Blekinge Institute of Technology
Doctorial Dissertation Series No 03/02
ISSN 1650-2159
ISBN 91-7295-009-9

Performance Prediction and Improvement Techniques for Parallel Programs in Multiprocessors

Magnus Broberg

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Sweden

Publisher: Blekinge Institute of Technology
Printed by Kaserntryckeriet, Karlskrona, Sweden 2002
ISSN 1650-2159
ISBN 91-7295-009-9

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible"
- Charles Babbage (1791-1871)

This thesis has been submitted to the Faculty of Technology, Blekinge Institute of Technology, in partial fulfilment of the requirements for the Degree of Doctor of Philosophy in Computer Systems Engineering.

Contact Information:

Magnus Broberg
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
S-372 25 Ronneby
SWEDEN
Email: Magnus.Broberg@bth.se
URL: http://www.ipd.bth.se/magnusb

Abstract

The performance of a computer system is important. One way of improving performance is to use multiprocessors with several processors that can work in parallel. Where multiprocessors are used, the programs must also be parallel in order to achieve high performance. However, it is not always easy to write parallel programs for multiprocessors; program developers need support in this area. Such support includes, for example, information regarding how well the parallel program scales-up when the number of processors increases and identification of performance bottlenecks; ideally, the result should be presented graphically. Bottlenecks arise both as a result of parallelization as well as traditional (sequential) code. Further, the developer may need to predict performance on other systems than the one used for development, since the development environment often is the (uni-processor) workstation on the developer's desk. One way of increasing the performance may be to bind threads on processors statically. Finding the optimal allocation is NP-hard and it is necessary to resort to heuristic algorithms. When heuristic algorithms are used we do not know how near/far we are from the optimal allocation. Finding a bound for the program's completion time shows what should be achievable using a heuristic algorithm.

In this thesis, I present techniques how to simulate a multiprocessor execution of a parallel program based on a monitored execution on a uni-processor. The result of the (simulated) multiprocessor execution is graphically presented in order to give feedback to the developer. The techniques can be used for heuristic algorithms to find an allocation of threads to processors. Further, I show an algorithm that identifies the critical path of the parallel program on a multiprocessor, thereby identifying the segments that are worthwhile optimizing. I also show how to calculate a tight bound on the minimal completion time for the optimal allocation of threads to processors. Finally, I discuss the implications of the choice of simulation model. The techniques and algorithms described have been manifested in a prototype tool which I have used to perform empirical studies. The tool has been validated using a real multiprocessor.

Acknowledgments

This work has been partially funded by the national Swedish Real-Time Systems research initiative ARTES, supported by the Swedish Foundation for Strategic Research.

To list all those involved in some way in my research would be a tedious task, and I would probably miss someone out anyway. However, I would still like to mention a few key people.

Firstly, I would like to thank my supervisors Professor Lars Lundberg and Dr. Håkan Grahn for their splendid guidance and stimulating discussions over the past few years while I have been working on my thesis. Secondly, I would like to thank my previous examiner (during the first two years of my research) Professor Per Stenström, Chalmers University of Technology. I would also like to thank Charlie Svahnberg and Daniel Häggander for listening to and criticizing some of my ideas and Henrik Persson for improving the speed of the Simulator. I would also like to express my thanks to Kamilla Klonowska for her contributions to one the of the papers included in this thesis.

Most of all, I would like to thank Maria for supporting me in such an excellent and unselfish way during the hard times, as well as for sharing all the joy in the good times. I would never have been able to carry this out without you!

List of Papers

The present thesis is based on the following papers. References to the latter will be made using the roman numbers applied to the particular paper.

[I] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs,” in Proceedings of the 12th International Parallel Processing Symposium, pp. 770-776, 1998.

[II] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads,” in Proceedings of the 13th International Parallel Processing Symposium, pp. 407-413, 1999.

[III] Magnus Broberg, “Performance Tuning of Multithreaded Applications for Multiprocessors by Cross-Simulation,” in Proceedings of the ISCA 13th International Conference on Parallel and Distributed Computing Systems, pp. 281-288, 2000.

[IV] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “An Allocation Strategy Using Shadow-processors and Simulation Technique,” in Proceedings of the ISCA 14th International Conference on Parallel and Distributed Computing Systems, pp. 387-394, 2001.

[V] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “A Tool for Binding Threads to Processors,” in Proceedings of the Euro-Par 2001, pp. 57-61, 2001.

[VI] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “Performance Optimization using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors,” in Journal of Parallel and Distributed Computing, Vol. 61, No. 1, pp. 115-136, 2001.

[VII] Magnus Broberg, Lars Lundberg, and Kamilla Klonowska, “A Method for Bounding the Minimal Completion Time in Multiprocessors,” Research Report 2002:03, (submitted for publication).

[VIII] Magnus Broberg, Lars Lundberg, and Håkan Grahn, “Selecting Simulation Models when Predicting Parallel Program Behaviour,” Research Report 2002:04, (submitted for publication).

Contents

Performance Prediction and Improvement Techniques for Parallel Programs in Multiprocessors

1 Introduction

The performance of a computer system is crucial. It seems as if the need for increasing computing power is never satisfied. If a uni-processor computer does not give the required performance, i.e., the work cannot be processed fast enough, an obvious solution is to let several processors work in parallel. Parallel processing is, therefore, an important way to increase the performance of demanding applications. Multiprocessors have several processors able to work in parallel. Applications developed for multiprocessors are thus likely to have high performance requirements.

If an application is to take advantage of a multiprocessor it must contain parallelism. However, it is not always easy to write parallel applications for multiprocessors. Such applications are often very complex and the software development process for multiprocessors is not as highly developed as the development process for sequential applications [55]. In this thesis I consider parallel programs consisting of several processes/threads. The processes/threads need to synchronize and communicate if they are to complete a common task.

Application developers thus need a great deal of support if they are to write parallel high-performance applications. Such support includes information regarding how well the parallel application scales-up when the number of processors increases as well as how to identify performance bottlenecks. The scale-up is often known as “scalability in machine size”, defined in [55] as “how well the performance [of an application] will improve with additional processors.” For example, if we have twice as many processors, we should ideally attain twice as high performance. Performance in this thesis relates to the execution time of the application [50]. The higher performance, the shorter execution time required.

A number of programming paradigms and multiprocessor platforms have emerged, such as the thread concept in Solaris 2.X. These have resulted in multi-threaded commercial applications (and non-commercial too) written for multi-

processors. These kinds of parallel applications are generally not developed for a specific number of processors, i.e., there is no single target environment. The reason for this is that the same application may be executed on a multiprocessor with few processors as well as on a machine with a large number of processors. Generally, it is the customer who decides the size of the multiprocessor; this is determined by the actual performance required and the price/performance ratio. The customer also wants to be able to buy additional processors and execute the same application with a higher performance. Developers must therefore make sure that the application runs efficiently on different numbers of processors. In some cases, developers want the multithreaded application to scale-up beyond the number of processors available in a multiprocessor today in order to meet future demands; the development environment may thus not be the same as the target environment. Often the development environment is the (uni-processor) workstation on the developer's desk.

The use of standards has many benefits. One such benefit is that a program using some standard, e.g. POSIX threads [20], can easily be ported (ideally only recompiled) to some other operating system such as Linux. However, the program may perform differently from one operating system to another. It is thus beneficial for the developer to be able to predict the performance on operating systems other than the one used for development.

Static binding of threads on processors means that the thread is only allowed to execute on the named processor. This contrasts with dynamic allocation where the thread may migrate from one processor to another during execution. It has previously been shown that allocation (static binding) of threads on processors can increase the performance [80]. This is particularly true in the case of NUMA (Non Uniform Memory Architecture) machines. Thus, finding good allocations is important. Basically the problem is NP-hard [40] and it is necessary to resort to heuristic algorithms. Performance prediction for a given allocation is useful finding a good allocation. When using heuristic algorithms we do not know how close (or far) we are from the optimal allocation. If we are able to find a tight bound on the minimal completion time for the optimal allocation, we can compare the bound with the completion time from the heuristic algorithm.

All of the above arguments lead us to the conclusion that the developer of a multithreaded application needs support when determining the performance, identifying performance bottlenecks, and identifying the critical path. Since many multithreaded applications are developed on a uni-processor workstation, it is desirable to provide the necessary support in this environment.

This thesis work explores techniques and methods for predicting and improving the performance of parallel programs. While writing this thesis I have developed a prototype research tool to implement and test my ideas in practice. Further, I have performed empirical studies on benchmark programs, synthetic programs, and skeleton versions of industrial programs. Finally, the predictions of the prototype research tool have been validated in a real multiprocessor environment.

The techniques used in this thesis are based on the assumption that the program is deterministic, i.e., the program has the same behavior regardless of the number of processors and/or scheduling policy. A relaxation of this assumption is found in paper VIII.

2 Research Results

All my research results are demonstrated in a practical tool, called VPPB (Visualization of Parallel Program Behavior). The techniques developed during my research are implemented in this tool. The VPPB tool enables the developer to monitor the execution of a parallel program on a uni-processor workstation, and then predict and visualize the program behavior on a simulated multiprocessor with an arbitrary number of processors.

The workflow when using the tool is shown in Figure 1. The developer writes the application and compiles it (a). The application is then executed on a uniprocessor, e.g., the developer's workstation (b). During execution data about the behavior of the application are collected (c). The recorded information is used to simulate a multiprocessor execution of the application. The Simulator is given specific parameters about the target hardware as well as the scheduling policy (d). The result of the simulation is displayed graphically (e) to the developer. Analysis methods can be applied to the simulated execution (f). The extended critical path (see Section 2.3) can be calculated and shown in the graphs as well as in a separate table. The minimal completion time for the application when using statically bound threads (see Section 2.3) can be calculated and visualized. Automatic finding of an allocation of the threads to the processors is carried out in (g) (see Section 2.2). The developer can now investigate the (simulated) execution and perform additional experiments by changing the simulation parameters (d) again. In this way, the developer can run several experiments using the same recorded information. If the developer needs to modify the application source code, she/he may start at (a) once again to verify the impact of the modifications.

The VPPB system is designed to work for C or C++ programs using the Solaris built-in thread package [20] and/or POSIX threads [8]. The target simulation environment is Solaris 2 and Linux 2.

This thesis comprises eight papers. The papers can be categorized according to how they contribute to the overall method as shown in Figure 2. Papers I, II, and III show the development of the most fundamental parts of the tool; this is particularly true of paper I, where the initial versions of the Recorder, Simulator and Visualizer are to be found. In paper II, extensions are made in order to handle I/O and multiple LWPs by monitoring the kernel of the operating system. In paper III, I evaluate the applicability of the tool in another target operating environment for performance prediction, i.e. Linux.

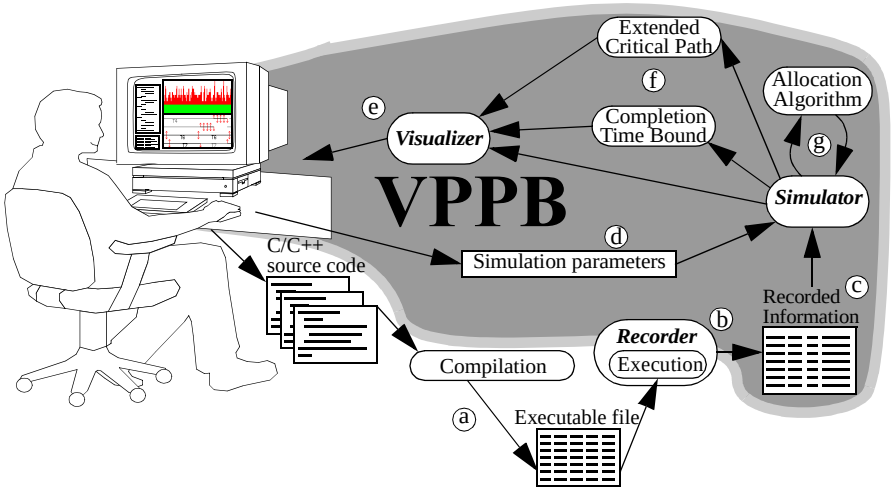


Figure 1: A schematic workflow of the techniques in this thesis.

Papers IV and V present how the VPPB tool can be used to evaluate the difference (in terms of execution time of the application) between allocations while searching using heuristic allocation algorithms.

Papers VI and VII present new analysis methods that have been developed during my research: a multiprocessors extension of the critical path analysis and an analytical method for bounding the minimal completion time for statically allocated processes. The latter method is a complement to the heuristic algorithms in papers IV and V.

Finally, in paper VIII I look more deeply at the simulation approach for performance prediction used in the tool and evaluate a number of alternatives.

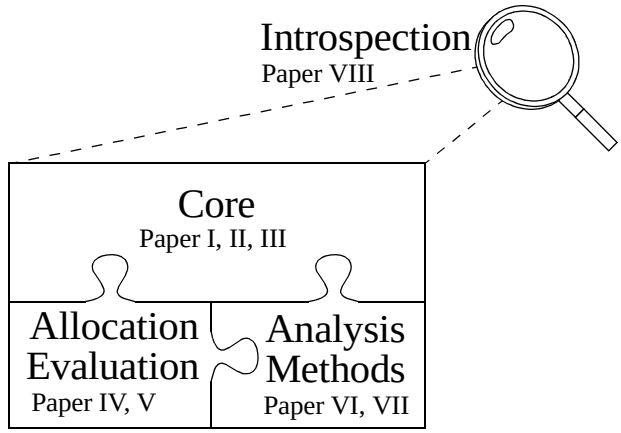


Figure 2: One way of looking at how the papers are related to each other.

2.1 The Core

The core papers contain the most fundamental parts of the proposed techniques. The papers included in this category are I, II, and III. The developer of a parallel program usually works on a uni-processor workstation. If there is no access to a multiprocessor, the developer has limited opportunities to estimate the real performance of the parallel program. One approach is to predict the performance by simulation. Further, if the performance is not sufficiently good, the developer needs feedback in order to improve the program. The core papers investigate how to show the performance of the parallel program for the developer and also give feedback to the developer for her/him to improve the program, where necessary. It should be possible to do all this on a uni-processor workstation.

I show in papers I, II, and III that it is possible to give the developer support on a uni-processor workstation, i.e. to estimate the performance of the parallel program and give feedback. I have defined three entities: the Recorder, the Simulator, and the Visualizer as shown in Figure 3. In the Recorder I have defined which activities on the uni-processor workstation to be monitored; the function calls for two Solaris libraries and the activities in the scheduler. I have also defined which parameters in the function calls need to be monitored. In the Simulator I have defined how to process the recorded data and how to use it as simulation input in order to mimic a multiprocessor execution. My simulation approach is based on one list of events per thread. I have made it possible for the developer to set the number of (simulated) processors regardless of the number of threads. I have shown how to visualize graphically the (simulated) multiprocessor execution in the Visualizer.

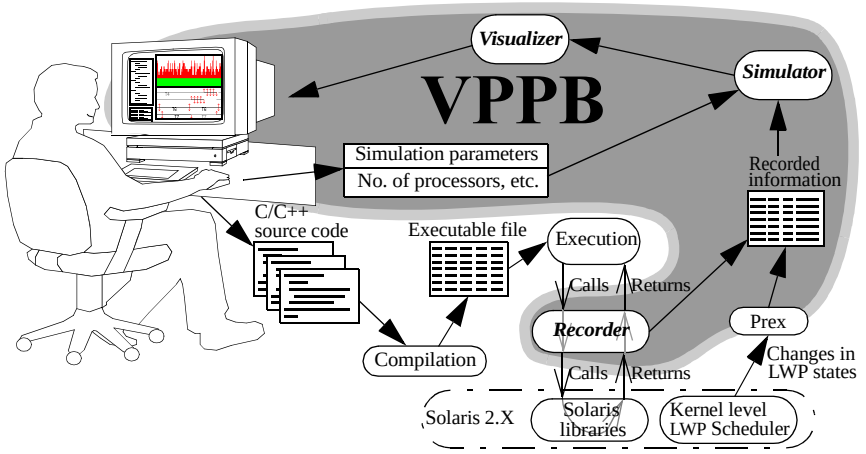


Figure 3: The workflow in the core papers.

Paper I contributes to the above by demonstrating how the Recorder, Simulator and Visualizer, operates on the Solaris operating system using user-level threads. Paper II demonstrates support for handling I/O and for managing kernel threads in the parallel program by tracing events and activities in the kernel. Finally, paper III enables the developer to simulate for another target environment (Linux) than that of the host environment (Solaris).

I have validated my performance predictions by using the SPLASH-2 benchmark suite and the skeleton of a large telecommunication application. The validation for the Solaris operating system proved accurate with an error less than 10% (on average 1.6%, as reported in paper III). On Linux the error was greater (on average 5.8%, as reported in paper III) since the monitoring operating system differs from the target operating system; we still consider this a viable approach, however. The tracing overhead was less than 2% of the total execution time for most of the applications in SPLASH-2. The simulation time is as shown in paper VIII less than half (down to 1/50,000) compared to when the application is executed on a uni-processor machine.

2.2 Allocation Evaluation

It is known that static allocation (binding) of threads on processors can increase performance and that static allocation is the only option in some systems. However, finding the optimal allocation is NP-hard and we must thus resort to heuristic algorithms. Papers IV and V investigate if the techniques developed in the core papers can be used to find good allocations. The user should not need to specify anything other than the simulation parameters in the core papers, e.g. the number of processors in the target multiprocessor. The research question in this part is thus: Is it possible to automate the search for an allocation without specifying (by hand) anything about the program?

I have found that it is possible to automate the search for an allocation without specifying (by hand) anything about the program. The two algorithms developed for papers IV and V illustrate this conclusion. In both cases the algorithms are compared to a traditional binpacking algorithm using several thousand automatically generated programs. Both algorithms give approximately 40% shorter execution times than the traditional binpacking algorithm in our parallel test programs. The basic approach is to use the recorded information from the core papers, and then use this information to find a good static allocation. The simulator then predicts performance; the allocation algorithm can use this information for improving the allocation. This is illustrated in Figure 4.

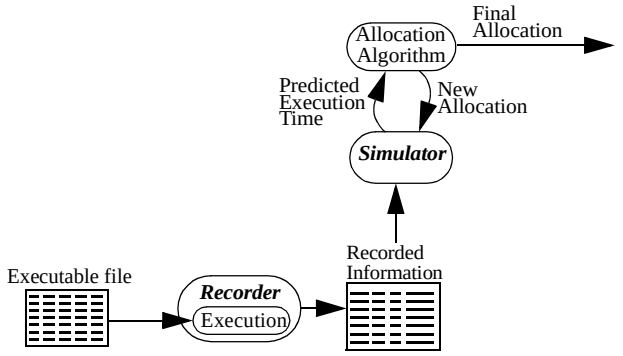


Figure 4: The use of the Recorder and the Simulator for finding allocations.

The algorithm in paper IV allocates the threads on the processors one by one; the threads which are not yet allocated are placed on one shadow processor each. When there are fewer processors than threads, the algorithm uses more processors than those available on the target machine while searching thanks to the shadow processors; as a result the algorithm cannot run on the target machine and without simulation techniques it would be impossible to use the algorithm. The algorithm in paper V uses a start allocation (the allocation from the binpacking algorithm) and then permutes the allocation in the search for a better allocation. The two algorithms have different properties; the first produces a result after a fixed time (which is given by a formula based on the number of processes and processors), whereas the second algorithm gives better and better allocations incrementally. This means that the second allocation algorithm can be stopped by the user or by some other criteria at any time and still give a result. Further, in its basic form the second algorithm never stops iterating; I have, however, defined a stop criterion. The latter is based on the observation that at some point further iterations will not be worthwhile. Using the stop criterion, I propose a new (third) algorithm that reduces the risk of getting stuck in a local maxima.

The allocation techniques used in papers IV and V use the recording and simulation facilities in the core (Section 2.1) without modifying of the core in any way. Papers IV and V thus give the core added value.

2.3 Analysis Methods

Two analysis methods belong to this category, and they are presented in papers VI and VII. Both methods are implemented using the core described in Section 2.1.

Performance tuning of sequential programs can be carried out through optimization of the code. The major issue here is to identify which segments of the program it is worthwhile to optimize. In sequential programs it is rather trivial and several commercial profiling tools already exist. However, in general it is not trivial in a parallel program. In paper VI the focus is on how to find the segments that are worthwhile optimizing for any combination of number of threads and number of processors. Showing the developer the relative impact of optimizing a segment is also important since this gives her/him the opportunity to prioritize the optimization efforts on the most worthwhile segments of the program. Another question addressed in this paper is how to present the segments for the developer. This is especially important since the developer may be used to tools for sequential programs; finding similar ways of representing the results is thus of great importance.

I have defined an algorithm in paper VI that identifies which parts are worthwhile optimizing. The algorithm identifies the (extended) critical path through the parallel program. The extension is used to express that the critical path is found for any combination of number of threads and number of processors and not only in special cases (see related work Section 3.4). I define the extended critical path as all the executed code segments of a program that when reduced with a small value will reduce the total execution time on a given number of processors. Different segments of the program have a different impact on the total execution time for the program. Although it is perhaps non-intuitive, I show that some segments may actually increase the execution time if optimized. Each segment is given a weight representing the optimization impact on the total execution time. All segments in the (extended) critical path are shown as thicker lines in the visualization graph. To further aid the developer, the segments are related to the functions in the program and a list of the functions (weighted by the weight of the segments) is shown in the same way as a traditional profiling tool illustrating the (weighted) relative time spent for each function in the (extended) critical path. The last-mentioned feature required the introduction of additional recording capabilities in the Recorder.

The second paper in this category, paper VII, addresses one drawback with heuristic algorithms for finding an allocation of the threads to processors, namely that one does not know if the heuristic algorithm has found an allocation that is close to or far from the optimal allocation. One way to solve this is to compute analytically a bound of the minimal execution time of the program on a given machine. If the execution time of a certain allocation is above this bound, then we know that it is possible to find a better allocation and that it is worth-

while to continue the heuristic search. To be useful the bound must be tight. In this paper our focus is on finding an analytical way to find such a tight bound.

The contribution in paper VII is an analytical method for bounding the minimal execution time of the program on a given machine. The method is based on parameters describing the parallel program; n is the number of processes/threads, V the parallel profiling vector showing to what fraction of the completion time a certain number of threads are executing simultaneously given that there is one processor for each thread, and z describes how often per time unit the program synchronizes. All these parameters can be obtained automatically from the recorded information in the core of the tool. The target hardware is defined in terms of k - the number of processors and t - the synchronization latency between processors. The analytical method is optimal in the sense that for at least some program P (defined by n, z, V) is equal to the bound.

My main contributions in paper VII are sections 1, 4.4 (except 4.4.3), 6, 7, 8 and 9. However, I have also made a number of smaller contributions in other sections, e.g., Section 2 and the formulas in Section 5.

Both analysis methods (papers VI and VII) need a multiprocessor execution of the parallel program with one processor per process/thread; the simulator from the core (see Section 2.1) thus plays an important role.

2.4 Introspection

The model used in the simulator was the same in papers I to VII. The fundamental issues in paper VIII deal with the possibility of other models for trace-based simulation. Are there other models? Do they work properly? How accurate are the predictions they produce? Which one should we use in which situation? Are there any trade-offs when selecting simulation model? All these questions need an answer.

In paper VIII I define three trace-based simulation models. The three simulation models are called the Direct Simulation Model, the Client-Server Simulation Model, and the Strict Sequence Simulation Model. The first model is the one used in the core (see Section 2.1). The last model is used in, for example, [34] and [44]. The remaining model is similar to the one used in [46]. I have analyzed the models and found that both the Direct and Client-Server Simulation Models may introduce deadlocks in an otherwise deadlock-free program. 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. However, the Client-Server Simulation Model and the Strict Sequence Simula-

tion Model may introduce artificial constraints into the program that may affect the accuracy of the predictions (too pessimistic). Quantifying the errors in each model shows that the Direct Simulation Model clearly yields the lowest average error. The Strict Sequence Simulation Model had the largest average error. Consequently, there is a trade-off between the accuracy in the predictions and the capability of avoiding erroneous deadlocks. I have defined a simple deadlock driven scheme for deciding when to use which model. The scheme provides the best (average) predictions possible, while avoiding erroneous deadlocks.

3 Related Work

We start by presenting work related to papers I, II, and III since these papers are the core of this thesis; visualization tools are examined in Section 3.1 followed by simulation tools in Section 3.2. Work related to thread allocation (papers IV and V) is presented in Section 3.3, followed by the results of investigations related to profilers and critical path analysis (paper VI) in Section 3.4. Work related to the analytical bounds of minimal execution time (paper VII) is found in Section 3.5, and finally, related work for paper VIII is discussed in Section 3.6.

3.1 Visualization of an Execution

The idea of displaying a representation of the execution of a parallel application graphically is not new. As can be seen in this section there are many tools. These tools visualize the execution of a parallel application on a multiprocessor. Table 1 presents some visualization tools with their fundamental characteristics in the columns. It should be noted that these characteristics do not necessarily reflect the full capabilities of each tool. The characteristics may also be subjective since not all references directly reveal the desired information for inclusion in the table;, in such cases analysis of screenshots, etc. has been used. The different columns in Table 1 are:

- Message-passing - if the tool supports the message-passing programming model or not. Message-passing often etails the use of PVM or MPI.
- Shared Memory - if the tool supports the shared memory programming model, (usually in terms of threads). Note that message-passing on a shared memory machine does not count.
- Post-mortem - if the tool shows the results after the termination of the application investigated.
- On-the-fly - if the tool shows the results during the execution of the application investigated.
- Re-play - if the tool re-plays (like a tape-recorder) the collected information in animated views (like a movie). Only the animated views are applicable for on-the-fly tools.
- Time-line - if the tool supports views where one of the axes is used to represent time. Where re-play is available, the time axis may be expanded as time progresses during re-play.
- Statistics - if the tool supports histograms, kiviats, calculations for standard deviation, or non-trivial metrics, etc.

Table 1: Some characteristics of the visualization tools.

Name / Reference	Message-passing	Shared memory	Post-mortem	On-the-fly	Re-play	Time-line	Stati-stics
ATEMPT [66, 67]	Yes	No	Yes	No	No	Yes	No
Carnival [91]	Yes	Yes	Yes	No	No	No	Yes
Conspector [110]	Yes	Yes	No	Yes	No	Yes	No
Falcon [45]	No	Yes	Yes	Yes	Yes	Yes	No
Gthread [142, 143]	No	Yes	Yes	No	Yes	Yes	No
Guideview [131]	No	Yes	Yes	No	No	No	Yes
IPS-2 [57, 94]	Yes	Yes	Yes	No	Yes	Yes	No
Jumpshot/(N)upshot [141]	Yes	No	Yes	No	No	Yes	No
Medea [21, 22]	Yes	No	Yes	No	No	No	Yes
Moviola [74]	Yes	Yes	Yes	No	No	Yes	No
MPP apprentice [27, 133]	Yes	No	Yes	No	No	No	Yes
Lei&Zhang [77]	Yes	No	Yes	No	No	Yes	No
OSE Illuminator [103]	Yes	Yes	No	Yes	No	Yes	No
Pablo [112, 113]	Yes	No	Yes	No	Yes	Yes	Yes
Parade [65]	Yes	Yes	Yes	No	Yes	No	Yes
Paradyn [93, 95]	Yes	Yes	No	Yes	No	Yes	Yes
ParaGraph [49]	Yes	No	Yes	No	Yes	Yes	Yes
ParaMap [58]	Yes	No	Yes	No	No	Yes	No
ParaVision [102]	Yes	No	Yes	No	Yes	Yes	No
PARvis [100]	Yes	No	Yes	No	Yes	Yes	Yes
PAT [28]	Yes	Yes	Yes	No	No	No	Yes
PATOP [18]	Yes	No	No	Yes	No	Yes	No
PMA [136, 137]	Yes	No	Yes	No	No	Yes	No
P-RIO [79]	Yes	No	Yes	No	Yes	Yes	No
Projections [121]	Yes	Yes	Yes	No	No	Yes	No
PROVE [61, 62]	Yes	No	Yes	No	No	Yes	No
PV [54]	Yes	Yes	Yes	Yes	Yes	No	No
PVaniM [24]	Yes	No	Yes	Yes	Yes	Yes	No
Quartz [2]	No	Yes	Yes	No	No	No	Yes
SCALEA [129]	Yes	No	Yes	No	No	No	No
SvPablo [30, 31]	Yes	Yes	No	Yes	No	No	Yes
TATOO [12]	Yes	No	Yes	Yes	Yes	Yes	No
TAU [90, 96, 120]	Yes	Yes	Yes	Yes	No	No	Yes
ThreadMon [23]	No	Yes	No	Yes	No	No	Yes
Tmon [59]	No	Yes	No	Yes	No	No	Yes
TNFView [127]	No	Yes	Yes	No	No	Yes	Yes
TraceView [87]	Yes	Yes	Yes	No	No	Yes	No
VAMPIR [68]	Yes	No	Yes	No	Yes	Yes	No
VGV [17, 51, 63]	Yes	Yes	Yes	No	Yes	Yes	No
Virtue [116]	Yes	No	Yes	Yes	No	Yes	No
VISTOP [134]	Yes	No	Yes	No	Yes	No	No
vt [56]	Yes	No	Yes	Yes	Yes	No	No
Xab [8]	Yes	No	Yes	Yes	Yes	No	No
XMPI [72, 73]	Yes	No	Yes	Yes	No	Yes	Yes
XPVM [41]	Yes	No	Yes	Yes	No	Yes	No

The number of publications each year indicates a constant interest in visualization tools. Most tools handle message-passing (84%), whereas only 47% handle shared-memory. Of the tools that support shared-memory, 67% also support message-passing. It would seem that a majority of the tools originate from the message-passing community. This is illustrated in [17], where VGV (a combined VAMPIR and Guideview tool) is presented as (the message-passing) VAMPIR extended with (the shared memory) Guideview, not vice versa.

