the *Recorder* is automatically inserted *between* the program and the standard thread library. Each time the program uses the routines in the thread library, the call passes through the *Recorder* (c) which records information about the call, i.e., the identity of the calling thread, the name of the called routine, the time the call was made, and other parameters. The *Recorder* then calls the original routine in the thread library. When the execution of the program finishes all the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking of the application, making our approach very flexible.

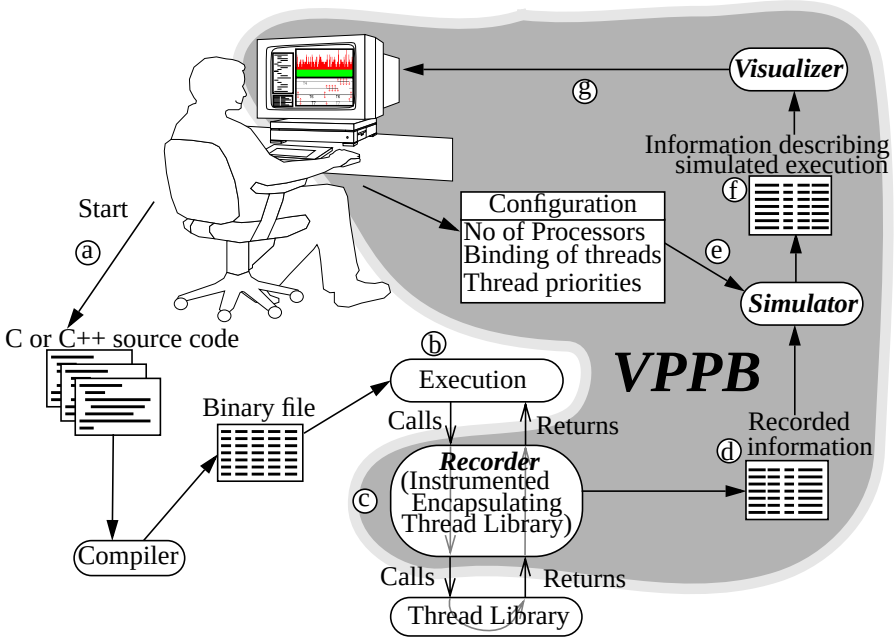

Figure 13: A schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 13. The simulator also takes the configuration (e) as input, such as the target number of processors, etc. The output from the simulator is information describing the predicted execution (f). In the simulator we have implemented the three simulation models earlier described, i.e. Direct Model, Client-Server Model, and Strict Sequence Model.

Using the *Visualizer* the predicted parallel execution of the program can be inspected (g). The Visualizer uses the simulated execution (f) as input. The main view of the (predicted) execution is a Gantt diagram. When visualizing a simulation, it is possible for the developer to use the mouse to click on a certain interesting event, get the source code displayed, and the line making the call that generated the event highlighted. With these facilities the developer may detect problems in the program and can modify the source code (a). Then the developer can re-run the execution to inspect the performance change.

The VPPB system is designed to work for C or C++ programs that uses the built-in thread package [10] and POSIX threads [6] on the Solaris 2.X operating system.

5.2. Benchmark Applications

In order to evaluate the correctness of each of the three simulation models, we have used a subset of the SPLASH-2 benchmark suite [22] running on the Solaris operating system with the Solaris thread package. In Table 4, the programs that we use are listed together with the data set sizes used. All SPLASH-2 applications are typical supercomputing applications. Since the SPLASH-2 applications are designed to create one thread per physical processor, one log file was generated for each processor set-up. Spinning locks were replaced with blocking locks as described in [5] since the trace tool does not handle spinning locks correctly.

Table 4: The parallel SPLASH-2 applications together with the data set sizes we used.