There are a lot of different views that has been developed in the visualization tools, e.g. ParaGraph contains more than 25 different views alone. Most sophisticated visualization, including 3-dimensional virtual reality views, 3-dimensional gloves, and speech recognition, can be found in Virtue.

3.2 Simulation Tools

There are a lot of tools for simulating multiprocessors. Some of these are found in Table 2. The tools are categorized based on the fundamental technique employed (described later in this section) and whether the tools present metrics only, or any graphical visualization of the execution flow. Absence of underlining means support for message-passing, single underlining means support for shared-memory, and double underlining means support for both. The lower right corner of Table 2 is described in more detail than the others since this is where VPPB belongs.

Table 2: Categorization of some simulation tools.

Technique	Simulation and metrics	Simulation and visualization
Direct-execution	DirectRSIM [35], EXCIT [69], MPI-SIM [1, 108], ParSim [117, 132], PERFSIM [128], SPASM [122], Tango [29], WWT-II [98, 99]	PROTEUS [15], Schumann [118]
Interpretation	PEPSIM [48], RSIM [35, 53, 105], SimICS [85, 86]	SimOS/Rivet [13, 114]
Static analysis	Balasundaram et. al. [7], PEPSY [9], PerFor [3, 4], P^3T+ [38, 39]	EDPEPPS [32, 33], PACE [101, 104]
Modeling	$CHIP^3S$ [106], PAMELA [42, 43], PerPreT [14], PIE [76, 119], Uysal et. al. [130]	-
Trace-based	Clement&Quinn [26], Dagger [46, 47], Eom&Hollingsworth [36, 37], Intrepid [25], Mendes [92], SPAN [123]	AIMS/MK [11, 138, 139], Demaine [34], PARAVER/DIMEMAS [44, 70, 71, 109], PS [5, 6], SIEVE [115], SPEEDY [89, 97], **VPPB**

3.2.1 Direct-Execution

This simulation technique is based on executing the application (for real) on the machine which carries out the simulation. At each instrumentation point (normally at calls to message-passing libraries and/or load/store instructions) the control is passed to the simulator, which simulates the action and passes back the control. The application must be synchronized frequently in order to prevent one thread/process executing ahead of the others too much. As a minimum, synchronization is carried out when the control is transferred to the simulator. A static analysis of the application (assembler/machine instruction level) is used to determine the number of clock cycles required for executing each basic block on the target host. Counters are inserted into the code to count the number of times each basic block has been executed since the last control transfer. Variants (e.g. PROTEUS, Tango, WWT-II) are to be found where a clock is updated at each basic block with the time for the execution of the basic block. DirectRSIM differs by tracing the execution path of the program during direct-execution; when the simulator is called it quickly interprets the path (instruction-by-instruction) and estimates the execution time.

3.2.2 Interpretation

This technique simulates the program instruction-by-instruction and is also known as “execution-driven simulation” or “program-driven simulation”. It is possible to model the whole hardware in detail with pipeline, memory, caches, buses etc. This technique has been used for a long time in commercial simulators developed by hardware manufacturers. Simulation of forthcoming hardware enables suitable software to be developed before the hardware even exists. Since each instruction is interpreted, focus is on the slow-down factor, ranging from 25-75 (SimICS) up to 500-50,000 (SimOS/Rivet) as compared to running the program directly on the simulation host. PEPSIM interprets pseudo code. The number of (real) instructions for implementing a pseudo instruction is used to estimate the execution time of the program.

3.2.3 Static Analysis

This is a compiler-based approach. By letting the compiler analyze the code, performance is predicted. This enables compilers to determine how to distribute arrays for parallel calculations (PEPSY, Clement&Quinn). PEPSY is targeted for one machine only; the execution time for an instruction is known. The other tools need measurements on the target machine. PerFor measures the execution

time for one iteration of each loop on the target machine. PACE translates the program into a CHIP^{3}S-model and executes sub-components of the program on the target machine to determine the execution times. Clement&Quinn probe programs (called training sets) and execute these on the target machine to determine certain metrics. EDPEPPS uses the source code (C with PVM) in order to build a queuing network graph. The execution time for a computational block is estimated by counting the number of different kinds of high-level language instructions (e.g. float addition). P^{3}T+ is coupled with the VFC (Vienna Fortran Compiling System). The sequential program (which is the input for VFC) is, however, monitored (by SCALEA [129]) and executed in order to determine branch probabilities, etc.

3.2.4 Modeling

These tools are normally targeted to estimate the performance before any code exists. CHIP^{3}S uses a programming language to express the model; the sole use of the language is performance prediction. PAMLEA uses a language built of binary semaphores (modeling shared resources), delays (modeling computation), and which parts of the program to be executed in parallel. The model of the target machine is expressed in the language. The two models are compiled and optimized into a single model. PIE uses predefined communication models that can be combined to capture the basic structure of the program. PIE uses terms of computation and communication intensiveness in the program. Uysal et. al. models the application by using application emulators. An emulator can be seen as a skeleton application acting from the point of view of communication like the real application. PerPreT uses task graphs to represent SPMD message-passing programs. Each communication- and computation step is defined by a formula in which N (the problem size) and P (the number of processors) act as parameters.

3.2.5 Trace-based

This technique is based on executing a program on a single- or multiprocessor, which produces a trace file representing the behavior of the program. The trace file is the basis for the simulation, normally with respect to the interactions (synchronizations/messages) and the execution times between the interactions. Clement&Quinn produce formulas which, given P (the number of processors) and/or N (the problem size), estimate the execution time for data-parallel message-passing programs. The formulas are based on the symbolic expressions in

loops, etc. Thus, an analysis of the source code is needed in conjunction with the trace. Both Intrepid and DAGGER are based on the message-driven language CHARM. SPAN determines the algorithmic speedup of a program. SPAN assumes independent processes synchronize via a barrier. Mendes considers prediction between machines of equal size only. The trace file is changed due to differences in processing speed and communication speed.

3.2.6 Trace-based Tools with Visualization

AIMS/MK is targeted for loop-parallelism, normally operating on arrays. AIMS contains a source-code to source-code translator for inserting probes. The user may interact, selecting what is to be monitored and where, or simply use the default. The (monitored) application is executed on the target machine and a trace-file is obtained. The trace file can be inspected by replaying the file, which shows both animated views and automatically scrolling time-line views. Statistics are also shown. Performance prediction is done for both P (the number of processors) and N (the problem size). Two different approaches to predictions are used. The first is the compiler-assisted complexity analysis that expresses the scalability as a function of N and P. Static information from the compiler, dynamic information from the trace file for determining time constants, and an algorithmic model over the network are needed. The compiler analysis only considers loops with a constant stride of the loop-variable. The second approach is simulation by defining a model. The model is created by monitoring interesting/important parts of the source code. The selection of constructs to be monitored is to a large extent determined by the user, and the accuracy of the predictions relies on the user's skill in determining the proper monitoring. In [11], the mean error is about 20%, and up to 30% in some cases. None of the approaches handles data-dependent complexities and conditions for communications. AIMS assumes well-balanced application loads and a contention-free network.

Demaine uses a library to trace the start and end of the program, and the sends and receives in point-to-point communication. No collective communication is handled. The trace file contains the sequence of the message-passing operation as well as the computation time between the operations. A simulator schedules the events and approximates the time that each event would have taken on an unloaded parallel system. The communication time is not contained in the trace file; the simulator predicts the time need for a message to travel form one processor to another. The communication model uses a startup time and the time relative to the size of the message. The timing is derived by repeatedly sending messages of different sizes in a ping-pong fashion between two processors on

the target machine, and the measuring the time. The down side of this method is that this requires access to the target host; on the other hand, changing the target host is easy. The simulator generates a log file viewable by the Upshot tool. Validation is performed on a cluster with four directly connected IBM RS/6000s and using two sort algorithms (bubble sort and quick-merge sort) and two matrix operations called “fan-in” and “fan-out”. The validation shows an error of 9.9%.

PARAVER uses an instrumented MPI library and requires no change to the application code. The simulator (DIMEMAS) simulates any number of machines with one or several processors. DIMEMAS considers parameters such as scheduling policies when several processes are on the same processor (Round-Robin and First-In-First-Out), data transfer time, communication start-up time, the number of communication links between the processors, and processor relative speed. The user can annotate sequential blocks of code that can be executed as parallel threads. It is the users responsibility to make dependency checks. The simulator uses the annotations for SMP predictions. The accuracy of the predictions is approximately 10%, but in [71] it is more than 20%. The visualization is basically a Gantt diagram based on the processor nodes. Communication is indicated between the processors as lines.

PS uses a hierarchy of models for simulation. The models ranges from the (physical) network, through network interfaces, TCP/IP protocol, PVM daemons, up to the PVM program. PVM programs are modeled on CPU bursts with communication in between. There are three ways to model a program. First, a PVM program can be instrumented and monitored on a uni- or multiprocessor which results in a trace file. The two other methods are modeling and direct-execution. PS handles heterogeneous clusters by providing the simulator with the relative speed of the nodes. The PS predictions have an error ranging up to 27.1% as reported in [6], and claimed to be at most 8% in [5]. Visualization is done using ParaGraph.

SIEVE is a framework for performing predictions and visualizations. The data collection has not been taken into consideration. The tool uses scripts to define the syntax of the data collected; any file format can be used. The data is stored in a relational database and is represented as a spreadsheet for the user. Macros are used to implement the simulator. The macros manipulate the data in the spreadsheet in order to re-schedule the execution. SIEVE is claimed to predict both N and P. The visualization in SIEVE is also configurable using macros. SIEVE can be seen as a framework for building a specific simulator with visualization ability. A demonstration is given with programs written in pC++. The

performance prediction in the examples is based on Amdahl's law. There is no validation of the predications.

Speedy is a part of the TAU tool kit. Speedy includes the ExtraP for the actual prediction. TAU traces an execution of a program with *n* threads on a uni-processor execution and using a non-preemptive version of the pC++ thread library. The (simulated) target machine has *n* processors, i.e., one processor for each thread. The recorded events are sorted on a per thread basis; each thread is then simulated from start to end. The processor model handles a speed-ratio between different processors. The network model includes start-up time, bandwidth, message types and sizes, network topology, and network contention. The barrier model is based on a linear master-slave algorithm, where thread 0 is the master and the other slaves. The validation performed using a simple matrix multiplication application shows an error larger than 50% [97].

3.2.7 Conclusions of Simulation Tools

From Table 2 it is clear that most simulation tools support message-passing (86%), and only 39% handle shared-memory. Of the tools that support shared-memory, 64% also support message-passing. This is almost identical to the situation with visualization tools (see Section 3.1). VPPB is the only tool in the trace-based category that supports shared memory programming using threads. Two tools included in Table 2 which handle shared memory are SIEVE and Speedy. Both tools handle the pC++ language [75, 88]. The pC++ language assumes one thread/process per processor and that global barriers are the only synchronization mechanism. In pC++, the shared memory is owned by a single processor; the other processors can read the values by explicit function calls. Thus, the programming model in pC++ differs from Solaris or POSIX threads. SPAN also handles shared-memory; however, the language is more simplified than pC++ since it only allows independent processes in the parallel sections. VPPB is as accurate as Demaine and Eom&Hollingsworth, and significantly better than the other trace-based tools (by a factor of 2 to 5).

Table 2 shows that most tools are either trace-based or direct-execution. One reason for the number of trace-based tools could be the impact from pure visualization tools (Section 3.1). Many of these pure visualization tools collect a trace, and it seems relatively straight forward to use the same technique to "reschedule" the trace and obtain a prediction for another system. AIMS/MK, PARAVER/DIMEMAS and Speedy are typical examples of this, since AIMS, PARAVER and Tau (includes Speedy) can be used without the corresponding simulator for visualization of a multiprocessor execution. Demaine and PS use

Upshot and ParaGraph, respectively, for the visualization of the predicted execution. Thus, even for these tools there is a strong connection with the pure visualization tools.

As with the pure visualization tools the size of the trace file may be a matter for concern, e.g. Pablo, that collects a trace-file, has been superseded by svPablo, which collects statistics only. Techniques using statistics are difficult to use for simulation purposes. However, 38% of all pure visualization tools are capable of on-the-fly visualization. This is analogous to the direct-execution approach. The obvious advantage is that no trace is needed. As opposed to the trace-based approach, where several experiments can be performed on the same trace file, direct-execution must re-execute the program for each new experiment.

The interpretation category can be seen as an extreme version of the direct-execution approach where everything is simulated. The obvious advantage is that any system can be simulated. These tools are useful for processor manufactures when it comes to reducing the cost of experimental processors on silicon. These simulators can also be shipped to program developers before the silicon chip exists. Thus, software can be available from the very first day. The drawback is the simulation speed: a slowdown of thousands is not unusual. This makes the tools less practical as one would wish, with the result that only a few tools have proved successful.

Modeling is another approach. Since a model can generally can be used for performance prediction only, users are perhaps reluctant to invest time in building the model. The last approach, static analysis, is complicated and limited in its scope due to the fact information for loop-bounds, etc., is often lacking in the code.

3.3 Thread Allocation Tools

Thread (or rather task) allocation is frequently considered in the real-time area. The main goal in real-time is to meet all (explicit) deadlines. The GAST tool [60] is somewhat similar to the VPPB discussed in papers IV and V. GAST originates from the real-time area. With GAST it is possible to automate scheduling by defining different scheduling algorithms. The GAST tool needs a specification of all tasks, their worst-case execution time, period, deadline and/or dependencies. This specification must be done by hand. High performance computing does not necessarily have either periods or deadlines. A task may only be a fraction of a thread, since it cannot synchronize with another task inside the task itself. Each thread must then be split into several tasks (by hand).

There are many heuristic allocation algorithms, e.g. simulated annealing [10, 64], tabu search [107], genetic algorithms [10]. I show in paper V that at some point further iterations of the heuristic algorithm are no longer worthwhile. Simulated annealing, for example, occasionally selects a worse allocation (the less good the allocation, the smaller the chance of selecting it) in order to avoid local maxima. The probability for selecting a worse allocation decreases over time as well. Thus, simulated annealing will jump frequently in the beginning and then less and less frequently as time goes on. This is in contrast to paper V, where I propose that one should select one starting point, use it for iterations as long as it is worthwhile, and only then select a new starting point, and so on.

3.4 Profiling Tools and Critical Path

Today, profiling tools cannot handle multithreaded programs on a multiprocessor in a satisfactory way. The problem is that the profilers assume that all executed code segments contribute to the total execution time. Table 3 illustrates some profilers for the Solaris operating system.

Table 3: Different profilers for the Solaris operating system.

Name/ Reference	Handle multithreading	Handle multiprocessors	Assumes that all executed code contributes to the total execution time
prof [126]	Yes	No	Yes
gprof [125]	Yes	No	Yes
tha [124]	Yes	Yes	Yes
Quantify [111]	Yes	No	Yes

All tools in Table 3 assume that all executed code contributes to the total execution time as in sequential programs. Quantify and tha are able to give a profile per thread. This helps the developer to some extent. However, as shown in paper VI, the critical path may include parts of several threads.

Previous critical path work handles one process/thread per processor [52, 94, 140]. In [140], an attempt to solve the issue when having fewer processors than threads is provided in a form of as a sketch. Unfortunately, the suggested solution did not work in all cases, as shown in paper VI.

3.5 Analytical Bounds

The basic approach in paper VII is to obtain a performance bound, which we know is achievable. This approach has been used for some time in real time systems, where it is well known that a set of n tasks is schedulable if the sum of the processor utilization is less than $n(2^{1/n} - 1)$ [19]. Similar results also exist for real-time process sets which operate under what is called an "age constraint" [81, 82].

Proof techniques similar to those in paper VII have been used for performance bounds in a number of application areas besides multiprocessor scheduling, e.g. cache memory systems [78], parallel accesses to multiprocessor memory systems [83], and message scheduling on a number of communication links [84].

3.6 Simulation Models

The Direct Simulation Model in VPPB was used as it intuitively mimics the primitives in Solaris. For semaphores there is no support for the thread to determine which other thread that woke it up. Thus, selecting the Direct Simulation Model in VPPB can be seen as a way of mimicing such behavior.

The Client-Server Simulation Model is similar to the one used in [46]. The language [47] supported by the tool is message-driven. This means that a process is triggered by incoming messages. Thus, each process acts as a server receiving messages, processing them, perhaps by sending new messages to other processes taking the role of a client, and also possibly sending back an answer. The Client-Server Simulation Model seems to have been chosen in order to mimic the language.

Strict Sequence Simulation Model is used in two MPI tools [34, 44]. The MPI receive primitives have a parameter that shows the sender of the message. This makes the Strict Sequence Simulation Model the most straightforward choice. In languages with a simple fork-join structure, the Strict Sequence Model works perfectly [89, 97, 115, 123]. The work described in [36, 37] seems to use the Strict Sequence Simulation Model in the models that model dependencies, although these are not explicitly expressed. Further, it could be noted that the different models in [36, 37] do not address the same issues as in paper VIII, since the concern is simulation speed vs. accuracy, in terms of the granularity of events, not the model as such. Mendes [92] shows that the Strict Sequence Simulation Model only handles deterministic programs (Mendes call these "stable"). Mendes is able to identify an unstable message-passing program and conclude that the tool cannot handle unstable programs.

Tools using direct-execution (Section 3.2.1) or interpretation (Section 3.2.2) techniques handle the non-deterministic issues, but at the expense of, for example, simulation speed. To summarize, the tools mentioned seem to use a simulation model that intuitively suits the environment.

4 Conclusions and Future Work

In this thesis I have presented techniques for performance prediction and improvement of parallel programs in multiprocessors. I have demonstrated how to simulate a multiprocessor execution of a parallel program based on a monitored execution of the program on a uni-processor. The (simulated) multiprocessor execution is graphically visualized. As my thesis shows, the prediction facility can be used for heuristic algorithms to find an allocation of threads to processors. Further, I have presented an algorithm that identifies the critical path of the parallel program on a multiprocessor. I have also shown how to calculate a tight bound on the minimal completion time for the optimal allocation of threads to processors. Finally, I have discussed the implications of one's choice of simulation model. The techniques described have been manifested in a prototype tool.

Although a number of issues have been dealt with in this thesis, there are additional aspects which deserve further investigation. One future extension of the techniques described here could be to handle processes and the communication between them. Software systems that include several different processes (with different source code as opposed to the SPMD, Single Program Multiple Data, approach often used by MPI and PVM) deserve consideration. The techniques should still be operational on a uni-processor workstation but be capable of simulating several computers connected via a network.

Another future direction for research could be to take a look at Java; the issues here are not only technical but also concern how to present results to the programmer in a convenient way. Java uses monitors and critical regions as language primitives for synchronization. On the Solaris Java Virtual Machine these primitives are mapped to Solaris mutexes, semaphores, etc. Thus, the actual primitives that are executed on the host are not the same as those the programmer wrote. How should the Solaris primitives be presented so the Java programmer (who perhaps has no knowledge of Solaris primitives) can improve the performance of the Java program? This is an interesting and important question.

An extension of the techniques might take the form of using the hardware performance counters available in modern processors. A number of APIs exist today (e.g. PAPI [16]), and therefore, tracing is perhaps not the primary issue here. However, integration of the statistical information that the hardware counters represent in the (current) event based simulation technique is a more challenging issue.

5 References

[1] W. Adve, R. Bagrodia, E. Deelman, T. Phan, and R. Sakellariou, “Compiler-Supported Simulation of Very Large Parallel Applications," Proc. Supercomputing’99, CD-ROM, 1999.

[2] T. Anderson and E. Lazowska, “Quartz: A Tool for Tuning Parallel Program Performance,” Proc. ACM Conference on Measurement and Modeling of Computer Systems, pp. 115-125, 1990.

[3] B. Armstrong and R. Eigenmann, “Performance Forecasting: Characterization of Applications on Current and Future Architectures,” Purdue Univ. School of ECE, High-Performance Computing Lab. Technical report ECE-HPCLab-97202, 1997. Available at http://ParaMount.www.ecn.purdue.edu/ParaMount/PerFore/publications.html, site visited 2 April 2002.

[4] B. Armstrong and R. Eigenmann, “Performance Forecasting: Towards a Methodology for Characterizing Large Computational Applications,” Proc. International Conference on Parallel Processing, pp. 518-525, 1998.

[5] 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.

[6] R. Aversa, N. Mazzocca, and U. Villano, “Design of a simulator of heterogeneous computing environments,” Simulation Practice and Theory. Vol. 4, No. 2-3, pp. 97-117, 1996.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, “A static performance estimator to guide data partitioning decisions,” Proc. 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 213-223, 1991.

[8] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam, “Visualization and Debugging in a Heterogeneous Environment,” Computer, Vol. 26, No. 6, pp. 88-95, 1993.

[9] R. Blasko, “Simulation Based Performance Prediction by PEPSY,” Proc. 28th Annual Simulation Symposium, pp. 341-349, 1995.

[10] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, “Scheduling Computer and Manufacturing Processes,” Springer Verlag, ISBN 3-540-61496-6, 1996.

[11] R. Block, S. Sarukkai, and P. Mehra, “Automated Performance Prediction of Message-Passing Parallel Programs,” Proc. Supercomputing 1995, CD-ROM, 1995.

[12] R. Borgeest, B. Dimke, and O. Hansen, “A trace based performance evaluation tool for parallel real time systems,” Parallel Computing, Vol. 21, No. 4, pp. 551-564, 1995.

[13] R. Bosch, C. Stolte, G. Stol, M. Rosenblum, and P. Hanrahan, “Performance analysis and visualization of parallel systems using SimOS and Rivet: a case study,” Proc. 6th International Symposium on High-Performance Computer Architecture, pp. 360-371, 1999.

[14] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy, “PerPreT - A Performance Prediction Tool for Massively Parallel Systems,” 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation Performance Tools ‘95, pp. 284-298, 1995.

[15] E. Brewer, C. Dellarocas, Colbrook, and W. Weihl, “PROTEUS: A high-performance parallel-architecture simulator”, Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, 1991. Available at http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-516.pdf, site visited 2 April 2002.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A portable programming interface for performance evaluation on modern processors,” International Journal of High-Performance Computing Applications,. Vol. 14, No. 3, pp. 189-204, 2000.

[17] H. Brunst, M. Winkler, W. Nagel, and H.-C. Hoppe, “Performance optimization for large scale computing: the scalable Vampir approach,” Proc. International Conference on Computational Science, Part II, pp. 751-760, 2001.

[18] M. Bubak, W. Funika, B. Balis, R. Wismüller, “Performance measurement support for MPI applications with PATOP,” PARA2000 Workshop on Applied Parallel Computing, pp. 288-295, 2000.

[19] A. Burns and A. Wellings, “Real-Time Systems and Programming Languages,” Addison-Wesley, 2001, ISBN 0-201-72988-1.

[20] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, 1997, ISBN 0-20-163392-2.

[21] M. Calzarossa, L. Massari, A. Merlo, D. Tessera, and M. Vidal, “A tool for performance analysis of parallel programs,” Rivista di Informatica, XXVIII(2), pp. 103-111, 1998.

[22] M. Calzarossa, L. Massari, A. Merlo, and D. Tessera, “Parallel Performance Evaluation: the MEDEA Tool,” in H. Liddel, A. Colbrook, B. Hertzberger, and P. Sloot, ed., High-Performance Computing and Networking, volume 1067 of Lecture Notes in Computer Science, Springer-Verlag, pp. 522-529, 1996.

[23] B. Cantrill and T. Doeppner Jr, “ThreadMon: a tool for monitoring multithreaded program performance,” Proc. 30th Hawaii International Conference on System Sciences, Vol. 1, pp. 253-265, 1997.

[24] C. Carothers, B. Topol, R. Fujimoto, J. Stasko, and V. Sunderam, “Visualizing parallel simulations that execute in network computing environments,” Future Generation Computer Systems, Vol. 15, No. 4, pp.513-529, 1999.

[25] G. Chillariga and B Ramkumar, “Performance prediction for portable parallel execution on MIMD architectures,” Proc. 9th International Parallel Processing Symposium, pp. 630-634, 1995.

[26] M. Clement and M. Quinn, “Automated performance prediction for scalable parallel computing,” Parallel Computing. Vol. 23, No. 10, pp. 1405-1420, 1997.

[27] Cray, “Introducing the MPP Apprentice Tool,” Publication Number 2511_3.0, Cray, 1997. Available at http://www.cray.com/craydoc/, site visited 2 April 2002.

[28] Cray, “PAT manual page,” 1999. Available at http://www.unics.uni-hannover.de/rrzn/gehrke/man_pat.html, site visited 2 April 2002.

[29] H. Davis, S. Goldschmidt, and J. Hennessy, “Tango: a multiprocessor simulation and tracing system,” Stanford University. Computer Systems Laboratory Technical Report CSL-

TR-90-439, 1990. Available at ftp://db.stanford.edu/pub/cstr/reports/csl/tr/90/439/CSL-TR-90-439.pdf, site visited 2 April 2002.

[30] L. De Rose and D. Reed, “SvPablo: A multi-language architecture-independent performance analysis system,” Proc. International Conference on Parallel Processing (ICPP’99), pp. 311-318, 1999.

[31] L. De Rose, Y. Zhang, and D. Reed, “SvPablo: A Multi-language Performance Aanalysis System,” Proc. 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools’98), pp. 352-355, 1998.

[32] T. Delaitre, P. Vekariya, R. Bigeard, T. Delaitre, G. Ribeiro Justo, S. Winter, and M. Zemerly, “EDPEPPS: An Integrated Graphical Toolset for the Design and Performance Evaluation of Portable Parallel Software,” Proc. Euro-Par, pp. 113-125, 1997.

[33] T. Delaitre, M. Zemerly, G. Justo, O. Audo, and S. Winter, “EDPEPPS: an integrated environment for the parallel development life-cycle,” Future Generation Computer Systems, Vol. 16, No. 6, pp. 585-595, 2000.

[34] 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.

[35] M. Durbhakula, V. Pai, S. Adve, “Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors,” Proc. 5th International Symposium on High-Performance Computer Architecture, pp. 23-32, 1999.

[36] H. Eom and J. Hollingsworth, “Achieving Efficiency and Accuracy in Simulation for I/O-Intensive Application,” Journal of Parallel and Distributed Computing, Vol. 61, No. 12, pp. 1732-1750, 2001.

[37] H. Eom and J. Hollingsworth, “Speed vs. accuracy in simulation for I/O-intensive applications,” Proc. 14th International Parallel and Distributed Processing Symposium, pp. 315-322, 2000.

[38] T. Fahringer and A. Pozgaj, “P^3T+: a performance estimator for distributed and parallel programs,” Scientific Programming, Vol. 8, No. 2, pp. 73-93, 2000.

[39] T. Fahringer, A. Pozgaj, J. Luitz, and H. Moritsch, “Evaluation of P^3T+: a performance estimator for distributed and parallel Programs,” Proc. 14th International Parallel and Distributed Processing Symposium (IPDPS 2000), pp. 229-234, 2000.

[40] M. Garey and D. Johnson, “Computers and Intractability,” W. H. Freeman and Company, 1979.

[41] G. Geist, J. Kohl, and P. Papadopoulos, “Visualization, Debugging and Performance in PVM,” Proc. of Visualization and Debugging Workshop, pp. 65-78, 1994.

[42] A. van Gemund, “The PAMELA Approach to the Performance Simulation of Parallel and Distributed Systems,” Proc. SCS European Simulation Symposium, pp. 365-370, 1993.

[43] A. van Gemund and G. Reijns, “Predicting Parallel System Performance with PAMELA,” Proc. International ASCI Conference, pp. 422-431, 1995.

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

[45] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu, “Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs,” Proc. 5th Symposium on the Frontiers of Massively Parallel Computation, pp. 422-429, 1994.

[46] A. Gürsoy and L. Kale, “Simulating message-driven programs,” Proc. International Conference on Parallel Processing, Vol. 3, pp. 223-230, 1996.

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

[48] M. Gutzmann and K. Steffan, “PEPSIM-ST: a Simulator Tool for benchmarking,” Proc. 2nd Joint International Conference on Vector and Parallel Processing (CONPAR’92 - VAPP V), pp. 665-676, 1992.

[49] M. Heath and J. Etheridge, “Visualizing the Performance of Parallel Programs,” IEEE Software, Vol. 8, No. 5, pp. 29-39, 1991.

[50] J. Hennessy and D. Patterson, “Computer Architecture a Quantitative Approach,” 2nd Edition, Morgan Kaufmann Publishers, ISBN 1-55860-329-8, 1996.

[51] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss, and R. Woo, “An Integrated Performance Visualizer for MPI/OpenMP Programs,” Proc. International Workshop on OpenMP Applications and Tools, pp. 40-52, 2001.

[52] J. Hollingsworth, “Critical Path Profiling of Message Passing and Shared-Memory Programs,” IEEE Transaction on Parallel and Distributed Systems, Vol. 9, No. 10, pp. 1029-1040, 1998.

[53] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, “Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors,” IEEE Computer, Vol. 35, No. 2, pp. 40-49, 2002.

[54] S. Hummel, D. Kimelman, E. Schonberg, M. Tennenhouse, and D. Zernik, “Using program visualization for tuning parallel-loop scheduling”, IEEE Concurrency, Vol. 5, No. 1, pp. 26-40, 1997.