Application	Description	Data set size/Input data
Ocean (contiguous)	Simulate eddy currents in an ocean basin	514-by-514 grid
Water-Spatial	Molecular dynamics simulation, O(N) algorithm	512 molecules, 30 time steps
FFT	1-D $\sqrt{n}$ Six-step Fast Fourier Transform	4M points
Radix	Integer radix sort	16M keys, radix 1024
LU (contiguous)	Blocked LU-decomposition of a dense matrix	768x768 matrix, 16x16 blocks
Raytrace	Producing a raytraced picture	teapot
Barnes	Simulates interaction between a number of bodies in three dimensions	2048 bodies
Cholesky	Factors a sparse matrix into the product of a lower triangular matrix and its transpose	tk29.0
Radiosity	Producing a raytraced picture	Default, batch mode, en 0.1

The applications differ in the way the amount of work is distributed among the processors. Some of the applications have a rather static distribution of data and work among the processors, e.g. Water-Spatial, FFT, Radix, LU. For those applications, the data is distributed equally among

the processors and often optimized for high locality in the computation and the data access pattern.

Raytrace, Cholesky, and Radiosity use dynamic load balancing. For example, Radiosity has highly irregular computation structure and data access pattern, and uses distributed task queues.

Ocean and Barnes fall in between the other two categories. The data partitioning in Ocean is static but since a multigrid equation solver is used, the grid size is changed in order to make the algorithm converge faster. As a result, the amount of work between the processors may differ. Barnes uses a tree structure to distribute the work, which may result in different amount of work between processors.

The size of the trace files along with the number of events, i.e. the number of calls to the thread library, is found in Table 6. As the intention for the SPLASH-2 benchmarks is to have one thread on each processor, there are four traces per benchmark representing one, two, four and eight threads. These traces will be used for validation of the three simulation models in Section 6. It could also be noted that all information used for the Direct Model is equivalent to the information needed in two other models.

Table 5: Size of trace file in bytes for different number of threads on the different applications.

Application	1 thread		2 threads		4 threads		8 threads	
	Size	# Events	Size	# Events	Size	# Events	Size	# Events
cholesky	21565	360	828596	14267	1698857	29270	3169733	54628
fft	2284	28	4302	57	7738	115	14583	230
lu	18166	304	36518	609	72679	1218	145082	2439
radix	6576	102	9426	147	16002	259	34695	578
barnes	1858783	32029	1863003	32096	1870900	32229	1887328	32507
ocean	177088	3010	353232	5978	704384	11914	1406661	23785
raytrace	8886972	153213	8893773	153327	8894029	153331	8892917	153311
radiosity	19984005	344518	20247398	349054	18410866	317388	18975282	327116
water	50617	853	95697	1614	185133	3136	362524	6180

6. Experimental Results

In Table 6 the results of the simulations are shown as predicted speed-up. The results are compared with the real speed-up, using a Sun Ultra Enterprise 4000 with eight processors. The error in the table is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where |-x| = |x| = x, for all x > 0.

As can be seen in Table 6 there are some cases where the predictions are quite inaccurate, we have highlighted errors larger than 20%, which we consider to be a large error. The Strict Sequence Model has problems with four of the nine applications and the Client-Server Model has problems with three applications. The Direct Model handle all applications with a maximum 9% error as shown in Table 6 and thus has no problems with any application. In all cases with large misprediction by the Client-Server Model and Strict Sequence Model the predictions are underes-

Table 6: Speed-up and error of the SPLASH-2 benchmark using the three simulation models.