[55] K. Hwang and Z. Xu, “Scalable Parallel Computing,” McGraw-Hill, ISBN 0-07-031798-4, 1998.

[56] IBM, “SP Parallel Programming Workshop visualization tool (vt),” http://www.mhpcc.edu/training/workshop/vt/MAIN.html, 2001. Visited 2 April 2002.

[57] R. Irvin and B. Miller, “Multiapplication Support in a Parallel-Program Performance Tool,” IEEE Parallel and Distributed Technology, Vol. 2, No. 1, pp. 40-50, 1994.

[58] R. Irvin and B. Miller, “A Performance Tool for High-Level Parallel Programming Languages,” In Programming Environments for Massively Parallel Distributed Systems, Eds. K. M. Decker et. al., pp. 299-313, 1994.

[59] M. Ji, E. Felten, and K. Li, “Performance Measurements for Multithreaded Programs,” Performance Evaluation Review, Vol. 26, No. 1, pp. 161-170, 1998.

[60] J. Jonsson, “GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques,” Proc. International Conference on Parallel Processing, pp. 441-450, 1998.

[61] P. Kacsuk, “Performance visualization in the GRADE parallel programming environment,” Proc. 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, pp. 446-450, 2000.

[62] P. Kacsuk, G. Dózsa, and T. Fadgyas, “Designing Parallel Programs by the Graphical Language GRAPNEL,” Microprocessing and Microprogramming, No. 41, pp. 625-643, 1996.

[63] S. Kim, M. Voss, B. Kuhn, H.-C. Hoppe, and W. Nagel, “VGV: Supporting Performance Analysis of Object-Oriented Parallel Applications,” Proc. Workshop on High-Level Parallel Programming Models and Supportive Environments, to appear, 2002.

[64] S. Kirkpatrick, “Optimization by Simulated Annealing: Quantitative Studies,” Journal of Statistical Physics, Vol. 34, No. 5-6, pp. 975-986, 1984.

[65] E. Kraemer and J. Stasko, “Creating an Accurate Portrayal of Concurrent Executions,” IEEE Concurrency, Vol. 6, No. 1, 1998.

[66] D. Kranzlmüller, S. Grabner, J. Volkert, “Debugging with the MAD environment,” Parallel-Computing, Vol. 23, No. 1-2, pp. 199-217, 1997.

[67] D. Kranzlmüller, C. Schaubschläger, and J. Volkert, “A Brief Overview of the MAD Debugging Activities,” Proc. 4th International Workshop on Automated Debugging, pp. 229-234, 2000.

[68] W. Krotz and H.-C. Hoppe, “PALLAS parallel tools a uniform programming environment from workstations to teraflop computers,” Proc. ParCo 97, pp. 349-358, 1997.

[69] K. Kubota, K. Itakura, M. Sato, and T. Boku, “Practical Simulation of Large-Scale Parallel Programs and its Performance Analysis of the NAS Parallel Benchmarks,” Euro-Par, pp. 244-254, 1998.

[70] J. Labarta, S. Girona, and T. Cortes, “Analyzing scheduling policies using Dimemas,” Journal of Parallel Computing, Vol. 23, No. 1-2, pp. 23-34, 1997.

[71] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, “DiP: A Parallel Program Development Environment,” Proc. Euro-Par, pp. 665-674, 1996.

[72] LAM / MPI Parallel Computing, “Manual page for XMPI 2.2,” http://www.lam-mpi.org/software/xmpi/help.php, 2001. Visited 2 April 2002.

[73] LAM Team, “XMPI -- A Run/Debug GUI for MPI,” http://www.lam-mpi.org/software/xmpi/, 2001. Visited 2 April 2002.

[74] T. LeBlanc, J. Mellor-Crummey, and R. Fowler, “Analyzing parallel program executions using multiple views,” Journal of Parallel and Distributed Computing, Vol. 9, No. 2, pp. 203-217, 1990.

[75] J. Lee and D. Gannon, “Object oriented parallel programming: experiments and results,” Proc. Supercomputing, pp. 273 - 282, 1991.

[76] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, “Visualizing performance debugging,” IEEE Computer, Vol. 22, No. 10, pp. 38-51, 1989.

[77] S. Lei and K. Zhang, “Performance Visualisation of Message Passing Programs Using Relational Approach,” Proc. 7th ISCA International Conference on Parallel and Distributed Computing Systems, pp. 740-745, 1994.

[78] H. Lennerstad and L. Lundberg, “Optimal Worst Case Formulas Comparing Cache Memory Associativity,” SIAM Journal of Computing, Vol. 30, No. 3, pp. 872-905, 2000.

[79] O. Loques, J. Leite, and E. Carrera, “P-RIO: A Modular Parallel-Programming Environment,” IEEE Concurrency, Vol. 6, No. 1, pp. 47-57, 1998.

[80] L. Lundberg, “Evaluating the Performance Implications of Binding Threads to Processors,” Proc. 4th International Conferece on High Performance Computing, pp. 393-400, 1997.

[81] L. Lundberg, “Fixed Priority Scheduling of Age Constraint Processes,” Proc. 4th International Euro-Par Conference, pp. 288-296, 1998.

[82] L. Lundberg, H. Lennerstad, “An Optimal Upper Bound on the Minimal Completion Time in Distributed Supercomputing,” Proc. ACM International Conference on Supercomputing, pp. 196-203, 1994.

[83] L. Lundberg, H. Lennerstad, “Bounding the Gain of Changing the Number of Memory Modules in Shared Memory Multiprocessor,” Nordic Journal of Computing, Vol. 4, No. 3, pp. 233-258, 1997.

[84] L. Lundberg, H. Lennerstad, “Using Recorded Values for Bounding the Minimum Completion Time in Multiprocessors,” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 4, pp. 346-358, 1998.

[85] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Hökberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” IEEE Computer, Vol. 35, No. 2, pp. 50-58, 2002.

[86] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, “SimICS/sun4m: A Virtual Workstation,” Proc. USENIX Annual Technical Conference, pp. 119-130, 1998.

[87] A. Malony, D. Hammerslag, and D. Jablonowski, “Traceview: A Trace Visualization Tool,” IEEE-Software, Vol. 8, No. 5, pp. 19-28, 1991.

[88] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan, “Implementing a parallel C++ runtime system for scalable parallel systems,“ Proc. Supercomputing, pp. 588 - 597, 1993.

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

[90] A. Malony and S. Shende, “Performance Technology for Complex Parallel and Distributed Systems,” Proc. Third Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000), pp. 37-46, 2000.

[91] W. Meira, Jr., T. LeBlanc, and A. Poulos, “Waiting Time Analysis and Performance Visualization,” Proc. 1st SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 1-10, 1996.

[92] C. Mendes, "Performance Prediction by Trace Transformation," 5th Brazilian Symposium on Computer Architecture, 1993.

[93] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measurement Tools,” IEEE Computer 28, Vol. 28, No. 11, pp. 37-46, 1995.

[94] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, “IPS-2: The Second Generation of a Parallel Program Measurement System”, IEEE Transactions on Parallel and Distributed Computing, Vol. 1, No. 2, pp. 206-217, 1990.

[95] B. Miller, J. Hollingsworth, and M. Callaghan, “The Paradyn Parallel Performance Tools and PVM,” Proc. 2nd Workshop on Environments and Tools for Parallel Scientific Computing, SIAM Press, pp. 201-210, 1994.

[96] B. Mohr, D. Brown, and A. Malony, “TAU: A Portable Parallel Program Performance Analysis Environment for pC++,” Proc. 3rd Joint International Conference on Vector and Parallel Processing, pp. 29-40, 1994.

[97] 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.

[98] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. Hill, J. Larus, and D. Wood, “Fast and Portable Parallel Architecture Simulators: Wisconsin Wind Tunnel II,” IEEE Concurrency, Vol. 8, No. 4, pp. 12-20, 2000.

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

[100] W. Nagel and A. Arnold, “Performance Visualization of Parallel Programs - The PARvis Environment -,” Proc 1994 Intel Supercomputer Users Group (ISUG) Conference, pp. 24-31, 1994.

[101] G. Nudd, D. Kerbyson, and E. Papaefstathiou, “Pace-A Toolset for the Performance Prediction of Parallel and Distributed Systems,” International Journal of High Performance Computing Applications, Vol. 14, No. 3, pp. 228-251, 2000.

[102] G. Nutt, A. Griff, J. Mankovich, and J. McWhirter, “Extensible Parallel Program Performance Visualization,” Proc. Mascots ‘95, pp. 205-211, 1995.

[103] OSE home page, “Illuminator”, http://www.ose.com/downloads/pdfs/products/illuminator_0106.pdf, 1999. Visited 2 April 2002.

[104] PACE home page, http://www.dcs.warwick.ac.uk/~hpsg/html/htdocs/public/content-pace.html, 2001. Visited 2 April 2002.

[105] V. Pai, P. Ranganathan, and S. Adve, “RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors,” Proc. 3rd Workshop on Computer Architecture Education, 1997.

[106] E. Papaefstathiou, D. Kerbyson, G. Nudd, and T. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” Proc. 8th ISCA International Conference on Parallel and Distributed Computing Systems, pp. 527-533, 1995.

[107] S. Porto, J.-P. Kitajima, and C. Ribeiro, “Performance Evaluation of a Parallel Tabu Search Task Scheduling Algorithm,” Journal of Parallel Computing, Vol. 26, No. 1, pp. 73-90, 2000.

[108] S. Prakash, E. Deelman, and R. Bagrodia, “Asynchronous parallel simulation of parallel programs,” IEEE Transactions on Software Engineering, Vol. 26, No. 5, pp. 385-400, 2000.

[109] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to Visualize and Analyze Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, 1995. Available at ftp://ftp.ac.upc.es/pub/reports/CEPBA/1995/UPC-CEPBA-95-03.ps.Z, site visited 2 April 2002.

[110] P. Ramachandran and L. Kale, “Web-based Interaction and Monitoring for Parallel Programs (Via Conspector),” Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Technical Report 99-05, 1999. Available at http://charm.cs.uiuc.edu/papers/ConspectorSC99.html, site visited 2 April 2002.

[111] Rational, “Quantify version 4.2,” http://www.rational.com.

[112] D. Reed, “Experimental Analysis of Parallel Systems: Techniques and Open Problems,” Computer Performance Evaluation, Eds: G. Haring et. al., Lecture Notes in Computer Science 794, pp. 25-51, Springer Verlag, 1994.

[113] D. Reed, R. Olson, R. Aydt, T. Madhyasta, T. Beckett, D. Jensen, B. Nazief, and B. Totty, “Scalable Performance Environments for Parallel Systems,” Proc. 6th Distributed Memory Computing Conference, pp. 562-569, 1991.

[114] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, “Complete computer system simulation: the SimOS approach,” IEEE Parallel & Distributed Technology: Systems & Applications, Vol. 3, No. 4, pp. 34-43, 1995.

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

[116] E. Schaffer, D. Reed, S. Whitmore, and B. Schaeffer, “Virtue: Performance Visualization of Parallel and Distributed Applications,” IEEE Computer, Vol. 32, No. 12, pp. 44-51, 1999.

[117] T. Schnekenburger, M. Friedrich, A. Weininger, and T. Schoen, “ParSim: A Tool for the Analysis of Parallel and Distributed Programs,” CONPAR 92, pp. 689-700, 1992.

[118] M. Schumann, “Automatic Performance Prediction to Support Cross Development of Parallel Programs,” Proc. SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 88-97, 1996.

[119] Z. Segall, L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” IEEE Software, Vol. 2, No. 6, pp. 22-37, 1985.

[120] S. Shende and A. Malony, “Integration and Application of the TAU Performance System in Parallel Java Environments,” Proc. the Joint ACM Java Grande - ISCOPE 2001 Conference, pp. 87 - 96, 2001.

[121] A. Sinha and L. Kale, “Projections: A Preliminary Performance Tool for Charm,” Parallel Systems Fair at International Symposium on Parallel Processing, pp. 108-114, 1993.

[122] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran, “A Simulation-based Scalability Study of Parallel Systems,” Journal of Parallel and Distributed Computing, Vol. 22 No. 3, pp. 411-426, 1994.

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

[124] Sun, “Sun WorkShop 4.0 AnswerBook: Beyond the Basics” Sun Microsystems Inc., pp. 42-65, 1996. Available at ftp://192.18.99.138/802-7044/802-7044.pdf, site visited 2 April 2002.

[125] Sun Man Pages, “gprof,” Sun Microsystems Inc., 1998.

[126] Sun Man Pages, “prof,” Sun Microsystems Inc., 1998.

[127] SunSoft, “tnfview Demo Guide,” 1995. The documentation is included in http://ftp.nsysu.edu.tw/Sun/specials/tnftools/tnfdemo.tar.gz, site visited 2 April 2002.

[128] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” Proc. 5th Symposium on the Frontiers of Massively Parallel Computation, pp. 396-405, 1994.

[129] H.-L. Truong, T. Fahringer, G. Madsen, A. Malony, H. Moritsch, and S. Shende, “On using SCALEA for Performance Analysis of Distributed and Parallel Programs,” Proc. Supercomputing 2001 Conference (SC2001), CD-ROM, 2001.

[130] M. Uysal, T. Kurc, A. Sussman, and J. Saltz, “A Performance Prediction Framework for Data Intensive Applications on Large Scale Parallel Machines,” Proc. 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers, pp. 243-258, 1998.

[131] J. Vetter, “VGV: A Performance Analysis Tool for Mixed OpenMP and MPI Parallel Programming Models,” Livermore Computer Training, 2002. Available at http://www.llnl.gov/computing/tutorials/vgv_session1/, site visited 2 April 2002.

[132] A. Weininger, T. Schnekenburger, and M. Friedrich, “Parallel and Distributed Programming with ParMod-C,” Proc. 1st International Conference of the Austrian Center for Parallel Computation, pp. 115-126, 1991.

[133] W. Williams, T. Hoel, and D. Pase, “The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D,” Programming Environments for Massively Parallel Distributed Systems, Eds.: K. M. Decker et. al., Birkhauser Verlag, pp. 333-345, 1994.

[134] R. Wismüller, “State Based Visualization of PVM Applications,” Proc. 3rd European PVM Conference, pp. 91-99, 1996.

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

[136] B. Wylie and A. Endo, “Annai/PMA multi-level hierarchical parallel program performance engineering,” Proc. 1st International Workshop on High-Level Programming Models and Supportive Environments, pp. 58-67, 1996.

[137] B. Wylie and A. Endo, “The Annai/PMA performance monitor and analyzer,” Proc. 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 186-191, 1996.

[138] J. Yan, “Performance Tuning with AIMS - An automated Instrumentation and Monitoring System for Multicomputers,” Proc. 27th Hawaii International Conference on System Sciences, Vol. II, pp. 625-633, 1994.

[139] J. Yan, S. Sarukkai, and P. Mehra, “Performance Measurements, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit,” Software-Practice and Experience, Vol. 25, No. 4, pp. 429-461, 1995.

[140] C.-Q. Yang and B. Miller, “Critical Path Analysis for the Execution of Parallel and Distributed Programs,” Proc. 8th International Conference on Distributed Computing Systems, pp. 366-375, 1988.

[141] O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward scalable performance visualization with Jumpshot,” International Journal of High-Performance Computing Applications, Vol. 13, No. 3, pp. 277-288, 1999.

[142] Q. Zhao, “Gthread - KSR Pthread Program Visualization Package,” College of Computing, Georgia Institute of Technology, 1994.

[143] Q. Zhao and J. Stasko, “Visualizing the Execution of Threads-based Parallel Programs,” Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Technical Report GIT-GVU-95-01, 1995. Available at ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports, site visited 2 April 2002.

Paper Section

Paper I

VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs

Magnus Broberg, Lars Lundberg, and Håkan Grahn
12th International Parallel Processing Symposium, 1998

I

VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Department of Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@ide.hk-r.se

Abstract

Efficient performance tuning of parallel programs is often hard. In this paper we describe an approach that uses a uni-processor execution of a multithreaded program as reference to simulate a multiprocessor execution. The speed-up is predicted, and the program behaviour is visualized as a graph, which can be used in the performance tuning process.

The simulator considers scheduling as well as hardware parameters, e.g., the thread priority, no. of LWPs, and no. of CPUs. The visualization part shows the simulated execution in two graphs: one showing the threads' behaviour over time and the other the amount of parallelism over time. In the first graph is it possible to relate an event in the graph to the code line causing the event. Validation using a Sun multiprocessor with eight processors and five scientific parallel applications shows that the speed-up predictions are within +/-6% of a real execution.

1. Introduction

Parallel processing is an important way to increase the performance. It is often easier to develop parallel applications for shared memory multiprocessors than for message passing systems. Shared memory multiprocessors are therefore becoming increasingly important.

The thread concept in the Solaris operating system [13] makes it possible to write multithreaded programs which can be executed in parallel. Having multiple threads does, however, not guarantee that a program will run faster on a shared memory multiprocessor. One major performance problem is that thread synchronizations may create serialization bottlenecks which are often hard to detect.

Removing serialization bottlenecks is referred to as *performance tuning*. Different tools for visualizing the behaviour of, and thus the bottlenecks in, parallel programs have been developed [1, 2, 3, 5, 6, 9, 10, 11, 12, 14, 16, 18]. The tuning process may benefit significantly from using such tools.

Some performance visualization tools show the behaviour of one particular monitored multiprocessor execution of the parallel program [1, 2, 3, 5, 6, 9, 10]. If we monitor the execution on a multiprocessor with four processors such tools make it possible to detect bottlenecks which are present when using four processors. The problem with this approach is the lack of support for detecting bottlenecks which appear when using another number of processors.

There are a number of tools which make it possible to visualize the (predicted) behaviour of a parallel program using any number of processors. However, these tools are either developed for message passing systems [12] or for non-standard programming environments [11, 14, 16, 18].

In this paper we present a performance prediction and visualization tool called VPPB (**V**isualization of **P**arallel **P**rogram **B**ehaviour). Based on a monitored uni-processor execution, the VPPB system shows the (predicted) behaviour of a multithreaded Solaris program using any number of processors. To the best of our knowledge, VPPB is the only available tool which supports this kind of flexible performance tuning of parallel programs developed for shared memory multiprocessors using a widely spread standardized parallel programming environment (Solaris).

Validation using five scientific multithreaded programs from the SPLASH-2 suite [19] and a multiprocessor with eight processors showed that VPPB was able to predict the behaviour very accurately. The maximum difference between the real speed-up and the speed-up predicted by VPPB was 6%, and for most cases the difference was less than or equal to 1%. As discussed above, the predictions are based on recordings from a monitored uni-processor execution. The time overhead for doing these recordings was less than 3% for all five programs.

The paper is structured in the following way. Section 2 gives a short overview. In section 3 the implementation is described. Section 4 describes the validation. A small case study is shown in section 5. Section 6 discusses the limitations and applicability. Section 7 concludes the paper.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in figure 1. The developer writes the multithreaded program, (a) in figure 1, 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 placed *between* the program and the standard thread library. Every 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.

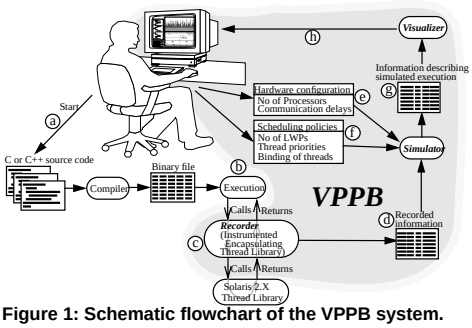


Figure 1: Schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The input for the simulator is the *recorded information,* (d) in figure 1, the hardware configuration (e), and scheduling policies (f). The output from the simulator is information describing the predicted execution (g).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (h). The Visualizer uses the simulated execution (g) as input. 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 for C or C++ programs that uses the built-in thread package in the Solaris 2.X operating system.

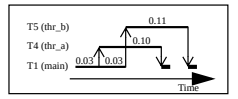
3. Tool Description and Implementation

3.1. The Recorder

In order to trace the behaviour of the program when executed on a uni-processor, the Recorder inserts probes when the program starts. The probes are inserted at specific events, i.e., before and after calls to the thread library, and they do not affect the behaviour or function of the program. For each event, the probes record the following information: when an event has occurred; the type of event, e.g., locking of a mutex; which object the event concerns, e.g., the identity of the mutex being used; the identity of the thread generating the event; and the location of the event in the source code. The data collected by the Recorder is kept in memory until the program terminates, then the recorded data is written to a log file. By using this technique, the intrusion is kept to a minimum.

We will use a small multithreaded program, found in the upper left corner of figure 2, as an example when demonstrating the functionality of the VPPB. The optimal parallel execution of this program can be found in the lower left corner of figure 2; a solid line denotes execution, no line that the thread is blocked, and an arrow represents an event. The Recorder executes the threads sequentially on a uni-processor. The output of the Recorder is the list of events found on the right side of figure 2. The sequentially ordered list is used as the behaviour profile when simulating a multiprocessor execution of the program.

```
void* thread(void *) {
 work();
}
int main() {
 thread_t thr_a, thr_b;
 thr_create(0, 0, thread, 0, 0, &thr_a);
 thr_create(0, 0, thread, 0, 0, &thr_b);
 thr_join(thr_a, 0, 0);
 thr_join(thr_b, 0, 0);
}
```



time	thread	action	duration
0.00	T1	start_collect	
0.0	T1	Running	0.03
0.03	T1	thr_create thr_a	0.04
0.0	T1	Running	0.03
0.10	T1	thr_create thr_b	0.02
0.0	T1	Running	0.05
0.17	T1	thr_join thr_a	0.05
0.0	T4	Running	0.10
0.32	T4	thr_exit	
0.0	T5	Running	0.11
0.43	T5	thr_exit	
0.0	T1	Running	0.10
0.53	T1	ok thr_join thr_a	0.05
0.0	T1	Running	0.01
0.59	T1	thr_join thr_b	0.05
0.0	T1	Running	0.10
0.74	T1	ok thr_join thr_b	0.05
0.0	T1	Running	0.01
0.80	T1	thr_exit	

Figure 2: An example program and the output from the Recorder. The operating system assigns the following thread identity numbers to the threads: main = 1, thr_a = 4, and thr_b = 5. We will use T1, T4, and T5, respectively, when we refer to the different threads.

Our current implementation of the Recorder is based on the ideas described in [8]. We insert a new library between the program and the dynamically linked library `libthread.so.1`, which implements the threads in Solaris 2.X. This is achieved by using the built-in facilities of Solaris with run-time linking and shared objects. The insertion is handled at program start-up by the run-time linker via an environment variable called `LD_PRELOAD`. The probes in the inserted library are exemplified in figure 3, where we show how `thr_exit` is implemented.

The probe does four things. First it looks up the address of the real implementation of `thr_exit` and stores it in a variable. This is done only the first time the probe is called. The next thing is to make a time stamp and store the data about the event. The next part of the code stores which source line the thread primitive was called from. Finally, the probe calls the original function in the Solaris thread library.

The time recorded for each event is wall clock time with a resolution of 1 microsecond. We are can not moni-

```
void thr_exit(void *status) {
    static void (* fptr)() = 0;
    if ( fptr == 0 ) {
        fptr = (void (*)())dlsym(RTLD_NEXT, "thr_exit");
        if ( fptr == NULL ) {
            (void) printf("dlopen: %s\n", dlerror());
            return;
        }
    }
    mthr_collect(THR_EXIT, thr_self(), BEFORE, -1);
    asm("set returnpointer, %l0");
    asm("st %i7, [%l0]");
    mthr_recallAddress();
    (*fptr)(status);
    return;
}
```

Figure 3: The implementation of the `thr_exit` probe.

tor the kernel switches between LWPs (lightweight processes) and are forced to do the monitoring on one single LWP. A more thorough discussion is found in Section 6.

We have divided the tracing of the source code location of the call to a probe into two steps. The first step is to record where the calling code is placed in memory. This is done by saving the return address which is the place where execution will continue after the function has returned. On the SPARC processor this return address is kept in a CPU register called i7. The second step translates the recorded memory addresses into specific source code lines. This is done by using a source code debugger and a small parser, which converts the output from the debugger into a format that is readable for the Simulator and Visualizer.

In every `thr_create` call, a function pointer is supplied. This pointer contains the start address of a new thread. The function pointer is recorded and the debugger is used to translate the address to the function name in the source code.

3.2. The Simulator

The Simulator emulates the scheduling in Solaris 2.5 [13]. In Solaris, threads are used at two levels [17]. The application programmers use user-level threads for expressing parallel execution within a process. Kernel threads are used within the operating system kernel. The kernel knows nothing about user-level threads.

Between the user-level and kernel threads are LWPs. Each Solaris process contains at least one LWP. In most cases, user-level threads are multiplexed on the LWPs of the process, such threads are referred to as unbound threads. It is, however, possible to bind a thread to an LWP. Compared to unbound threads, it is much more expensive to create and synchronize threads which are bound to an LWP [17].

There is a kernel thread for each LWP. However, some kernel threads have no associated LWP, e.g., a thread to service disk requests. Kernel threads are the only objects scheduled by the operating system, and they can either be multiplexed on the processors in the system, or bound to a specific processor. To some extent, the user can control thread scheduling, e.g. by binding threads to LWPs and LWPs to processors. It is also possible to indicate how many LWPs a certain process should have. The (user-level) threads can be created dynamically at run-time.

In the Simulator, threads may be manipulated in the following ways: Each thread can individually be unbound; bound to a LWP; or bound to a certain CPU. A thread that is bound to a CPU is automatically bound to an LWP. Each thread can individually be assigned a certain priority level. This will then override all manipulation of that thread's priority within the log file, e.g., the `thr_setprio` event for that thread will be ignored.

Binding a thread to a CPU can increase the speed of the program [7]. When a thread is moved to a different CPU, parts of the old cache contents has to be moved to the cache on the new processor. This may result in a performance-loss. The Simulator does not simulate the caches, but it is possible to use this facility to determine which thread to bind to which CPU in order to get the best result from a load balancing point of view.

Not only user-level threads has a priority level, but also the LWPs. The priority of an LWP is set by the operating system and is adjusted during run-time depending on, e.g., whether the LWP is interactive or only runs in batch mode. The simulator emulates the priority adjustment as it is handled in Solaris. The length of a time slice for an LWP is related to the priority level, thus we also adjust the time slice length during our simulation.

The following parameters can also be adjusted: the communication delay between the CPUs; the number of processors; and the number of LWPs. In this case the `thr_setconcurrency` in the program has no effect. The communication delay affects how fast an event on one CPU is propagated to another CPU.

The concept of `mutex_trylock` and similar try-operations are handled in the following way: If the thread gained access to the lock in the log file, the simulation will do a `mutex_lock`, otherwise no action is taken by the simulator. The `cond_timedwait` is handled as a delay if the operation timed out in the log file and as an ordinary `cond_wait` operation otherwise. Consequently, the information in the log file corresponds to a deterministic execution of the program with some minor exceptions, which are explained in Section 6.

Creating a bound thread is simulated to take 6.7 times longer than an unbound thread [17]. A synchronization on a semaphore takes 5.9 times longer [17] with bound threads than unbound. This value is used in the simulator for mutexes, conditions, and read/write locks, as well.

When running the Simulator, all events in the log file from the Recorder are sorted into a set of lists, one list for each thread as shown in figure 4.

Our simulation technique is an ordinary eventdriven approach. When the simulation starts, all threads are

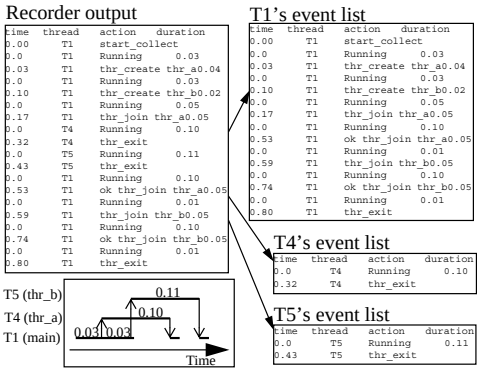


Figure 4: The Simulator's sorting of the log file from the Recorder. We use the same program as in figure 2.

marked as blocked except for the starting (main) thread, i.e., T1. In order to run the threads, the number of LWPs and CPUs specified by the user are simulated. Each (simulated) CPU picks a (simulated) LWP, which in turn picks a (simulated) thread. Each CPU executes the minimum time required for one of the threads to reach an event from the thread's list. The event is simulated and, if appropriate, some blocking or scheduling of threads or LWPs are done.

3.3. The Visualizer

The Visualizer offers two graphs: the parallelism vs. time graph, or *parallelism graph* for short; and the execution flow vs. time graph, or *execution flow graph* for short. The parallelism graph is the upper graph in figure 5. The higher the graph reaches the more parallelism exists in the application. The number of *running* threads are indicated with green. On top of the graph, all the threads that are *runnable* but not running are presented in red. It is easy see where the performance bottlenecks are in time as well as the potential parallelism. This kind of graph har previously been presented as two separete graphs in [15].

The execution flow graph (the lower graph in figure 5) contains more detailed information than the parallelism graph. In the execution flow graph the time is represented on the X-axis and the threads are represented on the Y-axis. A horizontal line indicates that the thread of that Y-position is executing, the lack of a line indicates that the thread can not execute, a grey line that the thread is ready to run but does not have any LWP or CPU to run on. Different events are displayed with different symbols and colours, e.g., all semaphores are shown in red, and the primitives `sema_post` and `sema_wait` are represented as an upward and a downward facing arrow, respectively.