Appli-cation	2 processors				4 processors				8 processors			
	Real	Predicted speed-up (error %)			Real	Predicted speed-up (error %)			Real	Predicted speed-up (error %)		
		Direct Model	Client-Server	Strict Sequence		Direct Model	Client-Server	Strict Sequence		Direct Model	Client-Server	Strict Sequence
cholesky	1.62	1.59 (1.9)	1.09 (32.7)	1.48 (8.6)	2.31	2.21 (4.3)	1.81 (21.6)	1.77 (23.4)	2.85	2.80 (1.8)	2.55 (10.5)	2.01 (28.2)
fft	1.55	1.52 (1.9)	1.52 (1.9)	1.52 (1.9)	2.14	2.06 (3.7)	2.06 (3.7)	2.06 (3.7)	2.62	2.57 (1.9)	2.57 (1.9)	2.57 (1.9)
lu	1.79	1.82 (1.7)	1.82 (1.7)	1.81 (1.1)	3.15	3.08 (2.2)	3.11 (1.3)	2.96 (6.0)	4.82	4.71 (2.3)	4.82 (0.0)	4.06 (15.8)
radix	2.00	1.99 (0.5)	1.99 (0.5)	1.99 (0.5)	3.99	3.98 (0.3)	3.98 (0.3)	3.98 (0.3)	7.79	7.91 (1.5)	7.91 (1.5)	7.90 (1.4)
barnes	1.97	1.97 (0.0)	1.99 (1.0)	1.90 (3.6)	3.38	3.57 (5.6)	3.72 (10.1)	3.28 (3.0)	5.33	5.81 (9.0)	5.83 (9.4)	5.07 (4.9)
ocean	1.97	1.95 (1.0)	1.95 (1.0)	1.70 (13.7)	3.87	3.72 (3.9)	3.67 (5.2)	2.22 (42.6)	6.65	6.47 (2.7)	6.08 (8.6)	2.19 (67.1)
raytrace	1.72	1.66 (3.5)	1.00 (41.9)	1.00 (41.9)	2.50	2.42 (3.2)	1.01 (59.6)	1.00 (60.0)	3.28	3.24 (1.2)	1.04 (68.3)	1.01 (69.2)
radiosity	1.86	1.91 (2.7)	1.39 (25.3)	1.07 (42.5)	3.75	3.62 (3.5)	1.91 (49.1)	1.31 (63.9)	6.31	5.96 (5.5)	2.40 (62.0)	1.45 (77.0)
water	1.99	1.97 (1.0)	1.97 (1.0)	1.97 (1.0)	3.95	3.85 (2.5)	3.87 (2.0)	3.79 (4.1)	7.27	7.24 (0.4)	7.05 (3.0)	6.90 (5.1)

timating the speed-up, i.e. the execution time is pessimistically predicted. All the applications that give large errors have dynamic behaviour, and thus the large errors could be expected.

The distinction between the models is also shown in the mean values of the errors; Direct Model has 2.2% error, Client-Server Model has 15.7% and the Strict Sequence Model has 21.9%. Also the median value of the errors shows that the Direct Model has the smallest error and the Strict Sequence Model the largest error.

7. A Scheme to Automatically Choose the Appropriate Simulation Model

Based on the properties of the three simulation models we have constructed a scheme, which demonstrates when to use a particular simulation model. This scheme is shown in Figure 14.

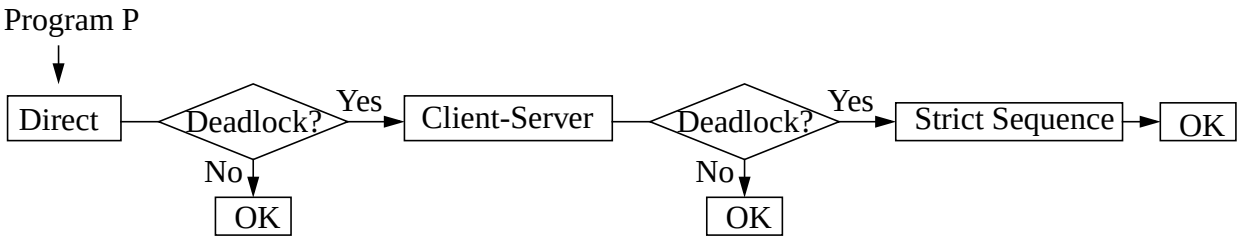

Figure 14: Practical scheme showing how to use the three simulation models.