The zoom utility can increase (or decrease) the magnification to an arbitrary magnification degree in steps of a factor of 1.5 or 3. The zoom keeps the left-most time fixed

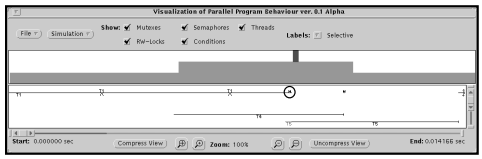


Figure 5: The execution parallelism and flow graphs after running a simulation.

in the execution flow graph. The user can mark a time interval in the parallelism graph , and the execution graph will automatically show only the marked interval.

When there are too many threads to fit in one display, irrelevant threads can be removed automatically. The compression only shows the threads active during the time interval shown in the execution flow graph. It is also possible to control which threads to be shown by hand, allowing the user to select which threads to show from a list.

By selecting a particular (interesting) event, e.g., when thread T1 joins with thread T4 (marked in Figure 5 with a circle), a popup window is shown that gives more information. The selected event starts to flash in the execution flow graph. The popup window gives information about the thread causing the event: the thread identity; the name of the function passed to the `thr_create` function; the time the thread started and ended; how long time the thread actually was working; and finally, the total execution time of the thread (including the time the thread was blocked or runnable). There are also information about the event: that the event was a join operation with thread T4; that the thread was running on CPU 0 in the simulated execution; when the event started, ended, and how long it took to perform; and the source code file and source code line.

The user can step to the previous or next event made by this thread. The execution flow graph is automatically scrolled in order to place the event in the centre of the window. The the popup window is updated with the corresponding data about the new event. Further, the user can find the next or previous similar event. This means that the next event caused by the same event type or variable, e.g., the next operation on the same mutex variable, will be found. Finally, the tool can start an editor with the source code file and highlight the line where the event took place.

4. Validation

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [19]. The programs that we use from the SPLASH-2 suite are: Ocean (with data set 514-by-514 grid), Water-Spatial (512 molecules, 30 time step), FFT (4M points), Radix (16M keys, radix 1024), and LU (contiguous, 768x768 matrix, 16x16 blocks). All programs that we use are from the scientific and engineering domain.

The other benchmarks in the SPLASH-2 suite could not be used as validations. Barnes, Radiosity, Cholesky, and FMM could not run in one single LWP as required by the Recorder. The reason is that these programs all spin on a variable, and since the thread never yields the CPU, no other thread could possibly change the value of that variable. The program Raytrace and Volrend could not be used since all tasks that are executed by a thread are put in a queue. Whenever a thread is idle it steals a task from another thread's queue. The impact of using one LWP gives the result that only one thread steals all tasks, since it never yields the CPU.

All executions were made on a Sun Ultra Enterprise 4000 with 8 processors and 512MByte memory. Since the SPLASH-2 programs are designed to create one thread per physical processor, one log file were made for each processor setup when using the Recorder.

Table 1 shows the measured and predicted speed-up for the five programs. The real speed-up is the middle value of five executions of the program. The values between paranthesis show the maximum and minimum speedup for the executions. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|.

With one exception the predicted speed-up is very close to the real speed-up, i.e., the error is less than 1.5%. The exception is Ocean where the error is 6.2% on eight processors. However, the values between brackets show that the variations in the real speedup is rather large , and the predicted value is within the interval defined by the executions.

Table 1: Measured and predicted speed-ups.

Application/ Speed-up		2 processors	4 processors	8 processors
Ocean	Real	1.97 (1.97-1.98)	3.87 (3.85-3.89)	6.65 (6.42-7.11)
	Pred.	1.98	3.89	7.06
	Error	0.5%	0.5%	6.2%
Water-spatial	Real	1.99 (1.99-2.00)	3.95 (3.94-3.97)	7.67 (7.37-7.76)
	Pred.	1.98	3.91	7.56
	Error	0.5%	1.0%	1.4%
FFT	Real	1.55 (1.54-1.55)	2.14 (2.14-2.15)	2.62 (2.61-2.63)
	Pred.	1.55	2.14	2.63
	Error	0.0%	0.0%	0.4%
Radix	Real	2.00 (1.99-2.00)	3.99 (3.98-3.99)	7.79 (7.77-7.81)
	Pred.	1.98	3.95	7.87
	Error	1.0%	1.0%	1.0%
LU	Real	1.79 (1.78-1.80)	3.15 (3.12-3.15)	4.82 (4.74-4.90)
	Pred.	1.79	3.14	4.81
	Error	0.0%	0.3%	0.2%

Due to the recordings, the monitored uni-processor execution takes somewhat longer than an ordinary uni-processor execution of the program. However, our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, which was obtained for Ocean, was 2.6% of the total execution time. Another concern was the size of the log files. The largest log file, obtained for Ocean, was 1.4 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused any problems for these programs.

Programs with fine granularity generate more synchronization events, and thus larger log files, per time unit than coarse grained programs. The maximum number of events per second for our programs was 653 (Ocean). The uni-processor execution time for the five programs ranged from 60 seconds to 210 seconds. The size of the log files could become a problem for very long executions of fine grained programs.

We have done experiments with log files up to 15 MByte. Unfortunately the time required for obtaining the predicted speed-up values, and also the graph visualizing the behaviour of the program, increases for large log files.

5. A Simple Example

We use a producer-consumer problem to demonstrate how the tool can be used for improving the performance of an application. There are 150 Producers, each implemented by a thread, which inserts ten items in the buffer and then exits. There are 75 Consumers, picking one item each from the buffer. A semaphore is used to represent the number of items in the buffer, insertion and fetching of items is controlled by one mutex. The buffer size is large enough to avoid producer stalling as a result of a full buffer.

We began with making a log file on a uni-processor computer. After simulating the log file, we found that the program ran only 2.2% faster on 8 CPUs. To find out the reason of the poor performance, we use the Visualizer. A small part of the simulated execution is found in figure 6. In the execution flow graph we see that no threads are actually running in parallel. We also see that all threads are being blocked by a wait on a mutex, the arrow facing downwards. By clicking with the mouse on the arrows, we reach the conclusion that it is the same mutex causing the blocking for all threads. The mutex is the one that we use to lock the insertion and fetching.

When we have pinpointed the performance bottleneck we have to find a solution to our problem. One solution is to have 100 buffers with their own mutex locks. We keep a mutex for the whole buffer system to lock the small amount of time to check which buffer to insert the item in. We also have different mutexes for inserting and fetching.

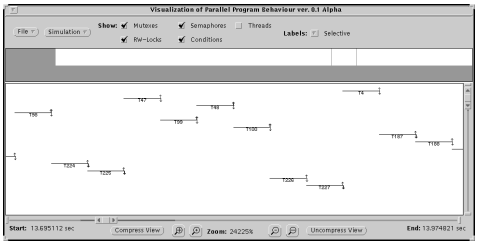


Figure 6: Parts of the execution of the initial program.

After making a new log of the improved program, we find that the program runs 7.75 times faster when using the simulated eight processor machine. A validation gives the speed-up of 7.90 on a real multiprocessor, thus the error in the prediction is only 1.9%. A picture of the simulated execution is found in figure 7. In the parallelism graph we can see that a larger number of thread are runnable but has no processor to run on. This is indicated by the high red part of the graph, and the constant low green part.

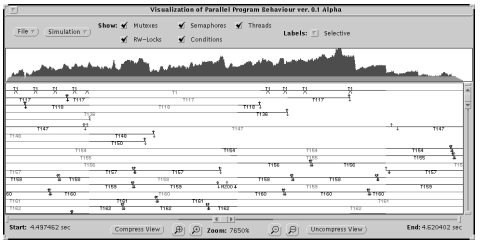


Figure 7: Simulated execution the improved program.

6. Discussion

Our approach is based on the assumption that the behaviour of the multithreaded program is (more or less) independent of the scheduling policy and the number of processors used for executing the program. Using the trace log file in a deterministic way when simulating and visualizing the execution may cause some problem. Some of them will be discussed below.

Conditions variables [17] are hard to simulate, since their behaviour depends on the value of an ordinary variable, which can not be traced by the Recorder. However, since it is common to use condition variables when implementing barriers, the simulator is designed to model the behaviour of a barrier as accurate as possible. The problem concerns the last thread that arrives at the barrier in the monitored execution. In the simulation that thread may be scheduled in a way that makes the thread arrive at the barrier before some other threads do. If the number of threads released during the recorded execution are less than those in the log file, the `cond_broadcast` will block the calling thread waiting for the correct number of threads to arrive at the barrier. Thus, the last thread arriving at the barrier releases all the waiting threads.

The primitive `thr_join` [17], waits until another (specific) thread has exited. It is possible to pass a wildcard to `thr_join`, meaning that the thread will wait for any thread the exit, which may not be the one that exited in the log file. Finally, the simulator does not consider the overhead for LWP context switches on a multiprocessor.

The Recorder can only be used when running one single LWP since the Recorder can not detect when an LWP's time slice is over and another LWP starts to execute. This makes it impossible to run a program with several threads where one thread executes in a tight loop during the whole execution since the loop will be the only one executing. Also having a thread spinning on a (ordinary, volatile) variable will cause a livelock for the same reason. Further reading on thread synchronization and scheduling can be found in [17]. Finally, our technique does not model I/O, and is therefore applicable only to CPU-intensive applications. We are currently working on solving this problem.

In the current implementation VPPB supports Solaris 2.X threads. However, the tool can easily be adjusted to support, e.g., POSIX threads [17] with only small modifications of the probes in the Recorder.

We have chosen not to use the recording facilities found in TNF [4], although the technique is similar to our Recorder. The main reason is that TNF uses a circular buffer to store the recorded information and thus information may be overwritten if the buffer is too small.

In [16] and [20] the authors stress the following issues: selective representation; integration between development time and run-time information; high-level and automated performance debugger; automated instrumentation of parallel programs; low overhead in monitoring program execution; and graphs and indices to expose performance bottlenecks. As mentioned throughout this paper all these requirements are met.

In parallel program development today there exists a number of tools, most ones with graphical (and even aural) displays. VPPB offers two different graphs, the execution flow graph and the parallelism graph. The execution flow graph is a commonly used graph, e.g., [5, 16]. We expect the parallelism graph to be very useful for detecting performance bottlenecks in large applications. A huge amount of graphs may cause more confusion than clarity of the performance problem as stated in [2]. Some other tools use statistical graphs [5]. The main problem with statistical graphs and data is that they often give only average values which are often useless since it is hard to identify when and where the program generated the statistics.

7. Conclusion

In this paper we describe a tool called VPPB, Visualization of Parallel Program Behaviour. The main goals are to predict the speed-up of a multithreaded application and to visualize the application's multiprocessor behaviour for the developer. The target programs are written in C or C++ and run on the Solaris 2.X operating system, an environment commonly used in both industry and academia.

Our approach relies on a monitored execution of the multithreaded application on a uni-processor. During that execution a log file is created containing all calls that the application made to the thread library. Then, the multiprocessor execution is simulated according to user supplied scheduling and hardware parameters. The result of the simulation is visualized graphically. The developer can then inspect the behaviour of the application as if it had been run on a multiprocessor without even having one.

The visualization of the execution is based on an execution flow graph along with some numeric data, a concept that previously has been shown to be successful [9]. The execution flow graph can be scrolled and zoomed, both in fixed steps and according to a specific time interval. A second graph shows the number of threads running at the moment, as well as the number of threads that are runnable but not running, i.e., the amount of available parallelism. VPPB gives the developer information enough to pin point the bottlenecks and correct them.

The tool also has a unique stepping facility, which gives the user of the tool a possibility to follow all operations on, e.g., a specific semaphore. Further, the tool supports an automatic mapping between an event in the execution flow graph and the source code line causing the event. It also starts an editor with the correct code line high-lighted.

We have validated the predicted speed-up using five benchmarks from the SPLASH-2 suite [19] and a multiprocessor with 8 processors. The predictions were found to be very accurate; for four of the applications the error was less than 2% as compared to a real multiprocessor execution. For the fifth application the error in the predicted speed-up was 6%. The intrusion made by the probes that collect the event log is very low; the execution time of the monitored application is prolonged by at most 3%.

References

[1] H. Chen, B. Shirazi, J. Yeh, H. Youn, and S. Thrane, "A Visualization Tool for Display and Interpretation of SISAL Programs," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 734-739, Oct. 1994.

[2] J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *Proc. Int'l Conf. on Supercomputing*, pp. 185-194, Jul. 1993.

[3] A. Hondroudakis, "Performance Analysis Tools for Parallel Programs," Edinburgh Parallel Computer Centre, The University of Edinburgh, Jul. 1995.

[4] S. Kleiman, D. Shah, and B. Smaalders, "Programming with threads," Prentice Hall, 1996, ISBN 0-13-172389-8.

[5] E. Kraemer and J. Stasko, "The Visualization of Parallel Systems: An Overview," *J. of Parallel and Distributed Computing*, Vol. 18, pp. 105-117, 1993

[6] S. Lei and K. Zhang, "Performance Visualization of Message Passing Programs Using Relational Approach," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 740-745, Oct. 1994.

[7] L. Lundberg, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications," *Proc. ISCA Int'l Conf. on Parallel and Distributed Computer Systems*, pp. 225-231, Sep. 1996.

[8] L. Lundberg and M. Roos, "Predicting the speed-up of multithreaded programs," *Proc. IEEE Conf. on High Performance Computing*, pp. 386-392, Dec. 1997.

[9] W. E. Nagel and A. Arnold, "Performance Visualization of Parallel Programs - The PARvis Environment," *Proc. 1994 Intel Supercomputer Users Group (ISUG) Conference*, pp. 24 - 31, May 1994.

[10] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter, "Extensible Parallel Program Performance Visualization," *Proc. Mascots '95*, 1995.

[11] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, "An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems," *Proc. 8th ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 527-533, 1995.

[12] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, "PARAVER: A Tool to visualize and Analyse Parallel Code," University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, Feb. 1995.

[13] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, "SunOS 5.0 Multithreaded Architecture," Sun Soft, Sun Microsystems Inc., Sep. 1991.

[14] S. R. Sarukkai and D. Gannon, "SIEVE: A Performance Debugging Environment for Parallel Programs," *J. of Parallel and Distributed Computing*, Vol. 18, pp. 147-168, 1993.

[15] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," MIT Press, 1989.

[16] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, 2(6):22-37, Nov. 1985.

[17] SunSoft, "Solaris Multithreaded Programming Guide," Prentice Hall, 1995.

[18] S. Toledo, "PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs," *Proc. 5th Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, Feb. 1995.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Annual Int'l Symp. on Computer Architecture*, pp. 24-36, Jun. 22-24 1995.

[20] J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurements, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit," *Software-Practice and Experience*, 25(4):429-461, Apr. 1995.

Paper II

Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads

Magnus Broberg, Lars Lundberg, and Håkan Grahn
13th International Parallel Processing Symposium, 1999

II

Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@ipd.hk-r.se

Abstract

Efficient performance tuning of parallel programs is often hard. We present a performance prediction and visualization tool called VPPB. Based on a monitored uni-processor execution, VPPB shows the (predicted) behaviour of a multithreaded program using any number of processors and the program behaviour is visualized as a graph.

The first version of VPPB was unable to handle I/O operations. This version has, by an improved tracing technique, added the possibility to trace activities at the kernel level as well. Thus, VPPB is now able to trace various I/O activities, e.g., manipulation of OS internal buffers, physical disk I/O, socket I/O, and RPC. VPPB allows flexible performance tuning of parallel programs developed for shared memory multiprocessors using a standardized environment; C/C++ programs that uses the thread package in Solaris 2.X.

1. Introduction

The thread concept in the Solaris 2.X operating system [10] makes it possible to write multithreaded C/C++ programs which can be executed in parallel. Having multiple threads does, however, not guarantee that a program will run faster on a shared memory multiprocessor. One major performance problem is that thread synchronizations may create serialization bottlenecks. Such bottlenecks are often hard to detect [6]. Removing serialization bottlenecks is referred to as *performance tuning*. In this context, it is important with tools that efficiently support the programmer in the performance tuning process, e.g., by visualizing the behaviour of, and thus the bottlenecks in, the parallel programs [2, 7, 8, 9, 11, 12, 14, 16].

In an earlier paper [2], we presented a performance prediction and visualization tool called VPPB (**V**isualization of **P**arallel **P**rogram **B**ehaviour). The target programs are written in C/C++ and run on the Solaris 2.X operating system, an environment commonly used in industry as well as in academia. Based on a monitored uni-processor execution, the VPPB system shows the (predicted) behaviour of a multithreaded Solaris program using any number of processors.

The first version of VPPB [2] could only monitor activities that took place in user space, i.e., only user level threads could be handled. Whenever a user level thread is blocked on an I/O operation, not only the user level thread is blocked, but also the corresponding kernel level thread (a.k.a. Light Weight Process, LWP) is blocked. The Solaris operating system then tries to dynamically create a new LWP to continue to execute some other user level thread. The first version of the tool could not manage several LWPs simultaneously and thus no blocking I/O.

The main contribution in this paper is an extension that overcomes the limitations above *by tracing the kernel level threads as well*. By recording all state transitions in the OS kernel for the LWPs, it is now possible to have several LWPs running at the same time. The tool can now handle various I/O activities, including physical disk I/O, socket communication, and RPC calls.

Validation has been done using ten benchmarks from the SPLASH-2 suite [15] and a skeleton of an I/O intensive commercial telecommunication application [6]. The simulated performance predictions were compared to real executions on a multiprocessor with eight processors. The maximum error of the predictions for those application are less than 10% for all applications and less than 4% for more than half of the applications.

The paper is structured in the following way. Section 2 gives a short overview of VPPB. In Section 3 the tracing part is described along with a discussion of how we sort and manage the collected data. The simulation part is described in Section 4. The validation part is found in Section 5 and the related work is found in Section 6. The paper concludes in Section 7.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in Figure 1. The developer writes the multithreaded program (a) in Figure 1.

When starting the monitored execution (b) on a uni-processor, the *Recorder* is automatically placed *between* the program and the standard thread library. Every time the program uses the routines in the thread library, the call

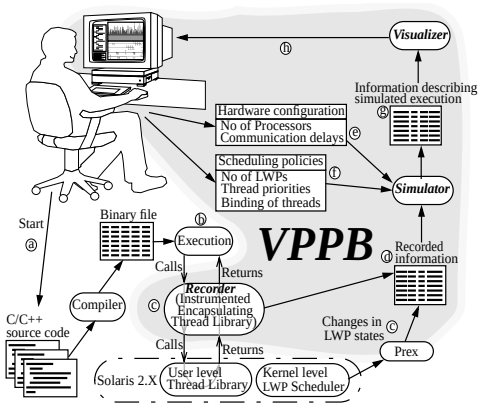


Figure 1: A schematic flowchart of the VPPB system.

passes through the *Recorder* (c) which records information about the call. The *Recorder* then calls the original routine in the thread library. Whenever an LWP does a state change the operating system informs a program called `prex`. `Prex` is a standard program on the Solaris platform used to create logfiles about LWP state changes. When the execution of the program finishes the two data files (from *Recorder* as well as `prex`) are sorted in time order and transformed into the file format (d) used in the first version of this tool. The recording is done without recompilation or relinking of the application.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 1. The simulator also takes the hardware configuration (e) and scheduling policies (f) as input. The output from the simulator is information describing the predicted execution (g).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (h). The Visualizer uses the simulated execution (g) as input. The simulated execution is shown as two graphs; one parallelism graph and one execution flow graph, as shown in Figure 2. It is possible for the developer to click on an 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.

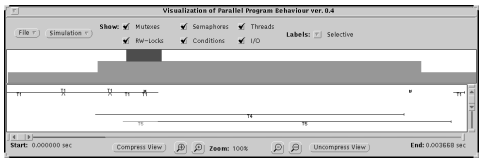


Figure 2: The execution flow and parallelism graphs.

3. Monitoring the Program Behaviour

The main contribution in this second version of VPPB is the ability to trace kernel threads in addition to user level threads. User level threads are non-preemtive and are executed on kernel threads. The kernel threads are preemtive and scheduled in a time-slice manner. This gave some implications in the first version of VPPB since only user level threads were traced and thus only one kernel thread was allowed. One example is the concept of spinning locks. Whenever a user level thread enters a spinning lock, the other threads can not pre-empt the thread. This makes the program to hang on the spinning lock. Another example is I/O. Whenever a user level thread blocks, the corresponding kernel thread is also blocked. The Solaris operating system will then, in order to continue execution of the multithreaded program, create new kernel threads for the other user level threads to execute on.

In Solaris, threads are used at two levels (see Figure 3) [13]. The application programmers use user-level threads for expressing parallel execution within a process. Kernel threads (a.k.a. LWPs) are used within the operating system kernel. The kernel knows nothing about user-level threads.

The LWPs are scheduled by the kernel in a preemtive round-robin fashion with timeslices from 20 milliseconds up to more than 200 milliseconds depending on the age of the LWP etc. A user level thread can be bound to a specific LWP or be floating around on the LWPs that are free.

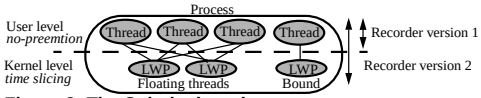


Figure 3: The Solaris thread structure.

3.1. Tracing user level threads

In order to trace the behaviour of the program when executed on a uni-processor, the Recorder dynamically inserts probes when the program starts. The probes are inserted at specific events, i.e., before and after calls to the thread library, and they do not affect the behaviour or function of the program. For each event, the probes record the following information: when an event has occurred; the type of event, e.g., locking of a mutex; which object the event concerns, e.g., the identity of the mutex being used; the identity of the thread generating the event; and the location of the event in the source code.

The user level tracing has been extended to capture some primitives in the `libc` library. In particular the `open`, `close`, `read`, `write`, and `fsync` primitives. To allow RPC calls [1] generated by, e.g., `rpcgen` [1], we also capture the primitives `putmsg`, `getmsg`, and `pipe`. One I/O operation will produce two different events in the log file; one event corresponding the CPU time needed to

execute the operation, and another event, called IO_wait, that corresponds to the time the thread was blocked.

One single I/O operation may consist of using the CPU several times with waiting times between. In order to keep the log file as small as possible all CPU time for one single I/O operation are concatenated into one. This concatenation is also done with the waiting time as well.

Our current implementation of the Recorder is based on TNF-probes [5]. The basic idea is to insert a new library between the program and the dynamically linked thread library. This is achieved by using the built-in facilities of Solaris with run-time linking and shared objects. The insertion is handled at program start-up by the run-time linker via the program `prex`. We also trace the source code location of the call to a probe.

3.2. Tracing kernel level threads

The first version of VPPB were restricted to have only one single LWP running at any time during the execution of the multithreaded program. The reason for this is that we could not trace the context switches of the LWPs in the kernel. This tracing can be done without any changes since Solaris 2.5 already have probes inside the kernel that trace this. The probes are implemented by using TNF-probes [5]. The super-user can extract the information by the `prex` command which gives as output a binary file which has to be merged with the ordinary user level trace from the Recorder. The approach that allows several LWPs to execute also makes it possible to trace programs with spinning locks. However, there is still a problem with poor performance prediction for programs with spinning locks as we will show in Section 5.1.

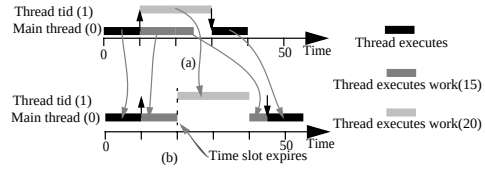
3.3. Basic merging and sorting of the two TNF-files

An example is used to illustrate the basic principles of how the sorting is done. The code is found in Figure 4. The main thread creates thread `tid` bound to an LWP. Then the main thread does some CPU bound work and then waits for thread `tid` and exits. The thread `tid` does some CPU bound work and exits.

The optimal execution on two processors can be found in Figure 5(a). Note that we have made some considerable simplifications in this example discussed at the end of this section, e.g., all thread primitives take no time to perform. The main thread starts at time 0 and creates thread `tid` at time 10. From time 10 to 25 the threads execute in parallel, then the main thread wants to join with thread `tid`. Thread `tid` will continue executing until time 30 and then exits. By this time the main thread can resume execution for the last 10 time units. Since the Recorder works on a uni-processor, the two threads can not execute in parallel, thus the execution may look like in Figure 5(b). At time 10 the main thread creates the thread `tid`, which waits for the CPU. At time 20 the time slot expires and the main thread leaves the CPU and the thread `tid` starts executing. When the thread `tid` has finished its execution, the main thread has 5 time units left until it reaches the join with thread `tid`. At time 45 the two threads has joined and the main thread executes until the end at time 55.

```
void *thr(void *in) {
  work(20); /* CPU bound work for 20 time units */
}
void main() {
  thread_t tid;
  thr_create(0, 0, thr, 0, THR_BOUND, &tid);
  work(15); /* CPU bound work for 15 time units */
  thr_join(tid, 0, 0); }
```

Figure 4: The source code of the example.



User level log file:

Time	LWP	Thread	Event
0	0	0	Start collect
10	0	0	Thread create 1
40	1	1	Thread exit
45	0	0	Thread join 1
55	0	0	Thread exit

Kernel level log file:

Time	LWP	Thread	Event
0	0	0	LWP starts running
20	1	1	LWP starts running
40	0	0	LWP starts running

(c)

Merged log file:

Length	Thread	Event
0	0	Start collect
10	0	Running
0	0	Thread create 1
15	0	Running
20	1	Running
0	1	Thread exit
0	0	Thread join 1
10	0	Running
0	0	Thread exit

(d)

Figure 5: Merging the user and kernel level log files.

The two log files generated during the execution are found in Figure 5(c). The user level log file consists of a start event at time 0 and at the same time in the kernel level log file the corresponding LWP starts running. The next event that occurs is the thread create event at time 10. However, the newly created thread `tid` does not start to execute until time 20, as seen in the kernel log file, when the corresponding LWP starts running. At time 40 the thread `tid` has finished, as indicated in the user level log file as well as in the kernel level log file, since the main thread continues to execute. At time 45, the threads joins as indicated in the user level log file, and finally, the main thread stops executing at time 55.

Now, we take a close look at the merging of the two log files into one single log file. During the merging we want to eliminate the concept of LWPs, thus only representing the behaviour of the user level threads. The resulting log file is shown in Figure 5(d). At time 0 we have the start event for the main thread and we see that its LWP is executing in the kernel log file in Figure 5(c). The LWP is running until time 20, i.e, the main thread is running 10 time units until it creates thread `tid` at time 10. At time 40 the thread `tid` has finished its execution, and we have to calculate how many time units it has executed as well as

the time the main thread executed. The LWP switch occurred at time 20, thus the main thread must have been running for 10 time units (from time 10 to 20) and the thread `tid` has been running for 20 time units (from time 20 to 40). The next event is at time 45. Since the main thread's LWP started executing at time 40, the main thread must been running for five time units before the join, and in order to make the merged log file compact, we add the previous running time for the main thread with this one. The resulting merged log file is found in Figure 5(d).

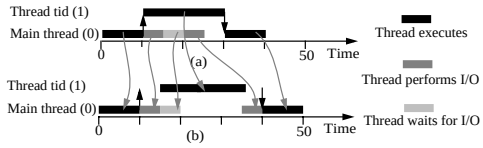
Other issues, must be considered, e.g., the kernel log file does not indicate which thread it is executing, the mapping must be done via the LWP identity. Also, the kernel threads are started before the corresponding user level thread starts and representing that time in the user level thread must be done in reverse order. The threads may float around on several LWPs, other processes may interact and put LWPs in both running and runnable states. Each user level event must have a start and stop time in order to measure how long the primitive took to execute. This made the user level log file to include nine lines in reality and the kernel level log file have 57 lines. Finally, things do not occur at the exactly same time in the kernel level log file as in the user level log file, and vice versa.

3.4. Merging and sorting the TNF-files with I/O events

We keep the example in Figure 4, but the main thread does not call `work(15)`, instead it writes to a file using the C standard primitive `write`. We have intentionally omitted the necessary `open` and `close` primitives in order to simplify the example. The optimal execution on two processors will look as in Figure 6(a). The main thread starts at time 0 and creates the thread `tid` at time 10. Immediately after, the main thread initiates a write to the disk. The writing of the file is finished at time 25 and the join with thread `tid` is reached at time 30. The main thread is finished at time 40. The thread `tid` is running between time 10 and 30, i.e., 20 time units.

The execution on a uni-processor system looks like in Figure 6(b), where we take a closer look at the write operation, time 10 to 45. The write operations include two important issues. The first issue is the time required by the processor to perform the write operation. The second is the time required by the disk to perform the write, meanwhile the processor may execute the other LWP and its thread. This is indicated in Figure 6(b) at time 15 since the thread `tid` starts executing when the main thread is waiting for the disk. Thread `tid` leaves the processor at time 35, and the write can be completed at time 40, the execution of the main thread continues and join with the thread `tid`, and finally ends at time 50.

The user level log file and the kernel level log file are found in Figure 6(c). In the kernel level log file at time 15 the main thread leaves the processor and the thread `tid` starts executing. At time 20, the waiting for the disk is over and the main thread's LWP can be put in the runnable state. At time 35 in the kernel level log file, the main thread's LWP starts executing to completion.



User level log file:

Time	LWP	Thread	Event
0	0	0	Start collect
10	0	0	Thread create 1
10	0	0	Write starts
35	1	1	Thread exit
40	0	0	Write ends
40	0	0	Thread join 1
50	0	0	Thread exit

Kernel level log file:

Time	LWP	Thread	Event
0	0	0	Starts running
15	0	0	Becomes blocked
15	1	1	Starts running
20	0	0	Becomes runnable
35	0	0	Starts running

(c)

Merged log file:

Length	Thread	Event
0	0	Start collect
10	0	Running
0	0	Thread create 1
0	0	Running
10	0	Write I/O
5	0	Wait I/O
20	1	Running
0	1	Thread exit
0	0	Thread join 1
10	0	Running
0	0	Thread exit

(d)

Figure 6: Merging the user and kernel level log files.

Whenever an I/O operation is performed, we collect two kinds of time information: The time that the I/O required the processor; and the time the I/O was waiting for the disk. These two times are represented as two events in the log file. The first event is the processor bound part of the I/O operation (write I/O) and the second event is the time the I/O operation was blocked (wait I/O) as shown in Figure 6(d). The I/O is the time from 10 to 40 in the user level log file. At time 15 in the kernel level log file the main thread becomes blocked. Thus, five time units execution on the CPU was needed before the actual writing to the (physical) disk. In the kernel level log file the main thread becomes runnable at time 20, i.e., the (physical) disk is ready and the wait for (physical) I/O is over. At time 35 the main thread start execute again as seen in the kernel log file. The write operation ends at time 40, i.e., the main thread needed yet five time units execution on the CPU in order to end the I/O operation. The two CPU bound parts of the I/O operation is added together. Otherwise the merging is performed as described in Section 3.3.

Note once again that the example is simplified, e.g., during one I/O operation the processor might have to wait several times for the (physical) disk and have to execute in between the waitings. Also, all state transitions between user space and kernel space are traced since each fundamental I/O operation is a system call. This make, together with the issues stressed earlier, that the user level log file consists of 15 lines in reality and the kernel level log file consists of 1340 lines in the case of writing a 10 million bytes large file. The merged log file consists of 15 lines.

4. Predicting the Program Execution

The Simulator mimics the scheduling in Solaris 2.5 [10]. The modeling of I/O follows the discussion in Section 3.1 with one part of the I/O to be considered as CPU bound and one part as waiting time. The Simulator first simulates the CPU bound time in a chunk just as any other event that only takes time. Then, the Simulator simulates the waiting for I/O to be completed, which is similar to a sleep primitive. However we simulate that only one thread can perform an I/O operation at the same time and only one I/O request, i.e., `IO_wait`, may be issued at the same time. This seems to mimic the OS quite accurate.

There are advantages and disadvantages of merging the parts of an I/O operation into one CPU bound part and one part that is representing the waiting time. The obvious advantage is that the log file will be shorter than if all the individual parts in were stored in the file. The obvious disadvantage is that we loose some information. However, this loss of information could actually be regenerated, to some extent, in the simulator by assuming that the internal I/O buffer within the kernel is of a particular size. Thus it is simple to calculate how many times the write operation must enforce a physical write, and thus a wait period, on the disk. This facility is not implemented and left to future development of the tool.

Much of the Simulator is kept from the first version of the tool [2], e.g., threads may be bound or unbound as well as the number of processor simulated is adjustable. Creating a bound thread is simulated to take 6.7 times longer than an unbound thread [13]. A synchronization on a semaphore takes 5.9 times longer with bound threads than unbound. The value is found in [13] and is incorporated in the simulator for semaphores as well as for mutexes, condition variables, and read/write locks.

5. Validation of the Predictions

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [15] and a skeleton of a telecommunication application that uses a lot of I/O in different manners. All executions were made on a Sun Ultra Enterprise 4000 with eight processors and 512 MByte memory. Our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, was obtained for Raytrace in the SPLASH-2 benchmark suite, was 31% of the total execution time. More than half of the log files caused less than 2% overhead. More than 75% of the log files were less than 1.5 Mbyte in size. The largest log file, which was obtained for Radiosity, was 19 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused any problems for these programs.

5.1. The SPLASH-2 benchmark suite

The programs that we use from the SPLASH-2 suite are: Ocean (with data set 514-by-514 grid), Water-Spatial (512 molecules, 30 time step), FFT (4M points), Radix (16M keys, radix 1024), LU (contiguous, 768x768 matrix, 16x16 blocks), Raytrace (teapot), Barnes (2048 bodies), Cholesky (tk29.O), FMM (2048 bodies), and Radiosity (Default, batch mode, en 0.1). Since the SPLASH-2 programs are designed to create one thread per physical processor, one log file was generated for each processor setup.

The first version of VPPB could only use five of the benchmarks [2]. This was because of spinning locks, as described in Section 3. Another issue is task stealing, i.e., a thread steals a waiting job from another whenever the stealing thread is idle. In the first version of the tool we could not handle several LWPs. This led to the result that the first thread that begun execute would steal all jobs from the other threads (which never got a chance to execute on the CPU). Thus, the recording showed that all work were done by one single thread and the others did nothing. When simulating this on a multiprocessor the load imbalance would be at its maximum. In this second version of VPPB we can handle several LWPs and thus the jobs may distribute better.

The 5 benchmarks we could use in the first version of the tool are discussed first. Then, we will look at the other benchmarks as well. Table 1 shows the measured and predicted speed-up for 5 programs from the SPLASH-2 benchmark suite we could use in the first version of VPPB. The real speed-up is the middle value of 5 executions of the program. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where $|-x| = |x| = x$, for all $x > 0$. As we can see in Table 1 the maximum error is 5.2%, which we consider to be a very low error. It is also an improvement with more than 16% as compared to the first version of the tool [2].

Table 1: Speed-ups for the first 5 benchmarks.

Application	2 processors			4 processors			8 processors		
	Pred	Real	Error	Pred	Real	Error	Pred	Real	Error
Ocean	1.95	1.97	1.0%	3.75	3.87	3.1%	6.47	6.65	2.7%
Water-spatial	1.97	1.99	1.0%	3.86	3.95	2.3%	7.27	7.67	5.2%
FFT	1.52	1.55	1.9%	2.06	2.14	3.7%	2.57	2.62	1.9%
Radix	1.99	2.00	0.5%	3.98	3.99	0.3%	7.91	7.79	1.5%
LU	1.82	1.79	1.7%	3.08	3.15	2.2%	4.72	4.82	2.1%

The benchmarks Barnes, Cholesky, FMM, and Radiosity could not be used in the first version of the tool since they use spinning locks. When a thread runs into a spinning lock, it will stay there for (in average) half a time slot until another thread can execute and possibly change the value of the lock. Raytrace uses a task stealing scheme, that might cause load imbalance if the tasks were executed

in the same order on a multiprocessor as on a uni-processor. These problems are clearly shown in Table 2 since the error may be as high as 62% (FMM).

Table 2: Speed-ups for the 5 benchmarks with *(without in italic)* spinning locks / load imbalance.

Application	2 processors			4 processors			8 processors		
	Pred	Real	Error	Pred	Real	Error	Pred	Real	Error
Raytrace	1.67	1.73	3.5%	2.19	2.69	18.6%	3.38	3.73	9.4%
Raytrace	*1.71*	*1.72*	*0.6%*	*2.42*	*2.50*	*3.2%*	*3.24*	*3.28*	*1.2%*
Radiosity	1.74	1.91	8.9%	3.09	3.72	16.9%	5.25	6.20	15.3%
Radiosity	*1.91*	*1.86*	*2.7%*	*3.63*	*3.75*	*3.2%*	*5.97*	*6.31*	*5.4%*
Barnes	1.72	1.95	11.8%	2.85	3.34	14.7%	4.28	5.77	25.8%
Barnes	*1.97*	*1.97*	*0.0%*	*3.57*	*3.38*	*5.6%*	*5.84*	*5.33*	*9.6%*
Cholesky	1.37	1.62	15.4%	1.98	2.31	14.3%	2.42	2.89	16.3%
Cholesky	*1.59*	*1.62*	*1.9%*	*2.21*	*2.31*	*4.3%*	*2.80*	*2.85*	*1.8%*
FMM	1.58	1.90	16.8%	1.99	3.50	43.1%	1.98	5.19	61.8%

The error was caused by the time a thread was bound to stay spinning on a lock until the time slice was over. By increasing the number of threads executing on a uni-processor, and thus causing more threads spin on the spinning locks, we exaggerated that behaviour. Another way of testing this is to, for each iteration in the spinning loop, voluntarily give up the processor and thus decrease the overhead with spinning locks. Tests conducted on Cholesky and FMM confirmed our thoughts.

The spinning locks in Radiosity, Barnes, and Cholesky could easily be replaced by semaphores and mutexes. The spinning locks in FMM could not be replaced easily and we did not change that program, since we do not want to perform too large changes to the programs. The results after replacing the spinning locks with blocking locks are found as *italic* rows in Table 2. Further, Raytrace has been slightly modified to avoid task stealing. As can be seen, the error drop dramatically, in most cases with at least 83%. The maximum error is now 9.6%.

5.2. The Billing Gateway

None of the SPLASH-2 programs contained any (significant) I/O, and in order to validate the I/O we used a telecommunication application developed by Ericsson Software Technology called Billing Gateway (BGw) [6]. We used a skeleton version of the BGw for our validation because the real BGw has an advanced graphical user interface, and that the application uses customer adjusted data input formats which made it hard for us to find proper loads. The skeleton was created together with Ericsson to mimic the characteristics of the original BGw and ended up consisting of around 1000 lines of C++ code. The original BGw consists of about 100,000 lines of code.

A principal sketch over the BGw (skeleton) is found in Figure 7. The BGw (skeleton) works as a kind of filter. The Socket receivers get the information to be filtered through a socket. The information chunk is 1 Mbyte large and consists of integers. As soon as all data are received the information is stored on disk, the disk is synchronized, i.e., all data is physically written to disk, and the receiver is ready for the next chunk of data. The Sorter reads the file created by the Socket receiver and puts all the integers in a binary tree. As workload all integers are converted to floating points and back to integers again during a traversion of the tree, this is repeated 1024 times. Finally, the Sorter stores the odd integers into one file and the even integers into another file. As previously the information on the disk are synchronized. The Consumers then read the data and discards it. The skeleton we use has eight Socket receivers, eight Sorters, and 16 Consumers as shown in Figure 7. Each Socket receiver were fed with two 1Mbytes chunks.

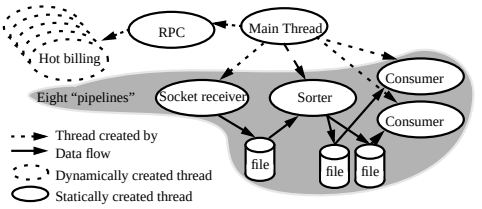


Figure 7: The organization of the BGw skeleton.

The skeleton is also able to consider hot billing which, as in the original BGw, is managed by RPC. Once an RPC call is made to the BGw, a new thread is created to process the data. The processing of the hot billing data is the same as described above. However no data are stored on disk. The skeleton received 5 RPC calls of 5 kbytes each of hot billing data. The skeleton performs, on average, approximately 800 Kbytes per second of I/O traffic on a Sun Enterprise 4000 with eight processors.

The result of the BGw skeleton can be found in Table 3. As can be seen, the predictions for this I/O application is very accurate, at most with 6.1% error.

Table 3: Speed-ups for the BGw skeleton.

2 processors			4 processors			8 processors		
Pred.	Real	Error	Pred.	Real	Error	Pred.	Real	Error
1.99	1.98	0.5%	3.98	3.75	6.1%	6.44	6.17	4.4%

6. Related Work

Some performance visualization tools show the behaviour of one particular monitored multiprocessor execution of the parallel program [3, 7, 16]. The problem with this approach is that there is no support for detecting bottlenecks which appear on another number of processors. TNF probes are used for a similar purpose in [3]. Another

tool [4] focus on the contention in multithreaded programs.

There are a number of tools, [8, 9, 14, 12, 11], which make it possible to visualize the (predicted) behaviour of a parallel program using any number of processors. However, these tools are either developed for message passing systems or for non-standard programming environments.

7. Conclusion

In this paper we have presented an improved version of the VPPB tool. This tool makes it possible to predict the speed-up and visualize the behaviour of a multithreaded C/C++ application using the Solaris 2.X thread package. It is a common environment in both industry and academia.

Our approach is based on a monitored uni-processor execution of the multithreaded program. Based on recordings from this execution and some parameters describing the target multiprocessor, the behaviour and execution time of the multithreaded program is predicted. The first version of the tool was not able to handle I/O. The main improvement in this version is that I/O can be handled because we monitor kernel threads as well. The monitoring is performed using the TNF probes in Solaris.

We have validated the predicted speed-up using the SPLASH-2 suite [15], an I/O intensive skeleton of a telecommunication application, and a multiprocessor with eight processors.

The first version of the tool could only handle five of the applications in the SPLASH-2 suite. The current version of the tool can handle all applications in the test suite. The maximum error in the speed-up predictions for the first five applications is 5%; in most cases the error is much smaller. The other applications in SPLASH-2 could not be handled by the first version of the tool because they contain spinning locks. These applications can now be handled. The maximum speed-up prediction error for these applications is relatively large, up to 62%. However, if we replace spinning locks with semaphores and mutexes the predictions become better. The maximum error was less than 10%, and more than half of the predictions had an error of less than 4%.

To validate the speed-up predictions for applications heavily depending on I/O we used a skeleton version of a large commercial telecommunication application. This validation shows that the simulation of I/O is very accurate; the maximum error is only 6%.

In the current version of the tool, we can handle any multithreaded Solaris program. Our technique requires no modification of the source code of the multithreaded program. The recording overhead is small for most applications, e.g., less than 2% of the total execution time for more than half of the SPLASH-2 applications.

References

[1] J. Bloomer, “Power Programming with RPC,” O’Reilly & Associates, Inc., ISBN 0-937175-77-3, 1992.

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

[3] B. Cantrill and T. Doeppner, “ThreadMon: A Tool for Monitoring Multithreaded Program Performance,” *Proc. 30th Hawaii Int’l Conf. on System Science*, pp. 253-265 Vol. 1, 1997.

[4] M. Ji, E. Felten, and K. Li, “Performance Measurements for Multithreaded Programs,” *Performance Evaluation Review*, vol. 26, no. 1, pp. 161-170, Jun. 1998.

[5] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996, ISBN 0-13-172389-8.

[6] L. Lundberg and D. Häggander, “Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor,” *1998 Int’l Conf. on Parallel Processing*, pp. 262-269, 1998.

[7] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter, “Extensible Parallel Program Performance Visualization,” *Proc. Mascots ‘95*, pp. 205-211, 1995.

[8] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” *Proc. 8th ISCA Inte’l Conf. on Parallel and Distributed Computing Systems*, pp. 527-533, 1995.

[9] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to visualize and Analyse Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, Feb. 1995.

[10] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, “SunOS 5.0 Multithreaded Architecture,” Sun Soft, Sun Microsystems Inc., Sep. 1991.

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

[12] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” *IEEE Software*, 2(6):22-37, Nov. 1985.

[13] SunSoft, “Solaris Multithreaded Programming Guide,” Prentice Hall, 1995.

[14] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” *Proc. 5th Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, pp. 396-405, Feb. 1995.

[15] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proc. 22nd Annual Int’l Symp. on Computer Architecture*, pp. 24-36, Jun. 22-24, 1995.

[16] J. Yan, S. Surukkai, and P. Mehra, “Performance measurements, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit,” *Software-Practice and Experience*, 25(4):429-461, Apr. 1995.

Paper III

Performance Tuning of Multithreaded Applications for Multiprocessors by Cross-Simulation

Magnus Broberg

ISCA 13th International Conference on Parallel
and Distributed Computing Systems, 2000

III

Performance Tuning of Multithreaded Applications for Multiprocessors by Cross-Simulation

Magnus Broberg
Department of Computer Science, University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden

Abstract

Performance tuning of a parallel application is often hard. The use of standards, such as POSIX threads, makes it possible to move a multithreaded application from one platform to another. Doing performance tuning for many platforms is even tougher since the implementation of the standards may vary on different operating systems. The developer needs tools for analysing how the application will behave on different operating systems in order to do adequate performance tuning.

In this paper we present a technique based on cross-simulation that will solve the issues above. The technique uses a monitored execution of a multithreaded application on a single processor workstation running the Solaris operating system. Then the technique, which has been implemented in a tool, simulates a multiprocessor with an arbitrary number of processors running either Solaris or Linux. The tool then displays the behaviour of the application on the selected target configuration.

Validation, using a subset of the SPLASH-2 benchmark suite, shows that the tool predicts speed-ups correctly. The average error in the predicted speed-up when simulating Linux is 5.8%. All this can be done using the ordinary Solaris workstation on the developer's desk, without even having a multiprocessor.

Key words: Multithreading, cross-simulation, multiprocessor, visualization, performance prediction.

1. Introduction

Writing parallel applications for multiprocessors is often hard. In many cases the reason for using a multiprocessor is to achieve more computing power for the application. Doing performance tuning of a multithreaded application for a multiprocessor is an important but tedious task and few tools are available for the developer. Most tools require the application to be executed on the target multiprocessor, e.g., [5, 8, 10, 19].

When introducing standards, such as POSIX threads [3], the aim is to make applications portable. This makes it possible to move one application from one operating system, e.g. Solaris, to another operating system, e.g. Linux. However, it is not obvious that an application tuned for one operating system will run efficiently on another operating system. Different characteristics for the operating systems may lead to different tuning or trade-offs when tuning the application for both operating systems.

The developer then has to tune the application for several operating systems, using different tools for the different operating systems. The developer must also have access to all combinations of multiprocessors and operating systems in order to do the performance tuning. Not only a tedious task to do, in many cases it is impossible (or expensive) to have all the machines needed. Also the developers would like to know if their application will handle larger multiprocessors than the ones available today and, thus, avoid rewriting the application when a larger multiprocessor is available.

In this paper we present a tool called VPPB (Visualization of Parallel Program Behaviour). The tool is capable of executing a multithreaded application (using POSIX Threads) on a single processor workstation with Solaris and shows how the execution behaviour if the application was executed on an SMP (Symmetric MultiProcessor) with an arbitrary number of processors, running either Solaris or Linux. The predictions have good accuracy. The tool has previously been used for predictions on Solaris for Solaris-threads [1, 2, 16] and has now been extended to cover the major parts of POSIX threads as well. The main thing is, however, that VPPB now also can mimic the Linux 2.2 operating system. Based on an execution of an application on a single processor Solaris workstation, the tool is able to predict the behaviour of the application on a multiprocessor running Linux. Thus, the tool is able to perform cross-simulation. We use the term cross-simulation when a program is monitored on one operating system and then simulated for another (target) operating system. The performance tuning of the application can be done on the developer's ordinary workstation without the need for a multiprocessor.

The rest of this paper is as follows. In Section 2 we give an overview of the VPPB system and in Section 3 Solaris, Linux, and POSIX threads are briefly discussed. Section 4 shows the validation of the predictions and Section 5 shows that the same application will execute differently on different operating systems. Section 6 discusses

the results and points at some future work, the conclusions are found in Section 7.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in Figure 1. The developer writes the multithreaded program (a) in Figure 1, compiles it and an executable binary file is obtained. After that, the program is executed on a single 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. Also some systems calls, e.g., read() are monitored in the same way for the clib 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.

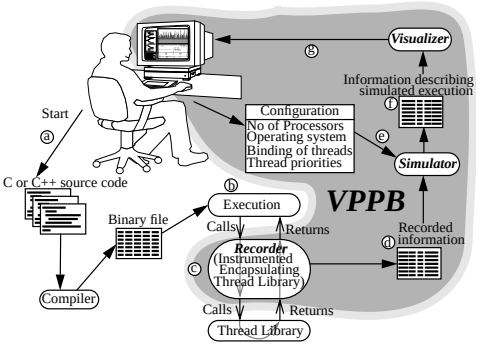


Figure 1: A schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 1. The simulator also takes the configuration (e) as input, such as the target operating system, number of processors, etc. The output from the simulator is information describing the predicted execution (f).

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 Gannt 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 [8] and POSIX threads [3] on the Solaris 2.X operating system. The Simulator is able to simulate both Solaris and Linux.

3. Some Key Differences Between Solaris and Linux

Both Solaris and Linux 2.2 implement the POSIX thread interface [3]. However, there are differences between the implementations. Solaris uses a two-layered approach illustrated in Figure 2 [7, 13]. At the user level there are user level threads. These threads are executed in a non preemptive way. This means that the thread must, in some way, voluntarily give up the execution in favor of another thread. This could be done explicitly or implicitly by calling a synchronization primitive that blocks the thread or releases a previously blocked thread with higher priority.

The kernel level threads (also known as LightWeight Process, LWP [16]) are scheduled by the kernel in a pre-emptive fashion. The scheduling is priority based, with a round robin policy on each priority level. The priority is changed over time, the longer time an LWP executes the lower priority and longer time slices it gets. Each CPU has its own run queue. In order to migrate an LWP to another CPU the LWP must have been in the queue for some predefined time without being selected by the current CPU. The user level threads are executed by the LWPs. This means that the kernel only schedules LWPs, the user level threads are not seen by the kernel. From the user level threads' point of view the LWPs could be seen as virtual CPUs. A user level thread might be bound to a specific LWP, i.e., the user level thread will in that case (indirectly) be scheduled in a pre-emptive manner. Non-bound user level threads will execute on the LWPs that are not bound to any thread. There could be many LWPs to serve this kind of user level threads.

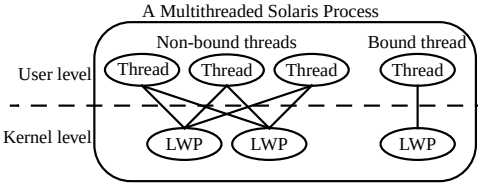


Figure 2: The thread model in Solaris. A bound thread may only execute on the LWP that it is bound to, whereasa non-bound thread may execute on any LWP that is not bound to a user level thread.

Linux [9] uses a single-layered approach, similar to merging the Thread and LWP concepts in Solaris together into one single unit. The creation of threads is made by cloning the parent thread, using the system call clone. This is quite similar to fork, but the cloned threads share the same address space, open files, etc. However, each thread gets its own process id and is scheduled as a process. The scheduling is pre-emptive and time-sliced, as for any other process in Linux. The time-slices are fixed (210 milliseconds). Linux calculates a goodness-value in order to select which thread/process to execute next. This goodness-value represents two things, first it represents how long the thread has executed in its current time slice. A thread that has been executed a small part of its time slice will be given a higher goodness value than threads that have executed a longer part of their time slices. The second thing is that a thread is given higher priority for the processor it previously ran on in order to run on the same processor again. This is done by increasing the goodness value for the thread when the processor it previously executed on looks for a new thread to execute. Newly awakened threads can force another thread from a processor when the newly awakened thread has a higher goodness value than the other thread.

There are differences between synchronizations as well. In Linux all locks have ordered queues. When a thread releases the lock, the first thread in the queue is given the lock and put in the running queue. In Solaris, there is no ordered queue (for guaranteed). When a thread releases a lock, it may lock it again, before any other thread (both those waiting in the queue or other) has been able to grab the lock.

4. Validation of the Predicted Execution Times

4.1. The SPLASH-2 benchmark suite

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [18]. The applications that we used from the SPLASH-2 suite are listed with the data set in Table 1. All applications that we used are from the scientific and engineering domain.

All executions were made on a Sun Ultra Enterprise 4000 with 8 processors and 512 MBytes memory. The operating systems in this validation was Solaris 2.6 and Linux 2.2 (RedHat 6.1). The compiler used for both Solaris and Linux was the egcs-1.1.2 (a.k.a. gcc 2.91.66). Since the SPLASH-2 applications are designed to create one thread per physical processor, one log file was made for each processor setup when using the Recorder. This in order to not change the intentions of the benchmarks when verifying them on a real multiprocessor. Thus, 9 applications running each on 4 different CPU setups generated a total of 36 log files. The benchmarks were modified in order to remove spinning locks and task stealing. When a thread runs into a spinning lock, it will stay there for (in average) half a time slot until another thread can execute and possibly change the value of the lock. In Solaris the time slot may be up to 0.2 seconds and this affects the execution time of the application when executing on a single processor machine during the logging. Raytrace uses a task stealing scheme, that might cause load imbalance if the tasks were executed in the same order on a multiprocessor as on a single processor. A further discussion on these matters is found in [1].

Table 1: The parallel applications together with the data set sizes we used.

Application	Data set size/Input data
Ocean (contiguous)	258-by-258 grid
Water-Spatial	512 molecules, 30 time steps
FFT	4M points
Radix	16M keys, radix 1024
LU (contiguous)	768x768 matrix, 16x16 blocks
Raytrace	balls4
Barnes	2048 bodies
Cholesky	tk29.0
Radiosity	Default, batch mode, en 0.1

4.2. Validation results

Table 2 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suit on the Solaris platform. The real speed-up is the middle value of 5 executions of the application. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where $|-x| = |x| = x$, for all $x > 0$.

Due to the recordings, the monitored single processor execution takes somewhat longer than an ordinary single processor execution of the application. However, our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, which was obtained for Radiosity, was 16.4% of the total execution time, but still more than half of the 36 log files caused less than 0.5% overhead. Another concern was the size of the log files. 75% of the log files were less than 2 MBytes in size. The largest log file, which was obtained for Radiosity, was 20.4 MBytes. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused problems for these applications.

Table 3 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suite

Table 2: Measured and predicted speed-ups for the benchmark applications on Solaris 2.6.

Application		2 proc.	4 proc.	8 proc.
Ocean	Real Speed-up	1.98	3.67	4.39
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.0%	8.7%	0.9%
Water-spatial	Real Speed-up	1.99	3.93	7.41
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	2.5%	2.3%
FFT	Real Speed-up	1.58	2.22	2.76
	Pred. Speed-up	1.59	2.23	2.80
	Error	0.6%	0.5%	1.4%
Radix	Real Speed-up	1.99	3.96	7.66
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.0%	0.0%	1.7%
LU	Real Speed-up	1.78	3.02	4.45
	Pred. Speed-up	1.78	3.00	4.50
	Error	0.0%	0.7%	1.1%
Raytrace	Real Speed-up	1.88	2.97	4.86
	Pred. Speed-up	1.88	2.94	4.83
	Error	0.0%	1.0%	0.6%
Radiosity	Real Speed-up	1.85	3.60	5.78
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	5.0%	1.9%
Barnes	Real Speed-up	1.94	3.56	6.35
	Pred. Speed-up	1.93	3.57	5.97
	Error	0.5%	0.3%	6.0%
Cholesky	Real Speed-up	1.60	2.30	2.94
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.4%

Table 3: Measured and predicted speed-ups for the benchmark applications on LINUX 2.2.

Application		2 proc.	4 proc.	8 proc.
Ocean	Real Speed-up	1.99	3.75	5.77
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.5%	10.7%	24.6%
Water-spatial	Real Speed-up	1.99	3.95	7.75
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	3.0%	6.6%
FFT	Real Speed-up	1.71	2.56	3.40
	Pred. Speed-up	1.59	2.23	2.80
	Error	7.0%	12.9%	17.6%
Radix	Real Speed-up	1.98	3.93	7.17
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.5%	0.8%	8.6%
LU	Real Speed-up	1.81	3.11	4.80
	Pred. Speed-up	1.78	3.00	4.50
	Error	1.7%	3.5%	6.2%
Raytrace	Real Speed-up	1.82	2.85	4.41
	Pred. Speed-up	1.88	2.94	4.83
	Error	3.3%	3.2%	9.5%
Radiosity	Real Speed-up	1.85	3.75	5.05
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	8.8%	16.6%
Barnes	Real Speed-up	1.94	3.51	6.03
	Pred. Speed-up	1.93	3.57	5.98
	Error	0.5%	1.7%	0.8%
Cholesky	Real Speed-up	1.60	2.30	2.93
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.0%

on the Linux 2.2 platform. The columns are the same as in Table 2.

The mean error for Solaris was 1.6% and when simulating the Linux platform the mean error was less than 5.8%. Thus, the error on the Linux platform is then on average 3.6 times larger than on the Solaris platform.

4.3. Study of the application with the largest error, Ocean

There are three applications in Table 3 that have large errors in the predictions. These are Ocean, FFT, and Radiosity. Ocean has the largest error and we will focus on that application for this study.

The first thing to notice is the total CPU time needed to execute the application (without the Recorder) behaves quite differently on Solaris and Linux. Since the monitoring is done on Solaris all the overhead, etc., on Solaris is incorporated in the monitoring. Thus, if Solaris behaves differently than Linux, the estimation can be no good. The CPU time needed to execute Ocean is shown as the first row in Table 4. The values are normalized to the case for 1 processor for easy comparison reasons. As can be seen, Solaris increases the needed CPU time with up to twelve percent when executing the 8 threaded Ocean. On the other hand, Linux needs three percent less CPU time to execute Ocean with 8 threads than with one thread. The difference between Solaris and Linux is then 15.5% (1.12 / 0.97).

By increasing the data set for Ocean to 514 and 1026, these differences decrease as shown in the two last rows in

Table 4: Normalized CPU time required to execute the OCEAN benchmark with different data sets on Solaris and Linux.

Data set	Operating System	1 Thread	2 Threads	4 Threads	8 Threads
258	Solaris	1.00	0.99	1.03	1.12
	Linux	1.00	0.96	0.97	0.97
514	Solaris	1.00	1.02	1.04	1.05
	Linux	1.00	1.00	1.01	1.00
1026	Solaris	1.00	1.00	1.00	1.01
	Linux	1.00	1.01	1.00	1.00

Table 4, to 5.0% and 1.1%, respectively for the case with 8 threads. It is then most likely that the predictions also will be more accurate for Linux as the needed CPU time does not differ. In Table 5 the predictions for Ocean on Linux is shown. As assumed the error in the predictions drops as the data set increases. The measured CPU time can thus act as an indicator of the degree of reliability in the prediction.