The rationale behind the scheme is to start with the most direct and uncomplicated model: the Direct Model. We have also shown that this model results in the most reliable predictions for many parallel programs. However, if the Direct Model results in a deadlock, then we must use

another model. Since the Strict Sequence Model introduces a number of serialization constraints as well as produces the largest error, we first try the Client-Server Model. The Strict Sequence Model is only used as a last resort when the Client-Server Model also results in a deadlock. The Strict Sequence Model will never result in a deadlock. There is no guarantee that a simulation that has not deadlocked is a correct simulation. However, a deadlock indicates an incorrect simulation, provided that the application is deadlock-free.

One might wonder if there are some practical problems concerning that an application trace potentially might be simulated trice. Once again using the SPLASH-2 benchmarks we find that the simulation time for the benchmarks varies between 1 millisecond and 3.3 seconds on a 300MHz Sun Ultra 10. The times for all simulations performed in this study is found in Table 7. It should be noted that the times in Table 7 do not include the time required to read the trace file into memory and store the resulting file on disk. This is because, when using the scheme the trace file needs only to be read once and obviously the resulting file will also be written only once. As can be seen in Table 7 there is no significant difference between the execution times of the three simulation models. We can also conclude that the simulation time is very short. The simulation times are significantly *shorter* than a real execution, ranging from less than half of the execution time in worst case down to less than 1/50,000 in the best case. If we compare the simulation times in Table 7 with the size of the trace files (or the number of events) in Table 5 we can see that the simulation time is depending on the size of the trace file.

Table 7: Simulation time in seconds for the applications using different number of processors and simulation models on a 300MHz Sun Ultra 10.

Application	Direct Model				Client-Server Model				Strict Sequence Model			
	1 proc.	2 proc.	4 proc.	8 proc.	1 proc.	2 proc.	4 proc.	8 proc.	1 proc.	2 proc.	4 proc.	8 proc.
cholesky	0.001	0.057	0.167	0.511	0.002	0.083	0.220	0.587	0.001	0.066	0.191	0.453
fft	0.001	0.002	0.003	0.006	0.001	0.002	0.003	0.006	0.001	0.002	0.002	0.005
lu	0.001	0.003	0.010	0.036	0.002	0.004	0.010	0.027	0.001	0.003	0.009	0.021
radix	0.002	0.003	0.004	0.010	0.002	0.003	0.005	0.011	0.002	0.003	0.004	0.009
barnes	0.086	0.121	0.175	0.302	0.153	0.184	0.233	0.383	0.093	0.110	0.143	0.236
ocean	0.024	0.050	0.135	0.440	0.012	0.036	0.096	0.275	0.008	0.027	0.077	0.184
raytrace	0.418	0.559	0.793	1.331	0.679	0.768	0.873	1.140	0.467	0.507	0.610	0.841
radiosity	0.963	1.348	1.782	3.047	1.727	2.067	2.315	3.219	1.039	1.190	1.352	1.860
water	0.004	0.009	0.028	0.091	0.004	0.012	0.025	0.072	0.004	0.012	0.027	0.066
Average	0.167	0.239	0.344	0.642	0.287	0.351	0.420	0.636	0.180	0.213	0.268	0.408
	0.348				0.423				0.267			

None of the benchmarks in the SPLASH-2 benchmark suite caused any deadlock in any of the three simulation models. Thus, applying the scheme on the SPLASH-2 benchmark suite would be equivalent to applying the Direct Model since the other models would never be used.

8. Discussion and Related Work

Trace-driven simulations have been used for some time, particularly for testing various aspects of memory systems [21]. The main concern when assessing the validity of this technique is the tracing perturbation [21] and whether traces generated from several executions will yield different simulation results. These concerns have been studied and found to be insignificant [11], although they are still present. However, in this paper, the issue is somewhat more fundamental.