Table 5: Ocean on Linux with different data sets. The compensated values are given in *italic*.

Data set / Speed-up		2 Processors	4 Processors	8 Processors
258	Real	1.99	3.75	5.77
	Pred.	1.90 *1.96*	3.35 *3.56*	4.35 *5.02*
	Error	4.5% *1.5%*	10.7% *5.1%*	24.6% *13.0%*
514	Real	1.84	3.67	5.27
	Pred.	1.94 *1.98*	3.71 *3.82*	4.51 *4.74*
	Error	5.4% *7.6%*	1.1% *4.1%*	14.4% *10.1%*
1026	Real	1.96	3.83	6.93
	Pred.	1.90 *1.88*	3.74 *3.74*	6.66 *6.73*
	Error	3.1% *4.1%*	2.3% *2.3%*	3.9% *2.9%*

The measured CPU time may not only act as an indicator, it could also be used to compensate the prediction. In the case of Ocean with data set 258 there is a difference of 15.5% between Solaris and Linux. Thus, the monitored execution on 8 processors was 15.5% longer on Solaris than on Linux and this made the predicted execution for Linux 15.5% longer as well, assuming an even distribution of the overhead over the whole execution. If the predicted execution was 15.5% longer, the predicted speed-up will only be 86.6% (1 / 1.155) compared to the speed-up without the differences in CPU time. Thus, by adding 15.5% more speed-up we are able to compensate for the differences in CPU time. The predicted speed-up is 4.35 and with a 15.5% increase it will be 5.02. The latter is much closer the real measured value of 5.77.

The result of calculating in the same way for the other data sets and number of processors is shown in Table 5 in *italic*. The *average* error in Table 4 has decreased from 7.8% to 5.6% due to this compensation. In three cases the predictions become worse than without compensation, however, those errors are still within the range of errors for the Solaris prediction in Table 2 as well as the errors reported in [1].

5. Scheduling Differences

When dealing with performance debugging, the speed-up metric does not give all the information needed. If the speed-up is not as the desired speed-up there are some kind of performance bottlenecks in the application. The VPPB system has, as mentioned in Section 2, the ability to show the execution flow as a Gannt diagram. This diagram can help the developer understand where the bottlenecks are and how to remove them.

In Section 4 there is very little difference between the predicted speed-up for the Solaris platform and the Linux platform. Although the speed-up is similar, the execution flow may not be the same. To illustrate that issue Figure 3 shows the same execution segment of the Barnes benchmark. Each horizontal line in the figure represents an executing thread over time. Different symbols indicate different events, e.g., an arrow facing downwards represents a locking operation. Different colours (appear as different gray shades in black and white printing) are used for mutexes, semaphores, etc. Though the execution flow is quite different in Figure 3 the source code is the same. This shows that the execution flow is different when using different operating system, and thus the performance bottlenecks may also differ between the operating systems.

6. Discussion and Related Work

6.1. Comments on the validation

Validation of the cross-simulation was done on a Sun Enterprise 4000 with 8 CPUs by executing 9 benchmarks from the SPLASH-2 benchmark suite. Only three applications had larger than ten percent error in the predicted speed-up. A further study of the application with the largest error (Ocean) shows that the error is reduced when the data set grows.

As the data set increases the overhead for executing Ocean on Solaris decreases. Since this overhead is included in the monitored data used for the cross-simulation the simulation results for small data sets may not be very accurate. By compensating the predictions with the measured overheads, the error of the predictions was reduced with up to nearly a factor of two.

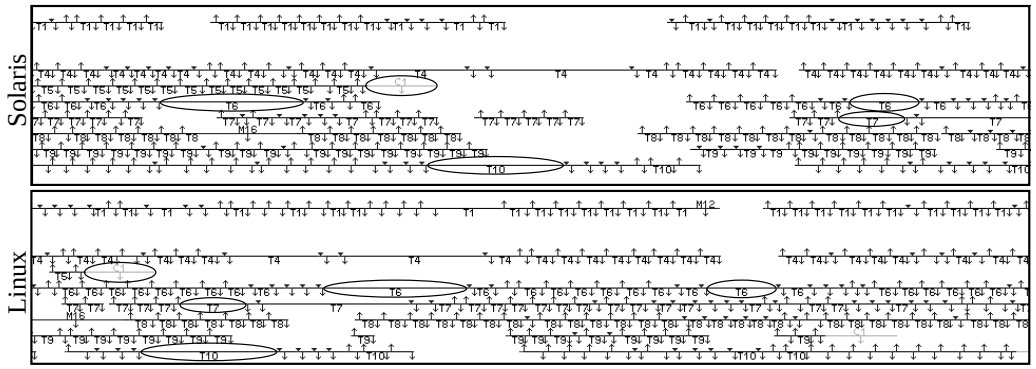


Figure 3: Execution flows for Barnes. Solaris is the upper graph, Linux is the lower. The five ovals in the Solaris graph indicates the same events as the five ovals in the Linux graph.

The reason for the increased usage of CPU time on Solaris, when increasing the number of threads in the Ocean application has not been found. A further study of the reason for this behaviour on Solaris is considered to be future work. Understanding why the error for the benchmarks FFT and Radiosity is almost twice as large as the largest error on Solaris could also be interesting future work.

Currently the VPPB system implements all the POSIX thread primitives that are common between Solaris and Linux except the thread cancelling primitives. The semaphore primitives are supported, although semaphores are not a part of the POSIX threads, but originate from POSIX.1b.

6.2. Other tools

There are a number of tools, shown in Table 6, which make it possible to visualize the (predicted) behaviour of a parallel application using any number of processors. However, these tools are either developed for message passing systems or for non-standard programming environments.

Only a few tools, PARAVER and SIEVE, can do some kind of cross-simulation. The PARAVER tool uses the DIMEMAS simulator [4]. DIMEMAS is a simulator for distributed memory multiprocessors, and the ability to reconfigure in order to mimic different operating systems is limited. In [4] there is no validation of the predictions at all. In SIEVE all simulation is performed by scripts working like macros on a spread-sheet, where the spread-sheet is the trace file. By supplying different scripts different operating system can be simulated. The SIEVE system does not address the issue of data monitoring. In [14] there is no validation of the predictions at all. Other tools, such as Tmon [6] is capable of tracing multithreaded applications on single processors, but not on multiprocessor machines.

7. Conclusion

POSIX threads are used to make multithreaded applications portable. However, the implementation of POSIX threads differs for different operating systems. Thus, an application will not always have the same performance and behaviour on different operating systems. Tuning multithreaded applications for a multiprocessor is hard. Tuning applications for good performance on several operating systems is even harder. Most tools use a real execution of the multithreaded application on a given operating system in order to give the developer support in the performance tuning, e.g., [5, 8, 10, 19]. It is often impractical and expensive to have several multiprocessors in order to run different operating systems.

Table 6: Comparison of some visualization tools similar to VPPB.

Name	Platform / *Language*	Cross-simulation	Comment
CHIP^{3}S [11]	Mathematica / *CHIP3S*	No	Pseudo language
PARAVER [12]	Any PVM3 platform	Yes	Message passing
PERFSIM [17]	TMC CM-5 / *CM-Fortran*	No	Limited visualization
PIE [15]	VAX 11/780 and 784, MicroVAX / *MP with Pascal*	No	Pseudo language
SIEVE [14]	BBN GP1000 / *pC++*	Yes	Hard to use the scripts

In this paper we have presented a tool, called VPPB, that based on an execution of a multithreaded application on an ordinary single processor Solaris workstation can predict the behaviour of the application on a multiprocessor with an arbitrary number of processors, running Solaris or Linux 2.2. The predictions are accurate, with a mean error of 5.8% for the predicted speed-ups for Linux, and even better for Solaris. The validation was based on 9 of the benchmarks from the SPLASH-2 benchmark suite on a Sun Enterprise 4000 with eight processors.

During a detailed study of the Ocean benchmark we have shown that it is possible to find an indicator that shows how reliable the predictions are. The indicator is based on the CPU time required to execute an application with a different number of threads. This indicator works on a single processor workstation with Solaris.

The Ocean benchmark also shown that the indicator can be successfully used to compensate the predictions in order to get more accurate predictions. With Ocean the error in the predictions was reduced with up to almost a factor of two. The average error decreased with 28%. Both the indicator and compensator technique was based on a single application and will need further study and, possibly, refinement.

Thus, we have shown that it is possible to use the described tool to predict the behaviour of a multithreaded application on a multiprocessor with either Solaris or Linux, with the means of a single processor workstation running Solaris.

8. Acknowledgments

We would like to thank Henrik Persson for rewriting the simulator for the Solaris threads in order to achieve a much faster simulator. This simulator was used as the base when adding ability for POSIX threads as well as support for the Linux platform. We would also like to thank Lars Lundberg and Håkan Grahn at the University of Karlskrona/Ronneby, Sweden, for reading earlier versions of this paper and giving very useful feedback in order to improve the paper.

References

[1] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads,” Proc. 13th Int’l Parallel Processing Symp., San Juan, Puerto Rico, pp. 407-413, 1999.

[2] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs,” Proc. 12th Int’l Parallel Processing Symp., Orlando, USA, pp. 770-776, 1998.

[3] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, 1997, ISBN 0-20-163392-2.

[4] S. Girona, T. Cortes, J. Labarta, V. Pillet, A. Peres, and E. Lopez, “Deriverable OPS4A of the project Basic research APPARC, Effect of short term scheduling on message passing multiprogrammed systems,” http://www.wi.leidenuniv.nl/CS/HPC/apparc-deliverables/OpS4a.html, 1994.

[5] M. Heath and J. Etheridge, “Visualizing the Performance of Parallel Programs,” IEEE Software, Vol 8(9), pp. 29-39, 1991.

[6] M. Ji, E. Felten, and K. Li, “Performance Measurements for Multithreaded Programs,” Performance Evaluation Review 26, no. 1, pp. 161-170, 1998

[7] S. Khanna, M. Sebrée, and J. Zolnowsky, “Realtime Scheduling in SunOS 5.0,” Proc. Winter ‘92 USENIX, 1992.

[8] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996, ISBN 0-13-172389-8.

[9] S. Maxwell, “Linux Core Kernel Commentary,” Coriolis Open Press, 1999, ISBN 1-57610-469-9.

[10] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measuring Tools,” IEEE Computer 28, vol 28, no. 11, pp. 37-46, 1995.

[11] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” Proc. 8th ISCA Int’l Conf. on Parallel and Distributed Computing Systems, Florida, USA, pp. 527-533, 1995.

[12] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to visualize and Analyse Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, 1995.

[13] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, “SunOS 5.0 Multithreaded Architecture,” Sun Soft, Sun Microsystems Inc., 1991.

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

[15] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” IEEE Software, Vol 2(6), pp. 22-37, 1985.

[16] SunSoft, “Solaris Multithreaded Programming Guide,” Prentice Hall, 1995, ISBN 0-13-160896-7.

[17] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” Proc. 5th Symp. on the Frontiers of Massively Parallel Computation, McLean, Virginia, IEEE Computer Society Press, 1995.

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

[19] Q. Zhao and J. Stasko, “Visualizing the Execution of Thread-based Parallel Programs,” Graphics, Visualization, and Usability Centre, Georgia Institute of Technology, Technical Report GIT-GVU-95-01, 1995.

Paper IV

An Allocation Strategy Using Shadow-processors and Simulation Technique

Magnus Broberg, Lars Lundberg, and Håkan Grahn
ISCA 14th International Conference on Parallel
and Distributed Computing Systems, 2001

IV

An Allocation Strategy Using Shadow-processors and Simulation Technique

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Department of Computer Science, Blekinge Institute of Technology
P.O. Box 520, S-372 25 Ronneby, Sweden

Abstract

Efficient performance tuning of parallel programs for multiprocessors is often hard. When it comes to assigning threads to processors there is not much support from commercial operating systems, like the Solaris operating system. The only known value is, in best case, the total execution time of each thread. The developer is left to the binpacking algorithm with no knowledge about the interactions and dependencies between the threads. The binpacking algorithm assigns, in the worst case, the threads to the processors such that the program will have the longest possible execution time. A simple example of such an program is shown in the paper. We present here a way of retrieving more information and a test mechanism that makes it possible to compare two different assignments of threads on processors also with regard to the interactions and dependencies between the threads. Also an algorithm is proposed that gives the best assignment of threads to processors in the case above where the binpacking algorithm gave the worst possible assignment. The algorithm uses shadow-processors and requires more processors than on the target machine during some allocation steps. Thus, a simulation tool like the one presented here must be used.

Key words: Thread allocation, simulation technique, monitoring tool, shadow-processors

1. Introduction

Parallel processing is an important way of increasing the performance of an application. Applications made for parallel processing are then likely to have performance requirements. For instance, a transaction based telecommunication billing system have performance requirements. It also have deadlines. The deadlines are often specified as the time to complete a transaction. The deadline is not specified on a thread/process level but on a transaction level perhaps involving several threads/processes. Although a missed deadline is not a life threatening event the company will loose income. Thus, predictability is an important issue in order to know that deadlines frequently will be met. By binding the threads/processes statically to processors we reduce the unpredictability of memory caches, etc. Selecting which thread should execute on which processor is vital for the performance. In Solaris [8], an commercial operating system that support multithreading on multiprocessors (SMPs, symmetric multiprocessors), there is little support for the application developer to make that choice. Basically the developer can only retrieve information about the number of threads and their respective execution time with a system tool called `tha` [11]. Based on that information a good selection of which threads to bind to which processors is hard, in practice the developer is left with the traditional binpacking algorithm [5]. Although the binpacking algorithm can not take any interaction and dependencies between the threads in account it is left as our only choice.

In this paper we present a way of retrieving more information and a test mechanism that makes it possible to compare two different assignment of threads on processors also with regard to the interactions and dependencies between the threads. We also propose a processor assignment algorithm called simple greedy algorithm (SGA) that uses the ability of the test. The SGA uses shadow-processors (shadow-processors executes the so-far not assigned threads, see Section 4) which means that the algorithm during search for the best allocation requires more processors than in the target machine. The algorithm is implemented in a simulator that can simulate any number of processors. An empirical study with 9,100 automatically generated applications shows that the new algorithm is gives in average 10% to 40% shorter execution time than the binpacking algorithm when having at least twice as many threads as processors. With less number of threads the SGA is still better but to a less degree. For some applications the improvement may be as much as 140%. Although the proposed new algorithm performs better than binpacking in most cases, there are cases when binpacking performs better. By including the binpacking algorithm in SGA, a combined algorithm called C-SGA, we can guarantee to never be worse than binpacking.

The paper is structured in the following way. In Section 2 a general overview of the tool that is the foundation for both the information gathering and the test mechanism is given. Some worst case discussions about the binpacking algorithm is found in Section 3. The description of the proposed algorithm is found in Section 4 with the empirical study in Section 5. Empirical studies for the combined algorithm are found in Section 6. Related and future work are found in Section 7 and in Section 8 the conclusions are found.

2. Overview of Tool

The tool used for information gathering and test is called VPPB (Visualization of Parallel Program Behaviour) and consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer* [1, 2]. The workflow when using the VPPB system is shown in Figure 1. The developer writes the multithreaded program (a) in Figure 1, 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 placed *between* the program and the standard thread library. Every 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. Whenever an LWP's state change the operating system informs a program called `prex`. `Prex` is a standard program on the Solaris platform used to create logfiles about various kernel events, such as state changes, page faults etc. Then the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking the application, making the tool flexible.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 1. The simulator also takes the hardware configuration and scheduling policies as input (e). The output from the simulator is information describing the predicted execution (f). By binding threads differently in (g) the effects on different bindings can be tested. The VPPB system is designed to work for C or C++ programs that uses POSIX threads [3] or the built-in thread package [12] in the Solaris 2.X operating system.

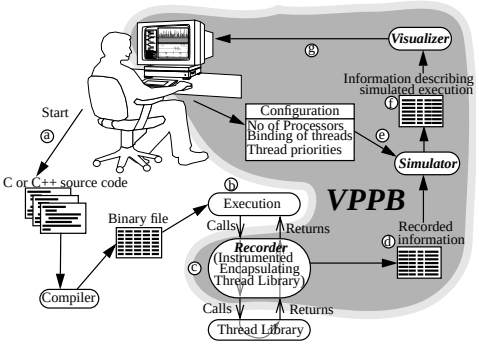


Figure 1: A schematic flowchart of the VPPB system.

3. A Worst Case Application for the Binpacking Algorithm

The binpacking algorithm is based on the only two parameters available from Solaris tools; the number of threads and each thread's execution time. The binpacking algorithm assigns the longest thread that yet has not been assigned to the processor with lowest aggregated execution time based on the so far assigned threads. In the worst case the binpacking algorithm will assign the threads in such way that the execution time will be the longest possible. This property is not desirable when having performance requirements.

Below is a simple example of an application that binpacking will assign in such way that the execution time of the application will be the longest possible. There are four threads; T_1, T_2, T_3, and T_4 to be executed on a two processor machine. The threads T_3 and T_4 are both depending on the results of both T_1 and T_2 before they can execute. The threads T_1 and T_2 are not depending on any other thread. The execution times for the threads are shown in 1, where $l \gg \varepsilon$. The binpacking algorithm will start to assign Thread T_1 (which is the longest thread) to any processor (since both processors have zero aggregated execution time), let's say processor A. Then thread T_4 (which is the second longest thread) is assigned to processor B, since it has zero aggregated execution time. Thread T_3 (the third longest thread) is assigned processor B, since processor B's aggregated time ($l + 2\varepsilon$) is smaller than processor A's ($l + 3\varepsilon$). Finally, thread T_2 (the shortest thread) is assigned to processor A, since processor A's aggregated time ($l + 3\varepsilon$) is smaller than processor B's ($(l + 2\varepsilon) + (l + \varepsilon) = 2l + 3\varepsilon$). The result is that thread T_1 and T_2 is assigned to processor A and thread T_3 and T_4 is assigned to processor B. This means that the threads in practice are executed in sequential. This is since while processor A executes T_1 and T_2, processor B with T_3 and T_4 must wait. After $2l + 3\varepsilon$ time units both T_1 and T_2 are finished and processor B can execute T_3 and T_4 for $2l + 3\varepsilon$ time units. Total execution time is then $4l + 6\varepsilon$ time units, which is also the largest possible execution time for these particular threads on a two processor machine.

Table 1: Data about the four threads.

Thread	Depends on	Execution time, $l \gg \varepsilon$	Assigned to
T_1	None	$l + 3\varepsilon$	Processor A
T_2	None	l	Processor A
T_3	T_1 and T_2	$l + \varepsilon$	Processor B
T_4	T_1 and T_2	$l + 2\varepsilon$	Processor B

4. The Simple Greedy Algorithm

Based on the information recorded by the tool, we are able to take synchronization behaviour into account when deciding which thread to bind to which processor. We use the tool's simulator in order to test the effect of placing a thread on a certain processor.

The heuristic we have chosen use a set of shadow processors that contains the so far unassigned threads. The threads are placed one by one on the most suitable target processor. The algorithm works as follows. First all threads are placed on one shadow-processor each. The threads are executing on these shadow-processors as long as they are not placed on a processor. Each thread is then moved from a shadow-processor to the most suitable processor, beginning with the longest executing thread. The thread is then tested on each processor by running the application in the simulator including the remaining threads on the shadow-processors. The thread is placed on the first processor that gave the shortest execution time. Then the next longest thread is moved from its shadow-processor and tested on each processor and so on. If we have T threads and P processors we will have to perform $P \cdot T$ tests before all threads are moved from the shadow-processors.

There is a reason for testing threads on processors that already are assigned a thread, even when there are processors that are not already assigned a thread. An example is found in Figure 2 The three threads in the example will execute to completion in 16 time units if they are assigned one processor each. If the first thread is assigned one processor and the other two executes on one shared processor the threads will execute to completion in 13 time units.

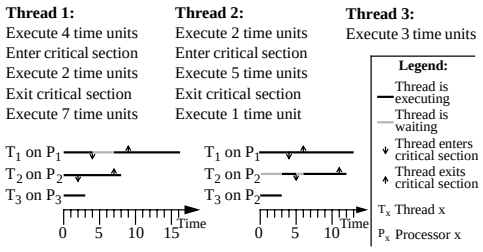
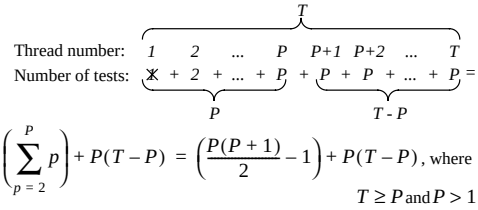


Figure 2: Illustrating the effect of assigning each thread their own processor. To the upper the three threads are specified. To the lower left the threads are assigned an own processor. To the lower right thread 2 and 3 shares processor 2, while thread 1 is assigned processor 1.

The number of tests can be reduced according to the following observation. Every processor is equal, then if there are several processors that still have not been assigned a thread, only one of them has to be tested. This is illustrated in Figure 3 where the first thread is tested on one processor, the second thread is tested on two processors (the processor which where assigned the previous thread and a free processor), the third thread is tested on the previously assigned processors and a free processor and so on until all processors have been assigned at least one thread each. After that each processor must be tested for the remaining threads. The first thread can directly be assigned any of the processors without any test, since every processor is free. In Figure 3 the worst case is shown where the P first threads are assigned an individual processor.



$$\left(\sum_{p=2}^{P} p\right) + P(T-P) = \left(\frac{P(P+1)}{2} - 1\right) + P(T-P), \text{ where } T \geq P \text{ and } P > 1$$

Figure 3: Calculating the maximum number of tests when having *T* threads and *P* processors.

We will now look how SGA handles the worst case example for binpacking in Section 3. The threads are specified as previously (see 1) and the machine has two processors. The longest thread (T_1) is assigned to any processor, lets say processor A. Then the second longest thread (T_4) is tested on processor A while the threads T_3 and T_4 is executing on a shadow processor each. The execution will look like in Figure 4(a) and the total execution time is $(l + 3\varepsilon) + (l + 2\varepsilon) = 2l + 5\varepsilon$. Then thread T_4 will be tested on processor B which is shown in Figure 4(b) with a total execution time of $(l + 3\varepsilon) + (l + 2\varepsilon) = 2l + 5\varepsilon$. This means that the execution times are equal and the first occurrence will be selected, i.e., thread T_4 will be assigned processor A. The third longest thread (T_3) is then tested on processor A as shown in Figure 4(c) with an execution time of $(l + 3\varepsilon) + (l + 2\varepsilon) + (l + \varepsilon) = 3l + 6\varepsilon$. Then thread T_3 is tested on processor B as shown in Figure 4(d) with a total execution time of $(l + 3\varepsilon) + (l + 2\varepsilon) = 2l + 5\varepsilon$. Thread T_3 is assigned to processor B since it resulted in the shortest total execution time. Finally, thread T_2 (the shortest) is first tested on processor A as shown in Figure 4(e) with an execution time of $(l + 3\varepsilon) + (l) + (l + 2\varepsilon) = 3l + 6\varepsilon$. Then thread T_2 is tested on processor B as shown in Figure 4(f) with an execution time of $(l + 3\varepsilon) + (l + 2\varepsilon) = 2l + 5\varepsilon$. Thread T_2 is assigned to processor B since it was the assignment with the shortest execution time. There is no other assignment that result in a shorter execution time than the resulting assignment, T_1 and T_4 on processor A and T_2 and T_3 on processor B.

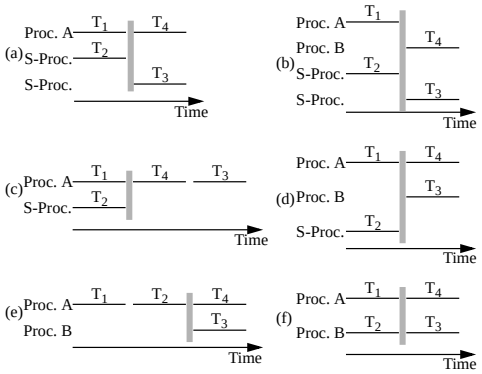


Figure 4: The different steps in the SGA algorithm for assigning the four threads in Section 3. A shadow processor is named S-Proc. The synchronization is the vertical bar, analogous to a barrier. The final assignment is the one in (f).

5. Empirical study: Simple Greedy Algorithm vs. Binpacking

In this study we have compared SGA with a binpacking algorithm. 9,100 applications have been generated with 3 to 37 threads (260 applications for each number of threads). Each applications contains critical sections protected by semaphores. The number of critical sections is between two and three times the number of threads. Each thread is generated by first executing for 2^x time units, where x is 0 to 15. Then, either the thread enters (if possible) a critical section or exits (if possible) a critical section. The probability for entering or exiting a critical section is fifty-fifty. If there is no critical section to enter or exit only the execution for 2^x time units, where x is 0 to 15, is generated. In order to avoid deadlock the critical sections are hierarchically numbered allowing a thread to enter a critical section only if the thread is not already in a critical section with a higher logical number. The threads are divided into one to half of the number of threads groups. Threads within one group only synchronizes with each other, thus, two threads in different groups will be independent. By a probability of 93.75% the thread continues to execute for 2^x time units, where x is 0 to 15, then enter or exit a critical section and so on. With a probability of 6.25% the thread will start terminating, by releasing the critical sections it has entered. Exiting each critical section is proceeded with an execution for 2^x time units, where x is 0 to 15. Finally, when all critical section are exited the thread executes for 2^x time units, where x is 0 to 15. These test applications are thought to mimic the core of a transaction based system, where each thread service a transaction. The transactions needs exclusive access to a number of resources, e.g., data structures, thus the critical sections. Some transactions are independent of other and some are not. This is why the threads are divided into groups.

In Figure 5 only the results of the applications with 4, 8, 12, 16, 20, 24, 28, and 32 threads respectively are shown, however, the other number of threads show similar results. The x-axis is the number of processors while the y-axis shows the (execution time for binpacking) / (execution time for SGA). The average curve shows the average value for all the applications with that number of threads. The max curve shows the maximum value for any application in the set, while the min curve shows the minimum value for any application in the set. As can be seen the average value is between 1.1 and 1.4 as long as there are twice as many threads as processors which means that our proposed algorithm makes the application run to completion 10% to 40% faster than the binpacking algorithm. There exists applications that can run to completion up to 2.4 times faster than the binpacking algorithm. On the other hand some applications can run to completion in only 71% of the time when using the binpacking algorithm than using SGA. This means that there are applications that the binpacking algorithm will handle better. Determine the assignment of an test application for multiprocessor with a given number of processors takes only a couple of minutes.

6. Combining the two algorithms: C-SGA

In Section 5 we showed that although in average SGA worked better than binpacking, there were cases where binpacking worked better. However, the tool that we used to implement the SGA can also be used to evaluate the binpacking algorithm. By also testing the binpacking algorithm, which requires only one more test in Figure 3, we can individually for each application choose between SGA or the binpacking algorithm. The resulting algorithm is called combined simple greedy algorithm (C-SGA) and the empirical result using the same applications as above is shown in Figure 6. The x-axis is the number of processors while the y-axis shows the (execution time for binpacking) / (execution time for C-SGA). The average curve shows the average value for all the applications with that number of thread. The max curve shows the maximum value for any application in the set. The min curve is not shown since it almost always (99.9% of the cases) will be 1.0. The average curve increased slightly (actually quite insignificantly), thus the number of applications with a value of less than 1.0 was very small in Figure 5. Out of the 9,100 test applications only 2.6% performed worse with SGA than with binpacking.

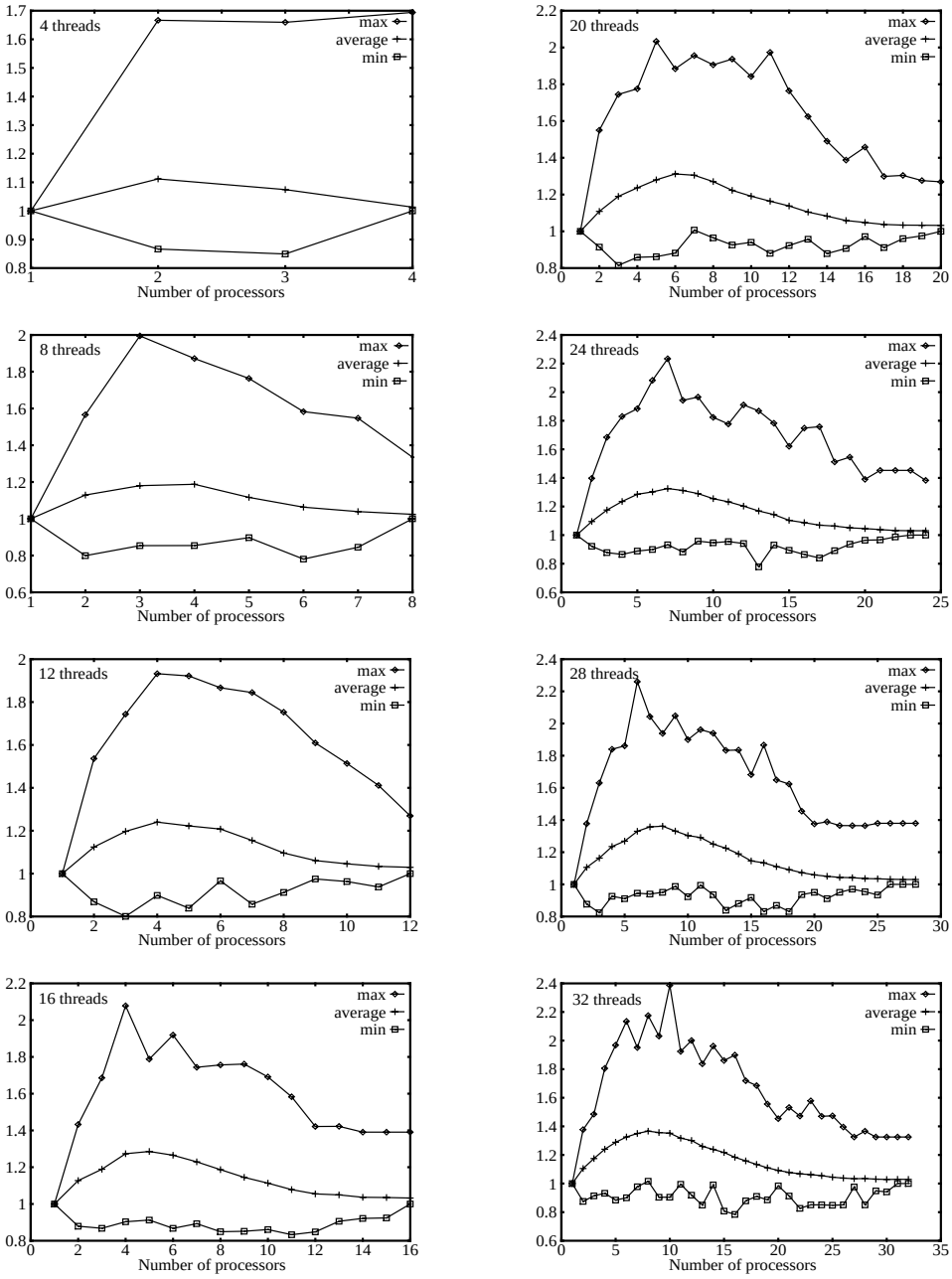


Figure 5: Simulation results for SGA. The Y-axis shows (Binpack execution time)/(SGA execution time).

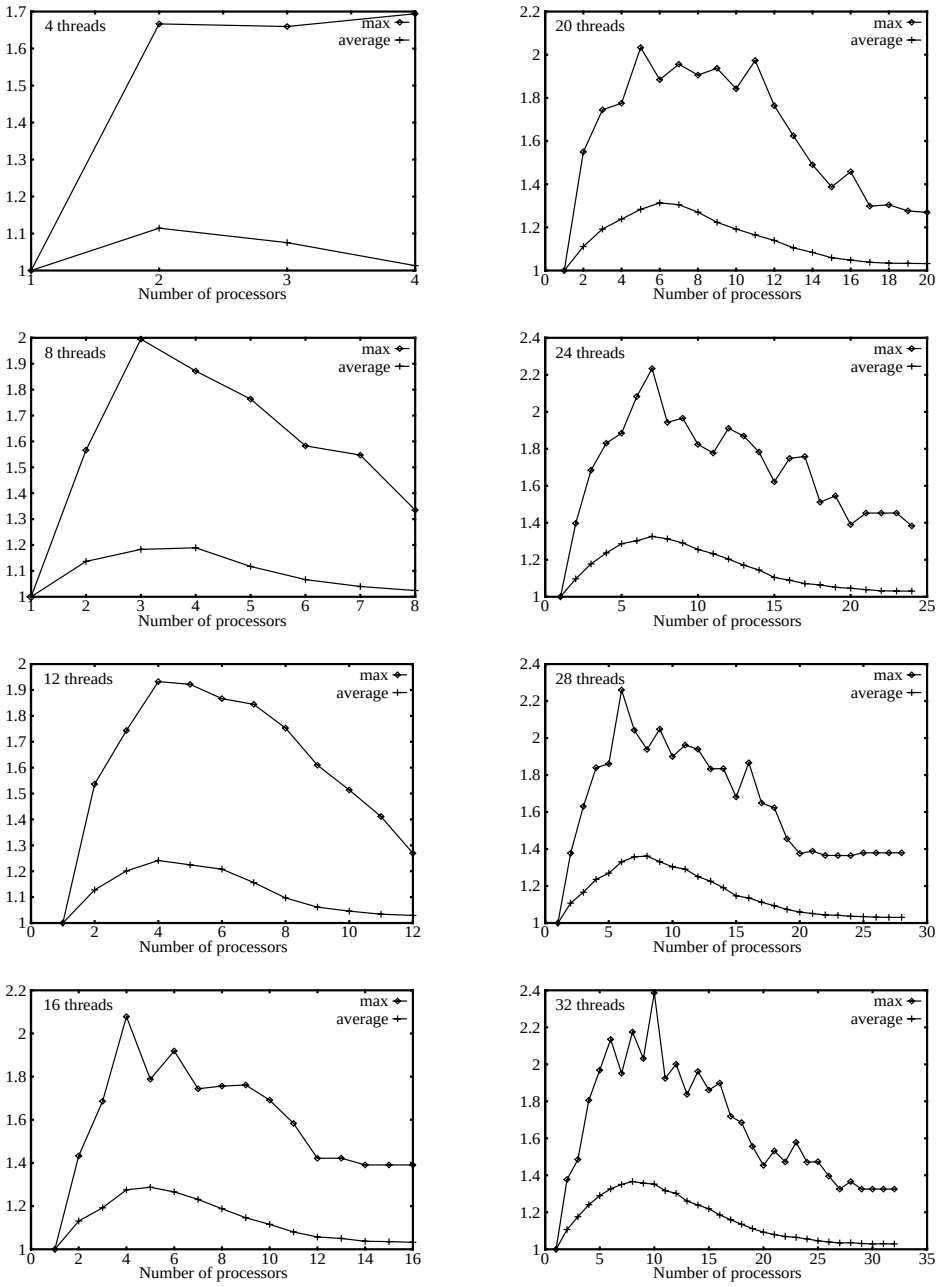


Figure 6: Simulation results for C-SGA. The Y-axis shows (execution time for binpacking) / (execution time for C-SGA).

In the case that we want an application assigned by binpacking to execute as fast as assigned by the C-SGA we would need in average up to 40% faster processors. If we on the other hand wants to be sure that binpacking will execute as fast as C-SGA we will, in worst case, need 2.4 times faster processors, based on the max curve in Figure 6. Another way of looking at the difference between binpacking and C-SGA is the number of additional processors that is needed when we use binpacking for the same performance as C-SGA. If we look at a four processors machine that run an application assigned by C-SGA, we would need seven processors in the machine to reach the same performance with binpacking. This result is fairly constant (+/-1 processor) for all applications in this study with eight threads or more. For the applications with less than eight threads, the number of threads sets the limit on the number of processors needed.

7. Related and Future Work

In the real-time community different assignment strategies for placing threads (or processes) on processors have been used for a long time [5]. Most of those strategies uses a test, called feasibility test, that determine if the current assignment will fulfil the given constraints. When it comes to multithreaded applications the goal is to make the application to run as fast as possible. Then we do not have any explicit deadlines specified for each thread. However, on a system level there might very well be specified deadlines, such as deadlines for processing an incoming event. This processing may include several threads performing a lot of work. Also these deadlines is not that hard, in other words nothing dramatic, like risk for human life, will occur if the deadline is occasionally not met. This leads us to make another kind of test that evaluates the assignment in such a way that we can say if assignment A is *better* than assignment B. To the best of our knowledge, simulation tools has not been used in this area before for processor assignment, although simulation tools like the one described in this paper previously exist, mainly for message passing systems [6, 7, 9, 10, 13]. The test mechanism can be used with other algorithms than SGA and C-SGA, for instance, simulated annealing [4] with probably even better result. This, however, is considered to be future work.

8. Conclusion

Placing threads on processors is not a trivial task. Actually, the problem is NP-complete [5]. This leaves us with heuristics. The more information we have about the application the better are our chances to achieve a good assignment. In Solaris, a multithreaded commercial operating system for multiprocessors, the support for the application developer to make an adequate assignment is limited. The operating system can provide the number of threads and their respective execution time.

However, binpacking does not deal with interactions between the threads. In this paper we use a tool to collect more information about an application and a tool for testing different assignments of threads on processors in order to determine the best of the assignments. We also propose an algorithm that uses test above, the algorithm is called SGA (simple greedy algorithm). The algorithm uses shadow-processors that makes it impossible to run the algorithm on the target machine, thus a simulation technique like the one in this paper must be used. An empirical study with 9,100 automatically generated applications shows that SGA in average gives 10% to 40% shorter execution time than binpacking when there are at least twice as many threads as processors, otherwise SGA gives shorter execution time but to a less degree. Occasionally SGA behaves worse than binpacking. However, the test can also cope with the binpacking algorithm. By combining the two algorithms, resulting in an algorithm called C-SGA (combined simple greedy algorithm), the result is guaranteed to be at least as good as binpacking and performs even slightly better in average than SGA.

References

[1] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads,” Proc. 13th Int’l Parallel Processing Symp., pp. 407-413, 1999.

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

[3] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, 1997, ISBN 0-20-163392-2.

[4] S. Kirkpatrick, “Optimization by Simulated Annealing: Quantitative Studies,” J.Statistical Physics, Vol. 34 no. 5-6, pp. 975-986, 1984.

[5] C. M. Krishna and K. G. Shin, “Real-Time Systems,” The McGraw-Hill Companies, Inc., 1997.

[6] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” Proc. 8th ISCA Int’l Conf. on Parallel and Distributed Computing Systems, pp. 527-533, 1995.

[7] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to visualize and Analyse Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, February 1995.

[8] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, “SunOS 5.0 Multithreaded Architecture,” Sun Soft, Sun Microsystems Inc., September 1991.

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

[10] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” IEEE Software, 2(6):22-37, November 1985.

[11] Sun Man Pages, tha, Sun Microsystems Inc., 1996.

[12] Sun Soft, “Solaris Multithreaded Programming Guide,” Prentice Hall, 1995

[13] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” Proc. 5th Symp. on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, February 1995.

Paper V

A Tool for Binding Threads to Processors

Magnus Broberg, Lars Lundberg, Håkan Grahn
Euro-Par, 2001

V

A Tool for Binding Threads to Processors

Magnus Broberg, Lars Lundberg, and Håkan Grahn

Department of Software Engineering and Computer Science
Blekinge Institute of Technology, P.O. Box 520, S-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@bth.se

Abstract. Many multiprocessor systems are based on distributed shared memory. It is often important to statically bind threads to processors in order to avoid remote memory access, due to performance. Finding a good allocation takes long time and it is hard to know when to stop searching for a better one. It is sometimes impossible to run the application on the target machine. The developer needs a tool that finds the good allocations without the target multiprocessor. We present a tool that uses a greedy algorithm and produces allocations that are more than 40% faster (in average) than when using a binpacking algorithm. The number of allocations to be evaluated can be reduced by 38% with a 2% performance loss. Finally, an algorithm is proposed that is promising in avoiding local maxima.

1 Introduction

Parallel processing is a way of increasing application performance. Applications made for parallel processing are also likely to have high performance requirements. Many multiprocessor systems are based on a distributed shared memory system, e.g., SGI Origin 2000, Sun's WildFire [4]. To minimize remote memory accesses, one can bind threads statically to processors. This can also improve the cache hit ratio [8] in SMPs. Moreover, some multiprocessor systems do not permit run-time reallocation of threads. Finding an optimal static allocation of threads to processors is NP-complete [7].

Parallel programs are no longer tailored for a specific number of processors, this in order to reduce the maintenance etc. This means that different customers, with different multiprocessors, will share the same application code. The developer have to make the application run efficiently on different numbers of processors, even to scale-up beyond the number of processors available in a multiprocessor today in order to meet future needs. There is thus no single target environment and the development environment is often the (single processor) workstation on the developer's desk.

Heuristics are usually able to find better bindings if one lets them run for a long period of time. There is a trade-off between the time spent searching for good allocations and the performance of the program on a multiprocessor. Another property of the heuristics is that the improvement per time unit usually decreases as the search progresses. It is thus not trivial to decide when to stop searching for a better best allocation.

In operating systems like Sun Solaris there is little support to make an adequate allocation. A tool called `tha` [10] only gives the total execution time for each thread enough for an algorithm like binpacking [7] which does not take thread synchronization into consideration. The result is quite useless, since the synchronizations do impact.

In this paper we present a tool for automatically determining an allocation of threads to processors. The tool runs on the developer's single-processor workstation,

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 57-61, 2001.

and does not require any multiprocessor in order to produce an allocation. The tool considers thread synchronizations, thus producing reliable and relevant allocations.

In Sect. 2 an overview of the tool is found. Empirical studies are found in Sect. 3. In Sect. 4 related and future work is found. The conclusions are found in Sect. 5.

2 Overview of the Tool

The tool used for evaluation of different allocations is called VPPB (Visualization of Parallel Program Behaviour) [1] and [2] and consists of the Recorder and the Simulator. The *deterministic* application is executed on a single-processor workstation, the Recorder is automatically placed *between* the program and the thread library. Every time the program uses the routines in the thread library, the call passes through the Recorder which records information about the call. The input for the simulator is the recorded information and an allocation of threads generated by the Allocation Algorithm. The output from the Simulator is the execution time for the application on a multiprocessor with the given size and allocation. The predicted execution time is fed into the Allocation Algorithm and a new allocation is generated. Simulations are repeated until the Allocation Algorithm decides to stop. The evaluation of a single allocation by the Simulator is called a *test*. The VPPB system works for C/C++ programs with POSIX [3] or Solaris 2.X [11] threads. In [1] and [2] the Simulator was validated with dynamically scheduled threads. We validated the tool on a Sun Enterprise 4000 with eight processors with statically bound threads and used nine applications with 28 threads, generated as in Sect. 3. The maximum error is 8.3%, which is similar to the previous errors found in [1] and [2].

3 Empirical Studies with the Greedy Algorithm

3.1 The Greedy Algorithm

The Greedy Algorithm is based on a binpacking algorithm and makes changes (swap and move) to the initial allocation and test it. If the new allocation is better it is used as the base for next change and so it continues, see Fig. 1. The algorithm keeps track of all previous allocations in order to not test the same allocation several times.

3.2 The Test Applications Used in This Study

In this study we used 4,000 automatically generated applications divided into eight groups of 500 applications with 8, 12, 16, 20, 24, 28, 32, and 36 threads, respectively. Each application contains critical sections protected by semaphores. The number of critical sections is between two and three times the number of threads. Each thread executes first for 2^x time units, x is 0 to 15 throughout this section. Then, with a probability of 50%, the thread enters (if possible) a critical section or exits (if possible) a critical section. Deadlocks are avoided by a standard locking hierarchy. The threads are divided into groups. Threads within one group only synchronizes with each other. The number of groups is between one up to half of the number of threads. With a probability of 93.75% the thread continues to execute for 2^x time units then enter or exit a critical section and so on. With a probability of 6.25% the thread will start terminating by releasing it's critical sections. Exiting each critical section is proceeded with an execution for 2^x time units. Finally, the thread executes for 2^x time units.

```
Allocation bestAlloc = allocateAccordingToBinpack();
Time bestExecutionTime = simulate(bestAlloc);
addToAllocationHistory(bestAlloc);
algorithm(bestAlloc);

procedure algorithm(Allocation alloc) {
  Time executionTime;
  if(random() > 0.5 and allThreadsAreNotOnTheSameProcessor())
    swapARandomThreadWithARandomThreadOnAnotherProcessor(alloc);
  else
    moveARandomThreadToAnotherRandomProcessor(alloc);
  if(allocationIsAlreadyInAllocationHistory(alloc))
    { alogrithm(bestAlloc); return; }
  addToAllocationHistory(alloc);
  executionTime = simulate(alloc);
  if(executionTime == bestExecutionTime)
    { alogrithm(alloc); return; }
  if(executionTime < bestExecutionTime)
    { bestAlloc = alloc; bestExecutionTime = executionTime; }
  alogrithm(bestAlloc);
}
```

Fig. 1. The greedy algorithm in pseudo code

3.3 Characterizing the Greedy Algorithm

The Greedy Algorithm presented in Sect. 3.1 will actually never stop, thus in order to investigate its performance we had to manually set a limit. We chose to continue the algorithm until 500 tests had been performed. We also stored the so far best allocation when having done 20, 60, ... , 460, 500 tests, shown in Fig. 2. When the number of tests is high yet another 40 tests will not decrease the application's execution time much.

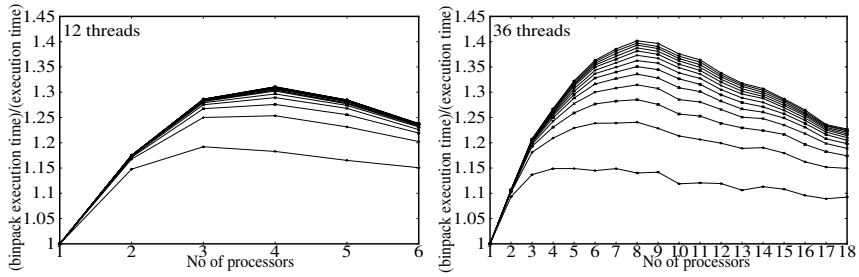


Fig. 2. The average gain by using the Greedy Algorithm as compared to binpacking

3.4 A Threshold for the Greedy Algorithm

The graphs in Fig. 2 clearly shows that it should be possible to define a threshold that stops the Greedy Algorithm to do more tests when the gain will not be significant. The stop criterion for the Greedy Algorithm is defined as to stop when a number of consecutive tests have not gained anything in execution time. We have empirically found that 100 consecutive tests is a good stop criterion if we accept a 2% loss in performance compared to performing 500 tests. By reducing the gain by 2% we reduce the number

of tests by 38%. This is of important practical value since the number of tests performed is proportional to the time it takes to calculate the allocation. Performing 500 tests took at most two minutes for any application in the population used in this study.

3.5 Local Maxima and Proposing a New Algorithm

There is always the danger of getting stuck in a local maximum and the Greedy Algorithm is not an exception. In order to investigate if the algorithm runs into a local maximum we used the previous application population with 16 and 24 threads. By giving the Greedy Algorithm a random initial allocation, instead of an allocation based on binpacking, we reduced the risk of getting stuck in the same local maximum.

The result of using 10 x 500 vs. 1 x 5000 tests and 10 x 50 vs. 1 x 500 tests is shown in Fig. 3. As can be seen, sometimes it is better to run ten times from different starting allocations than running the Greedy Algorithm ten times longer from a single starting allocations and sometimes it is not. The reason is found in Fig. 2. As the number of tests increases the gain will be less and less for each test. Thus, when the number of tests reaches a certain level the Greedy Algorithm is close to a local maximum and ten times more tests will gain very little. By using ten new initial allocations that particular local maximum can be avoided, and if the new initial allocations are fortunate a better result is found when reaching the same number of tests. This is what happened in the case with 10 x 500 and 1 x 5000. On the other hand if the number of tests is low the Greedy Algorithm still can gain much with running the same initial allocation for ten times longer. This opposed to running ten Greedy Algorithm with random initial allocation to the previously low tests. This is the case for 10 x 50 and 1 x 500.

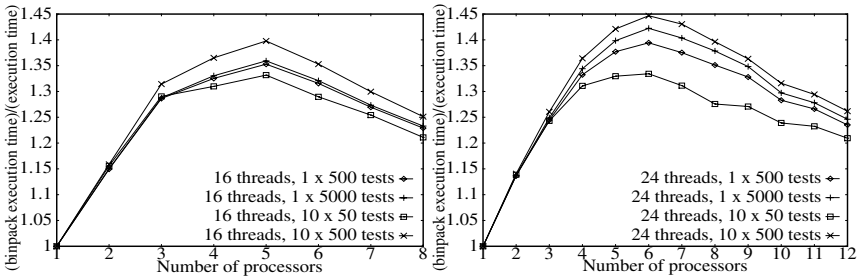


Fig. 3. The difference between 1 x 500 vs. 10 x 50 tests, and 1 x 5000 vs. 10 x 500 tests

Based on the findings above and the threshold we propose a new algorithm, called Dynamic Start Greedy Algorithm. The algorithm is the Greedy Algorithm with an initial allocation based on binpacking. When the threshold has been reached a new initial allocation is created and the algorithm continues until the threshold is reached again. The new algorithm uses the threshold in order to determine whether it is useful to continue running or not. At the same time the algorithm is able to stop running and choose a new initial allocation before it runs too long time with an initial allocation The history of already tested allocations should be kept from one initial allocation to the next.

The Dynamic Start Greedy Algorithm continues with an initial allocation until it reaches a local maximum, then it jumps to a new initial allocation to investigate if it is better. This is the inverse of simulated annealing [6] that jumps frequently in the beginning and more seldom after a while.

4 Related and Future Work

The GAST tool [5] is somewhat similar to the tool described here. GAST originates from the real-time area. With GAST it is possible to automate scheduling, by defining different scheduling algorithms. The GAST tool needs a specification of all tasks, their worst case execution time, period, deadline and dependencies. This specification must be done by hand, which could be a very tedious task to do. Also, high performance computing does not necessarily have either periods or deadlines and a task in GAST may only be a fraction of a thread, since a task can not synchronize with another task inside the task. Each thread must then (by hand) be split into several tasks.

The Greedy Algorithm could be compared, using the tool described in this paper, by replacing the algorithm with simulated annealing [6], etc. This, however, is considered to be future work.

5 Conclusion

We have presented and validated a tool that makes it possible to execute an application on a single processor workstation and let the tool find an allocation for the application on a multiprocessor with any number of processors. The tool uses a Greedy Algorithm that may improve the performance of an application with more than 40% (in average) compared to the binpacking algorithm. We have also shown that it is possible do define a stop criterion that stops the algorithm when there have been no gain for a certain time. By trading off a speed-up loss of 2% we reduced the number of tests performed with 38%. Finally, a new algorithm is proposed that seems promising in giving even better allocations and reducing the risk of getting stuck in a local maxima.

References

1. Broberg, M., Lundberg, L., Grahn, H.: Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads. Proc. IPPS (1999) 407-413
2. Broberg, M., Lundberg, L., Grahn, H.: VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs. Proc. IPPS (1998) 770-776
3. Butenhof, D.: Programming with POSIX Threads. Addison-Wesley (1997)
4. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. Proc. 5th Int Symp. on High Performance Computer Architecture (1999) 172-181
5. Jonsson, J.: GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques. Proc. ICPP, (1998) 441-450
6. Kirkpatrick, S.: Optimization by Simulated Annealing: Quantitative Studies. J. Statistical Physics **34** (1984) 975-986
7. Krishna, C., Shin, K.: Real-Time Systems. The McGraw-Hill Companies, Inc. (1997)
8. Lundberg, L.: Evaluating the Performance Implications of Binding Threads to Processors. Proc. 4th Int Conf. on High Performance Computing (1997) 393-400
9. Powell, M., Kleiman, S., Barton, S., Shah, D., Stein, D., Weeks, M.: SunOS 5.0 Multithreaded Architecture. Sun Soft, Sun Microsystems, Inc. (1991)
10. Sun Man Pages: tha, Sun Microsystems Inc. (1996)
11. Sun Soft: Solaris Multithreaded Programming Guide. Prentice Hall (1995)

Paper VI

Performance Optimization using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Journal of Parallel and Distributed Computing, Vol. 61, No. 1, 2001

VI

Journal of Parallel and Distributed Computing **61**, 115–136 (2001)
doi:10.1006/jpdc.2000.1667, available online at http://www.idealibrary.com on IDEAL®

Performance Optimization Using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors

Magnus Broberg, Lars Lundberg, and Håkan Grahn

Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Soft Center, S-372 25 Ronneby, Sweden
E-mail: Magnus.Broberg@ipd.hk-r.se, Lars.Lundberg@ipd.hk-r.se, Hakan.Grahn@ipd.hk-r.se

Received November 22, 1999; revised May 24, 2000; accepted May 25, 2000

Efficient performance tuning of parallel programs is often hard. Optimization is often done when the program is written as a last effort to increase the performance. With sequential programs each (executed) code segment will affect the completion time. In the case of a parallel program executed on a multiprocessor this is not always true, due to dependencies between the different threads. Thus, certain code segments of the execution may not affect the completion time of the program. Optimization of such code segments will not increase the performance. In this paper we present an approach to optimize performance by finding the extended critical path of the multithreaded program. The extended critical path analysis is a generalization of the critical path analysis in the sense that it also deals with more threads than processors. We have implemented the extended critical path analysis in a performance optimization tool. The tool allows the user to determine the extended critical path of a multithreaded application written for the Solaris operating system for any number of processors based on execution on a single processor workstation.

Key Words: Multithreading; multiprocessor; critical path analysis; performance optimization; performance analysis.

1. INTRODUCTION

Parallel processing is an important way to increase the performance of computationally demanding applications, and applications developed for multiprocessors are likely to have high performance requirements. However, it is not always easy to write parallel applications for multiprocessors. The application developers need support in order to write parallel applications with high performance.

A number of commercial multiprocessor platforms have emerged. The developer must make sure that the application runs efficiently on different numbers of processors, since it is the customer that decides the size of the multiprocessor based on

0743-7315/01 $35.00

the actual performance requirements and the price/performance ratio. In some cases, the developers want the multithreaded application to scale-up beyond the number of processors available in a multiprocessor today in order to meet future needs. Thus, there is no single target environment and the development environment may not be the same as the target environment. Often the development environment is the (single processor) workstation on the developer's desk.

Just like the code in sequential applications, the code in parallel applications can be optimized in order to reduce the completion time of the application and thus increase the performance. The optimization is done by reducing the execution time for certain code segments, preferably the code segments that have the largest impact on the completion time of the application. In the case of a sequential application, all executed code segments contribute to the completion time. When executing a parallel application on a multiprocessor all code segments may not contribute to the completion time. Therefore, it is hard for the developer to know which code segments will actually reduce the completion time of the parallel application, thus making it hard to prioritize the optimization efforts.

Consider the program in Fig. 1. We assume that all functions (a–d) take an equal amount of time to execute and that signaling takes no (or negligible) time. When executing the program on an SMP (symmetric multiprocessor) with three processors (or more) the execution will look like Fig. 1. Imagine that function `a( )` is optimized with a small value ε; then the completion time will be 2ε shorter. Further, imagine that function `b( )` is optimized with a small value ε, then there will be no reduction in the completion time at all. When optimizing functions `c( )` or `d( )` with a small value ε the reduction will be equal to ε. Thus, optimizing function `a( )` will have the largest impact. However, when executing the program on one processor, function `b( )` will be the function with the largest impact (3ε).

The functions execute for the same amount of time in this example. Neither different execution length of the different functions nor different execution length for different invocations of the same function (due to, e.g., different input parameters) are limitations in our technique (see Section 3.2).

We define the *extended critical path* as all the executed code segments of a program that, when reduced with a small ε, will reduce the completion time on

```
Thread 1:                Thread 2:                Thread 3:
Execute a()              Execute b()              Wait for event X
Signal event X           Wait for event Y         Execute c()
Execute b()              Execute d()              Signal event Y
Wait for event Z         Signal event Z           Execute b()
Execute a()              End                      End
End
```

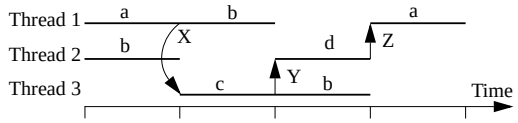


FIG. 1. A simple program with three threads and the execution on a multiprocessor with three processors.

a given number of processors. Consequently, all code segments of a program executed on a single processor will be considered as part of the extended critical path. This is because if we reduce *any* part of a program on a single processor it will result in a shorter completion time.

Ordinary profiling tools, such as Quantify [13], only consider the (extended) critical path in the case of one processor. They will indicate that the program in Fig. 1 spends most of the time in function `b( )` and suggest that function for optimization. This is shown in Fig. 2 where the functions `a( )` to `d( )` are found at the top; the other functions shown are only for internal use in the Solaris operating system. The tools and methods described in [3, 8, 18] have critical path analysis and will show that `a( )` is the most beneficial to optimize. However, those tools and methods assume that there is only one thread (or process) scheduled on each processor.

Our approach with the extended critical path generalizes the two extreme cases above. Not only in the case of one processor for *all* threads, or one processor for *each* thread, but *for any number of processors regardless of the number of threads.* Thus, with our approach it is also possible to analyze the application in Fig. 1 for the case of two processors. The techniques presented here do not require access to a multiprocessor. The approach used is to trace the behavior of a parallel application on a single processor workstation. Using these recordings, information about the performance of the parallel application on a multiprocessor with an arbitrary number of processors is obtained by simulation. As a proof of concept, a tool called visualization and predication of parallel program behavior (VPPB) has been modified to show that the methods are applicable in a real world programming environment.

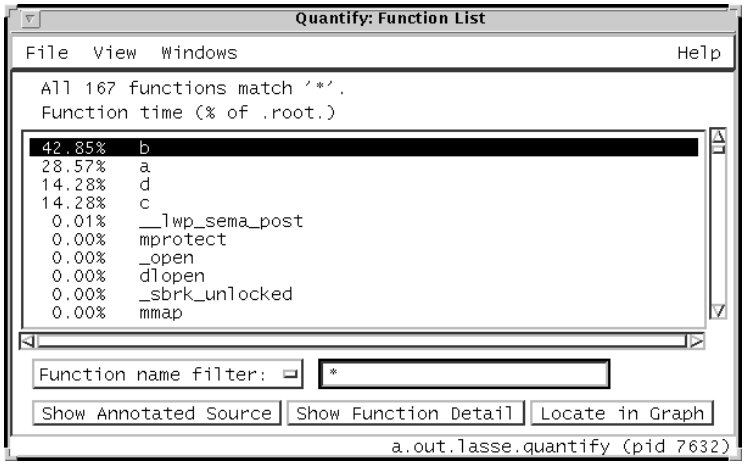


FIG. 2. Quantify's list of the functions to optimize.