A commercial simulation tool called Dimemas [12] with the corresponding tracing tool MPIDtrace [17] is similar to VPPB. MPIDtrace produces event lists based on message-passing communication events and intervening calculation regions measured in execution time only. This is basically the same idea as in VPPB. MPI [16] has primitives that allow the application to receive messages from any process as well as a particular specified process. However, tests performed with MPI on Linux show that the MPIDtrace tool does not distinguish between the two variants and the resulting trace file shows as if the primitives were specifying a particular process, namely the process that sent the message during tracing. Another commercial tool, Vampirtrace [18], produces traces for Dimemas on both Sun Solaris and Linux. We tested Vampirtrace and found that it handles the receiving from any processor in the same way as MPIDtrace. The simulator, Dimemas, has to use the Strict Sequence Model (due to the nature of the input) in *all* situations. This will in some cases result in artificial and unnecessary restrictions resulting in pessimistic predictions. The work in [7] does only handle the point-to-point communication in MPI. However, the same approach as in Dimemas is taken, resulting in the Strict Sequence Model. The prediction tool for ADA [13] is also using the Strict Sequence Model. The PS [1] tool for PVM programs does not address the issues brought up in this paper at all.

The Client-Server Model is similar to the message-driven approach in [9]. A special-purpose programming language, called Dagger [8], is used to write programs that are entirely driven by messages, i.e. each process has a number of entry-points that are explicitly triggered by defined messages. This means that the out-of-order simulation that is performed in the Client-Server Model is explicitly expressed in the programming language. With the help of information generated by the Dagger compiler on a specific class of programs the message-driven approach will accurately handle the out-of-order simulations.

Some programming languages use a simple hierarchical fork-join structure. The parallel FORTRAN simulator in [20] represent such a situation. With these restrictions the Strict Sequence Model can successfully be used. Similar restrictions apply to the pC++ simulators in [14, 15, 19].

The scheme in Section 7 is based on the assumption that the user of the models (or a tool) is not aware of what kind of program is being simulated, which is the case for general Sun Solaris programs written in C or C++. Obviously, if the user knows that the program is written in a particular model, then that model should be used from the start for that particular program. An example could be if we *always* use the receive primitives in MPI which specify the sender, we know that the Strict Sequence Model is appropriate. It can also be possible to have certain parts of a program modelled using one simulation model, and another part of the program using another simulation model, e.g. by annotating the source code (with, e.g. annotations for entry-points, indicating the Client-Server Model) and these annotations are reflected in the trace. Another possibility is to start with the Direct Model for thread synchronization (which is usually cooperative [5]) and with

the Client-Server Model for communication between processes (which are usually competitive [5]).

9. Conclusions

We have identified three simulation models that can be used when simulating the multiprocessor performance of parallel supercomputing programs. The basic technique, i.e. simulating multiprocessor performance based on a monitored uni-processor execution, has been used by a number of commercial tools and research prototypes [1, 2, 3, 4, 7, 12, 13, 14, 15, 19, 20]. Each of the three simulation models have been used previously. However, the choice of model in these previous cases seems to have been more or less arbitrary and the authors of these papers do not seem to have been aware of any other model than the one that they have selected for their particular study or tool.

We have in this paper, evaluated the three models on a set of real world parallel programs using a multiprocessor with eight processors. There is no significant difference between the simulation times of the three simulation models. The simulation times are significantly shorter than a real execution. The evaluation showed that the predictions based on the Direct Model were the most accurate ones, the Client-Server Model had the second best predictions, and the Strict Sequence Model resulted in the worst predictions. However, we have also demonstrated that the Direct Model may (erroneously) result in a deadlock when simulating a deadlock-free program. We have also shown that some of the programs that result in a deadlock using the Direct Model are handled correctly by the Client-Server Model. Some programs may, however, result in (erroneous) deadlocks for both the Direct Model and the Client-Server Model. The Strict Sequence Model will never result in a deadlock since the traced event order (which obviously did not cause any deadlock) is kept in the simulation. The deadlock problem associated with the Direct Model and Client-Server Model has, to the best of our knowledge, not been identified before.

Consequently, there is a trade-off between the accuracy in the predictions (where the Direct Model is the best and the Strict Sequence Model the worst), and the capability of avoiding erroneous deadlocks (where the Strict Sequence Model is the best). Based on our findings, we have defined a simple deadlock driven scheme for deciding when to use which model. Our scheme provides the best (average) predictions possible, while avoiding erroneous deadlocks.

Acknowledgements

This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

References

[1] R. Aversa, A. Mazzeo, N. Mazzocca, and U. Villano, “Heterogeneous system performance prediction and analysis using PS,” IEEE Concurrency, Vol. 6, No. 3, pp. 20-29, 1998.

[2] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs,” in Proc. 12th International Parallel Processing Symposium, pp. 770-776, 1998.

[3] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads,” in Proc. 13th International Parallel Processing Symposium, pp. 407-413, 1999.

[4] M. Broberg, L. Lundberg, and H. Grahn, “Performance Optimization using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors,” Journal of Parallel and Distributed Computing, vol. 61, no. 1, pp. 115-136, 2001.

[5] A. Burns, “Concurrent Programming,” Addison-Wesley, ISBN 0-201-54417-2, 1993.

[6] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, 1997.

[7] E. Demaine, “Evaluating the Performance of Parallel Programs in a Pseudo-Parallel MPI Environment,” Proc. 10th International Symposium on High Performance Computing Systems (HPCS’96), CD-ROM, 1996.

[8] A. Gürsoy and L. Kale, “Dagger: combining the benefits of synchronous and asynchronous communication styles,” in Proc. International Parallel Processing Symposium, pp. 590 - 596, 1994.

[9] A. Gürsoy and L. Kale, “Simulating message-driven programs,” in Proc. International Conference on Parallel Processing, pp. 223 - 230, 1996.

[10] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996.

[11] E. Koldinger, S. Eggers, and H. Levy, “On the Validity of Trace-Driven Simulation for Multiprocessors,” in Proc. 18th International Symposium on Computer Architecture, pp. 244-253, 1991.

[12] J. Labarta and S. Girona, “Sensitivity of Performance Prediction of Message Passing Programs,” in Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 620-626, 1999.

[13] L. Lundberg, “Predicting the Speedup of Parallel Ada Programs,” in Proc. Ada Europe International Conference, pp. 257-274, 1992.

[14] A. Malony and K. Shanmugam, “Performance extrapolation of parallel programs,” Proc. International Conference on Parallel Processing, Vol. 2, pp. 117-120, 1995.

[15] B. Mohr, A. Maloney, and K. Shanmugam, “Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs,” Proc. Joint Conference PERFORMANCE TOOLS ‘95 and MMB ‘95, pp. 254-268, 1995.

[16] MPICH, “Manual pages MPICH 1.2.1”, http://www-unix.mcs.anl.gov/mpi/www/, 2001.

[17] MPIDtrace, http://www.cepba.upc.es/dimemas/, 2001.

[18] Pallas Inc., “Vampirtrace,” http://www.pallas.de/pages/vampirt.htm, 2001.

[19] S. Sarukkai and D. Gannon, “SIEVE: A Performance Debugging Environment for Parallel Programs,” Journal of Parallel and Distributed Computing, 18, pp 147-168, 1993.

[20] K. So, A. Bolmarcich, F. Darema, and V. Norton, “A Speedup Analyzer for Parallel Programs”, in Proc. International Conference on Parallel Processing, pp. 654-662, 1987.

[21] R. Uhlig and T. Mudge, “Trace-driven Memory Simulation: A Survey,” in Performance Evaluation: Origins and Directions, Lecture Notes in Computer Science, Springer-Verlag, pp. 97-139, 1999.

[22] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in Proc. 22nd Annual International Symposium on Computer Architecture, pp. 24-36, 1995.

Selecting Simulation Models when Predicting Parallel Program Behavior
av Magnus Broberg, Lars Lundberg, Håkan Grahn

ISSN 1103-1581
ISRN BTH-RES--04/02--SE

Printed by Kaserntryckeriet AB, Karlskrona 2002