The paper is structured as follows: Section 2 describes the algorithm for calculating the extended critical path. In Section 3 we present a working tool with the extended critical path analysis implemented and show by three practical examples in Section 4 the need for such a tool. Discussion is found in Section 5, and related work is found in Section 6. In Section 7 future work is discussed. The conclusion is found in Section 8.

2. THE EXTENDED CRITICAL PATH ALGORITHM

In this section we describe how the extended critical path algorithm works. First, we describe the algorithm for identifying the extended critical path in the special case with one thread per processor. Then, we describe how to calculate the time spent by each function executed in the extended critical path. Finally, we show how to calculate the extended critical path for multithreaded programs with more runnable threads at the same time than there are processors.

2.1. Finding the Extended Critical Path with One Thread per Processor

We start to look at the extended critical path algorithm under the condition that the program is executed on a sufficient number of processors and thus no threads have to share a processor at any time. This is the same assumption used in [3, 8, 18]. We assume that explicitly synchronized events happen at the exact same time, such as one thread releasing another thread. This means that a thread starts to execute at exactly the same moment as the other thread releases it. This assumption is realistic, does not change the algorithm in principal, and is kept here in order to simplify the presentation. A *segment* is the execution for a thread between two synchronizations, in this case we also regard the beginning and the end of the thread as a synchronization. The algorithm is found in Fig. 3 in pseudo-code.

The algorithm starts at the end of the execution and follows the last thread, called *i*; it continues backward until the thread *i* was blocked for some reason. Then the algorithm finds the event by thread *j* that released thread *i*. The algorithm then continues to follow the releasing thread *j* backward until it also was blocked. Thus, in that manner the algorithm continues until the start of the program. All code segments of the program that the algorithm goes through are part of the extended critical path.

```
segment = find_last_executing_segment();
stop = find_first_executing_segment();

mark_for_critical_path(segment);
while(segment != stop) {
  previous_segment = find_previous_segment_for_the_same_thread(segment);
  if(start_time_for(segment) == end_time_for(previous_segment)) /* Was blocked? */
    segment = previous_segment; /* Not blocked */
  else
    segment = find_segment_with_end_time_equal_to_start_time_of(segment); /* Blocked */
  mark_for_critcal_path(segment);
}
```

FIG. 3. Pseudo-code for finding the critical path with unlimited number of processors.

To illustrate the algorithm we use the example program with three threads in Fig. 4. The execution times for the different functions are illustrated in the lower part of Fig. 4, where also the execution of the program on three processors is illustrated.

Following the algorithm described above we start at the end of the execution, i.e., thread 3 at time 11. Tracing thread 3 backward makes us go through functions `f( )`, `a( )`, `h( )`, and finally `g( )`. At time 4 to 5 the thread was not executing; i.e., it was blocked. The thread (3) was waiting for event *Z*. Thread 1 releases thread 3 at time 5; thus we continue the algorithm with thread 1. Thread 1 executes through functions `b( )` and `a( )` without being blocked until the start of the program at time 0.

Thus the extended critical path for the program is thread 1 executing functions `a( )` and `b( )` and thread 3 executing functions `g( )`, `h( )`, `a( )`, and `f( )`. It is easily verified that optimizing any *other* part of the program will not affect the completion time when using three processors.

2.2. Calculating Function Execution Times Contributing to the Extended Critical Path

In the previous section, all the synchronizations were done between the function calls. This is not always the case, since synchronizations could appear inside the functions as well. This complicates the calculation of the time spent in certain functions during the extended critical path. This calculation is done after the extended critical path analysis; thus we only concentrate our effort on those parts that are in the extended critical path. We will calculate three values *per function*:

```
Thread 1:                Thread 2:                Thread 3:
Execute a()              Wait for event X         Wait for event Y
Signal event X           Execute c()              Execute d()
Execute b()              Signal event Y           Wait for event Z
Signal event Z           Execute g()              Execute g()
Execute c()              Wait for signal Y        Execute h()
Wait for event X         Execute d()              Signal event X
Execute e()              End                      Execute a()
Execute i()                                       Signal event Y
End                                               Execute f()
                                                  End
```

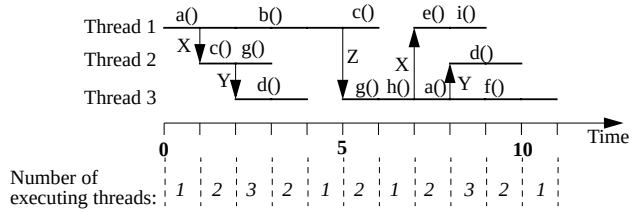


FIG. 4. The optimal execution of the program with three threads. The values in *italic* indicate the number of executing threads.

- The number of calls. This is the number of times the function was called during the extended critical path.
- The total execution time (TET). This time is the accumulated time, in the extended critical path, the program was executing in the function.
- The total execution time including all descendant functions (TETID). This time is the accumulated execution time for the function and its descendants during the extended critical path.

The first value is determined by calculating the number of function entrances along the segments in the extended critical path. The other two values are calculated by re-creating the function call stack. For each segment in the extended critical path, the function on the top of the stack will increase its TET with the length of the segment. The other functions on the stack will increase its TETID with the length of the segment. The re-creation of the function call stack is needed since the function entrance may not be in the extended critical path, but parts of the execution in the function may be.

2.3. Finding the Extended Critical Path with CPU Constraints

In Section 2.1 we defined the algorithm for finding the extended critical path when an unlimited number of processors are available. Unfortunately, this is not always the case in real life. The work done in [3, 8, 18] does not address that. Thus, we need to adjust the algorithm to fit whenever there are more runnable threads than there are processors. In that case, some threads at some times must be multiplexed by the scheduler on one processor.

We keep the example in Fig. 4, but look at what happens if we only have two processors available. The number of executing threads is indicated for each time unit as the figures in *italic*. The interesting parts are when the number of threads is larger than the number of processors. In the example this is time 2 to 3 and time 8 to 9 where three threads are executing on the two available processors. One way of modeling the multiplexing is to consider the processors to run more slowly as the number of threads increases. The processors run at only 2/3 speed for time 2 to 3 and time 8 to 9; i.e., we assume that the scheduling is ideal with infinitely small time slices and no scheduling overhead. This assumption has previously been used in some performance prediction tools with accurate result when compared with real scheduling [7].

In Fig. 5 pseudo-code for finding the extended critical path is found. Before explaining the algorithm we define two different kinds of segments:

- A *segment* is the execution for a thread between two synchronizations, in this case we also regard the beginning and the end of the thread as a synchronization, using one processor per thread. In Fig. 4 each "execute" statement is a segment.
- A *time segment* is the time period when no threads synchronize or are started or finished in the optimal execution. In Fig. 4 each time unit is a time

```
calculate_execution_time() {
  exec_time = 0;
  for_each_time_segment {
    exec_time = exec_time + length_of_time_segment * max(number_of_threads/P, 1);
  }
  return exec_time;
}
calculate_extended_critical_path() {
  original_execution_time = calculate_execution_time();
  for_each_segment {
    decrease_segment_execution_time_with(epsilon);
    set_segment_weight((original_execution_time - calculate_execution_time())/epsilon);
    increase_segment_execution_time_with(epsilon); /* to restore the execution time */
  }
}
```

FIG. 5. Pseudo-code for finding the extended critical path with limited number of processors. P is the (limited) number of processors available.

segment. Consequently, since we assume that no events happen exactly at the same time, the number of time segments is equal to the total number of segments for each thread.

The first function (`calculate_execution_time`) in Fig. 5 calculates the time required to execute the application. First we divide the execution into time segments. The time required for executing such a time segment depends on the number of currently executing threads. If the number of executing threads during the time segment is less than or equal to the number of processors available then the execution time is the same as the time segment length. When there are more threads than processors, the time required to execute that time segment will be longer. How much longer is given by the number of threads divided by the number of processors. This is then multiplied with the time segment length in order to get the executing time of the time segment.

The second part of Fig. 5 (`calculate_extended_critical_path`) describes how to identify the segments that are in the extended critical path. First the total execution time required is calculated. Then, for each segment, the execution time is calculated if that segment is shorted by a small value ε, less than the shortest time segment above. In Fig. 4, this corresponds to less than one time unit. If the execution time is not affected by this change, the segment is not part of the extended critical path. In other cases the segment is part of the extended critical path. By dividing the difference of the execution times with the selected ε we get a weight of how much impact the segment has on the extended critical path.

Following the algorithm in Fig. 5 the extended critical path for the application in Fig. 4 when executed on two processors will be: thread 1, `a( )`, `b( )`, `e( )`, and `i( )`; thread 2, `g( )`; and thread 3, `g( )`, `h( )`, `a( )`, and `f( )`. The path `a( )` and `b( )` on thread 1 and `g( )`, `h( )`, `a( )`, and `f( )` on thread 3 is the critical path for an unlimited number of processors as discussed in Section 2.1. The execution time is found in Fig. 6a and is 12 time units. If we, e.g., select function `g( )` on thread 2, the algorithm will shorten the execution time for `g( )` on thread 2 with a small value, e.g., one half time unit. Then the execution will look like Fig. 6b. The time segment where `g( )` is included is now from time 2 to 2.5 and the following time segment is from time 2.5 to 4. The resulting execution time is now 11.75 and

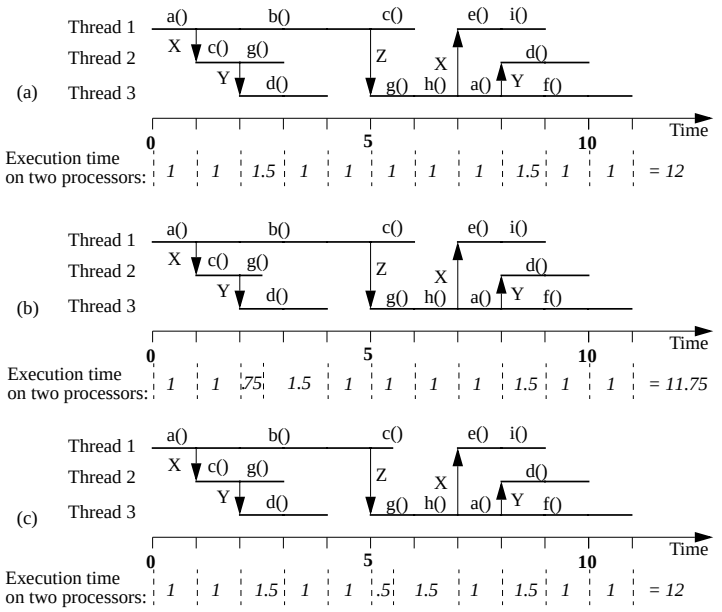


FIG. 6. A program with its three threads. Each time segment is indicated with dotted lines. The values in *italic* indicate the time required to execute that time segment.

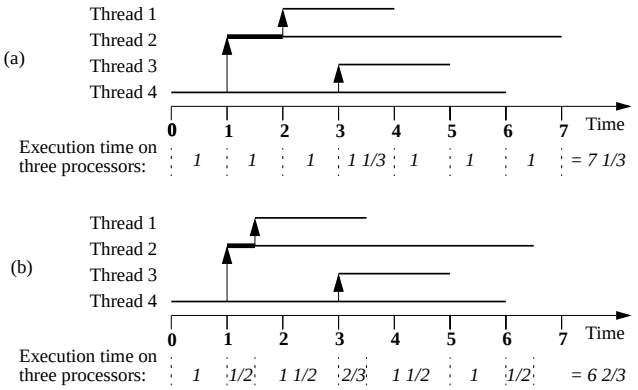


FIG. 7. An example showing that the weight for a segment may be more than one. (a) The nonoptimized execution. (b) The thicker line optimized with half a time unit. In this example the weight for the thicker line is 1.333.

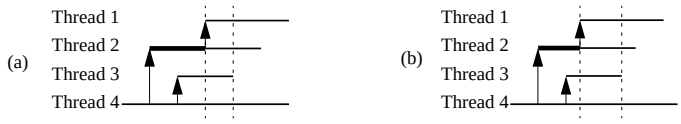


FIG. 8. An example of how optimization can extend the total execution time for an application.

this is less than in Fig. 6a. Thus, function `g( )` on thread 2 is part of the extended critical path and the weight is $0.5 = (12 - 11.75)/0.5$.

A segment, e.g., function `c( )`, on thread 1 is not part of the extended critical path as shown in Fig. 6c since the execution time is the same as in Fig. 6a.

Now, we have found the extended critical path and can calculate how much a function spent in the extended critical path as discussed in Section 2.2, but with the difference that the function times are multiplied with the weight for the segment.

The weight of a segment can be more than one, as shown in Fig. 7. When optimizing the thicker segment with, say, half a time unit and executing on three processors two things happen. First, the period of time when there are four processors is reduced; thus, total execution time is gained. Second, the last thread executing (thread 2) decreases by half a time unit. The total effect is 2/3 time units by optimizing half a time unit. The weight will then be $1.333 = 0.667/0.5$. Note that when analyzing the same example on two processors the same segment will have a weight of 1.0; i.e., the weight of the segment *increases* when going from two processors to three.

Another aspect of the extended critical path algorithm described is that the algorithm also identifies which segments have a *negative* impact on the total execution time when optimized. Consider the example in Fig. 8a. When this application is executed on three processors an optimization of the thicker segment will increase the total execution time. This is because optimizing the thicker line will shift thread 1 and the time segment when there are four threads competing for the processors is even longer as shown in Fig. 8b. Our algorithm will give the segment a negative weight.

3. IMPLEMENTATION OF THE EXTENDED CRITICAL PATH ALGORITHM

The extended critical path analysis has been implemented in a tool called VPPB [1, 2]. The extended critical path analysis increases the capacity of the tool in a natural way.

3.1. Short Description of the VPPB Tool and the Environment

VPPB is a tool for performance optimization of multithreaded programs written in C/C++ on the Solaris 2.X operating system. This is an environment common in both academia and industry. The VPPB tool combines two things: visualization of a multithreaded program's behavior, and prediction of how the program will execute on a multiprocessor.

The tool traces the execution of a multithreaded program on a single processor workstation. The tracing is performed by wrapping the thread library in Solaris 2.X and recording all the calls made by the program. The recorded information includes data about when, by which thread, with what parameters, etc., the call was issued. The recorded information is saved to file upon program exit. Thus, the behavior of the program is traced. Additional information about the preemptive scheduling of the LWPs (lightweight processes) [16] is also collected with a Solaris system command called `prex` described in [1]. The next step is to simulate the recorded information. The Simulator mimics a multiprocessor with any number of processors, the Solaris scheduling model, and different configurations of the threads [2].

The simulated execution is displayed graphically with two graphs. The first graph shows the amount of parallelism over time, as well as the number of runnable threads, i.e., the number of threads ready to execute but with no processor available. The second graph shows the execution as a Gant diagram based on the threads, with all synchronization (semaphores, mutexes, etc.) as symbols through the execution. It is possible to get more information about a single event (such as a signal on a semaphore at a given time) including the source code line where the call was made. With this information the developer can easily detect and identify performance bottlenecks.

3.2. Introducing Extended Critical Path Analysis

The introduction of extended critical path analysis requires that all the function entrances and exits must be traced. The insertion of function probes is done as a stage in the compilation phase of the program. Previously, there was no need for any recompilation or special compilation. The compilation is now done in three stages for each source code file. The first stage is to compile the source code into assembler code. The assembler code is then parsed in the next stage and for each function entrance–exit probes are inserted to record the events. We are then able to probe each invocation of each function. We use the same kind of recording probes, TNF (trace normal form), as previously used in the tool [1]. To simplify the implementation of the parser it assumes that the compiler uses the default optimization. It made the implementation easier because a function will then have one single exit point and the function calls will look the same even if the function is a leaf. A more sophisticated parser could, however, deal with those issues. Also, more extensive optimization may inline functions instead of making explicit calls; this cannot be dealt with (this is, however, a common limitation in profilers). The third step is then to compile the modified assembler code into object code. This stage is performed by the ordinary C/C++ compiler. The probes keep track of the function call depth for each thread. The developer may set a function call depth limit where the function calls are no longer recorded in order to avoid the recursive algorithm generating large amounts of recorded data. In the work by Hollingsworth in [3] the collected data were limited by only considering a few functions to be traced. The functions were selected by hand by the user. However, the technique by Hollingsworth will not automatically avoid recursive function calls generating large amounts of collected data.

The Simulator is extended with the algorithms discussed in Section 2. All the different synchronization primitives in the Solaris 2.X thread library are handled, including semaphores, mutexes, read–write locks, and condition variables as well as the creation and joining of threads.

The Simulator is used to obtain the execution of the program with one thread per processor. After the (simulated) execution is obtained, the extended critical path algorithm can be applied with any number of processors as the argument, as discussed in Section 2.3, and the result is displayed on a function level as discussed in Section 2.2 and by making the lines thicker in the Gant diagram for the segments in the extended critical path.

4. THREE PRACTICAL EXAMPLES

4.1. Parallel Quick Sort

In this example we have used a parallel version of the quick sort algorithm. We sorted 2,000,000 random integers. The algorithm used is shown in pseudo-code in Fig. 9. As can be seen, new threads will be created as long as the number of items to be sorted is greater than 100,000.

The structure of the program can be illustrated as a tree, where the function `RecursiveQuickSort` represents the leaves in the tree and the function `ParallelQuickSort` the other nodes. Since the tree is unbalanced there will always be one leaf that is the last to finish execution. This is illustrated in Fig. 10. Most of the work is performed in the leaves. With one processor per thread only the path to the latest leaf will be considered and in this path the most time will be spent in the function `ParallelQuickSort`. The path is shown in Fig. 10 with the dashed line.

```
int data[2000000];

ParallelQuickSort(int left, int right) {
/* Sort the integers to the left and right of the selected pivot.
Calculate start and end index for the next calculations to left and right. */
  if(right - left > 100000) {
    thread_create(ParallelQuickSort, leftStart, leftEnd);
    thread_create(ParallelQuickSort, rightStart, rightEnd);
  }
  else {
    RecursiveQuickSort(leftStart, leftEnd);
    RecursiveQuickSort(rightStart, rightEnd);
  }
}

RecursiveQuickSort(int left, int right) {
/* Sort the integers to the left and right of the selected pivot.
Calculate start and end index for the next calculations to left and right. */
    RecursiveQuickSort(leftStart, leftEnd);
    RecursiveQuickSort(rightStart, rightEnd);
}
```

FIG. 9. Pseudo-code for the ParallelQuickSort algorithm.

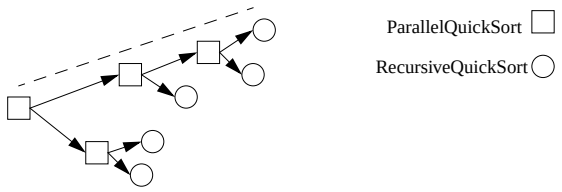


FIG. 10. An illustration of the tree. The length of the edges illustrates how much data are to be sorted by the following node.

When the extended critical path analysis is applied, it yields that with a few processors (five or less) the recursive function will dominate the extended critical path. However, with six processors or more our analysis shows that the parallel function dominates. Thus, depending on the number of processors in the target machine, the most suitable function for optimization will be different. This is shown in Fig. 11.

We optimized the functions with approximately 20%. A verification of the quick sort program was made on a Sun Enterprise 4000 with eight processors. The result is found in Fig. 12 and shows that the sort program behaves as predicted in Fig. 11; i.e., the crossover occurs between five and six processors.

The execution overhead for the instrumentation in this example is less than 2%. This overhead includes all instrumentation needed to do the simulations as well, not only the data needed for extended critical path analysis.

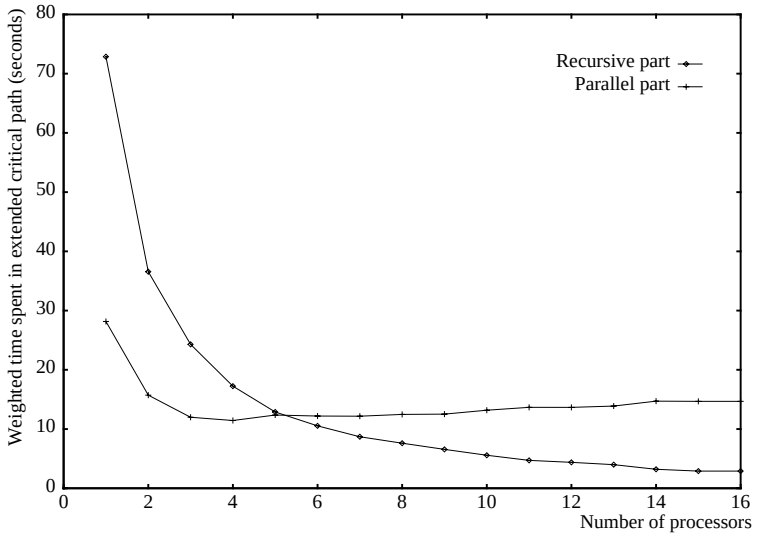


FIG. 11. The time for the recursive sort function and the parallel sort function spent in the extended critical path on different numbers of processors.

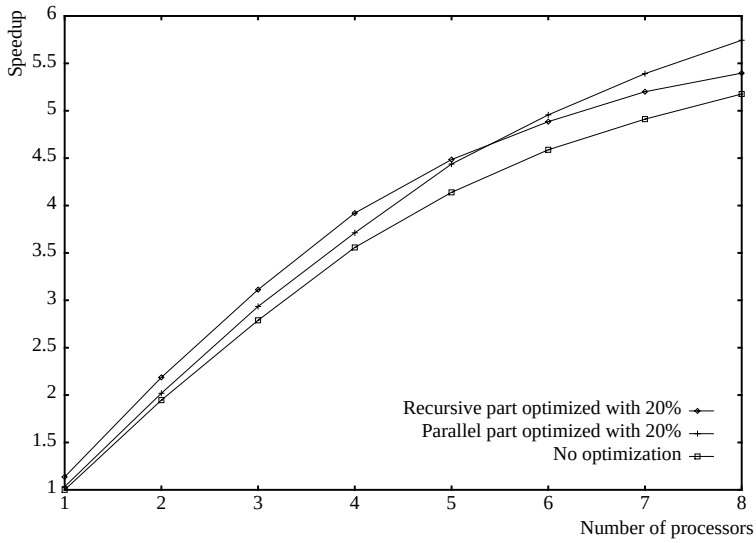


FIG. 12. Validation of the predictions on a multiprocessor with eight processors.

4.2. Finding the n Largest $h(x)$ for N Numbers x_i

The second application used in this study is a general search problem. The problem is to find the n largest $h(x)$, for N numbers x_i $(1 \leqslant i \leqslant N)$ where $n \ll N$. If the function $h(x)$ has local maximas and is computationally expensive then searching

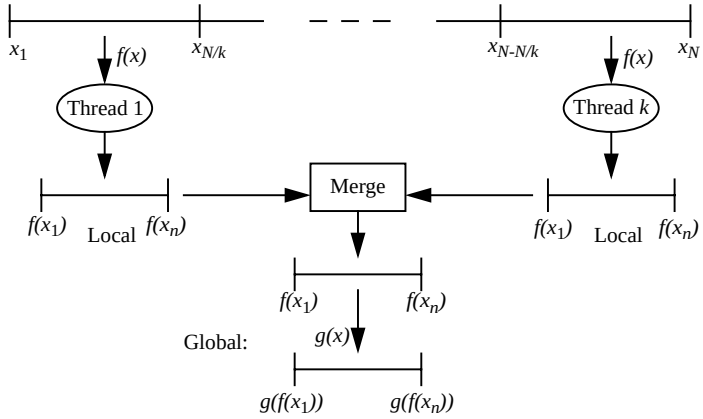


FIG. 13. Schematic picture of the search principle.

$$N = 10000000$$
$$n = 5000$$
$$k = 8$$
$$h(x) = \sum_{n=0}^{671} e^{\sqrt{\left|\left|x + 2x^2\right| + n\right|}}, \text{ which gives us}$$
$$f(x) = \left|x + 2x^2\right| \text{ and } g(x) = \sum_{n=0}^{671} e^{\sqrt{|x+n|}}$$

FIG. 14. The parameters and functions used for the study.

through each $h(x_i)$ would be computationally expensive. If it is possible to identify two functions $f(x)$ and $g(x)$ such that $h(x) = g(f(x))$ where $f(x)$ is computationally inexpensive and $g(x)$ is monotone, then the following solution would be applicable. Assume we have k threads. Apply $f(x)$ on each x_i with a subset of N/k elements for each thread and find the n largest $f(x_i)$ for each thread. Merge the n largest $f(x_i)$ for all threads into one vector containing the n largest $f(x_i)$ for the entire set $x_i (1 \leqslant i \leqslant N)$. Then $g(x)$ will only have to be applied to n numbers at the end. In Fig. 13 a principal sketch is found.

When applying this algorithm with the parameters found in Fig. 14 we get the execution times for $f(x)$ and $g(x)$ in the extended critical path as found in Fig. 15. As can be seen it is in this case most beneficial to optimize $f(x)$ for up to four processors, whereas $g(x)$ is most beneficial to optimize for more than four processors.

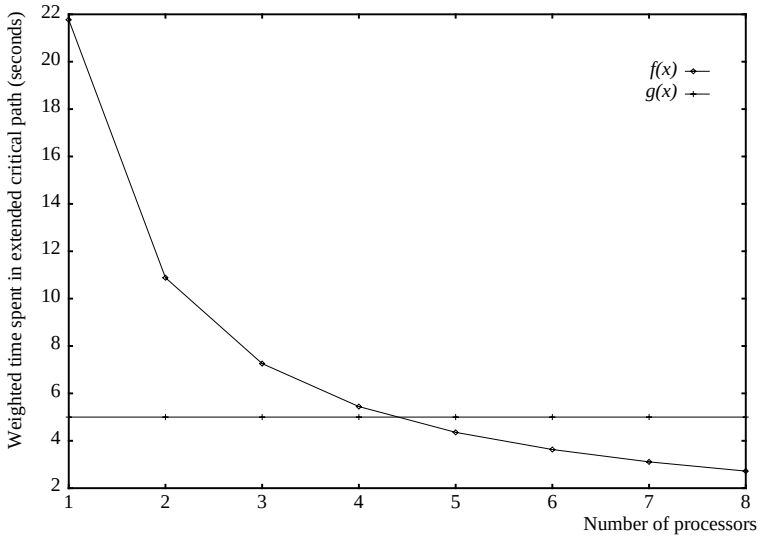


FIG. 15. The time $f(x)$ and $g(x)$ spent in the extended critical path on different numbers of processors.

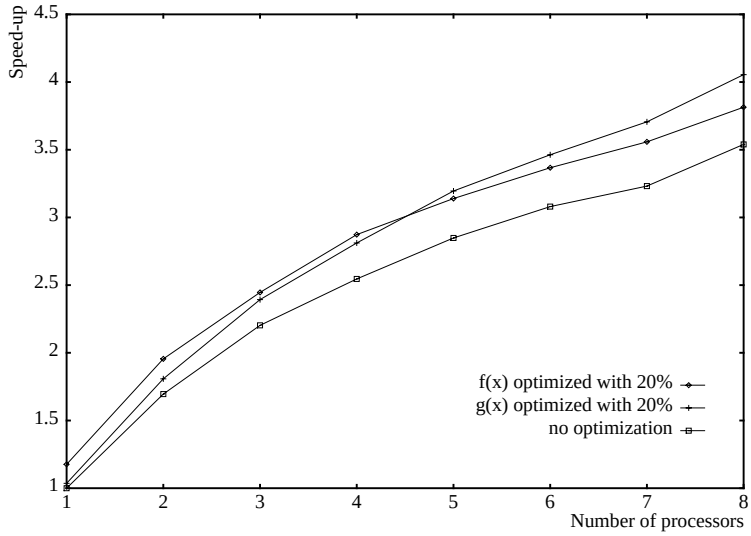


FIG. 16. Validation of the predictions on a multiprocessor with eight processors.

This is verified on an eight-way multiprocessor, as shown in Fig. 16 where $f(x)$ has better speedup than $g(x)$ for four processors or less. The situation is the opposite for five processors or more.

4.3. Billing Gateway

The third application used in this study is based on a commercial telecommunication application built by Ericsson, called BGw (billing gateway) [4]. We used a skeleton version of the BGw for our validation because the real BGw uses a lot of I/O, which is currently not supported by the extended critical path. The original BGw consists of about 100,000 lines of C++ code.

A principal sketch over the BGw (skeleton) is found in Fig. 17. The BGw (skeleton) works as a kind of filter. The Readers get the information to be filtered. As soon as all data are received the information is stored on disk, the disk is synchronized, i.e., all data are physically written to disk, and the receiver is ready for the next chunk of data. The Sorter reads the file created by the Reader, sorts and converts the data, and feeds two Writers. The Writers then read the data and send it further in the system. The skeleton had no I/O and consists of two Readers, two Sorters, and four Writers as shown in Fig. 17. Each Reader got three packets of information.

If the extended critical path analysis (shown in Fig. 18) is applied, it is shown that with a few processors (one or two) the Writer will dominate the extended

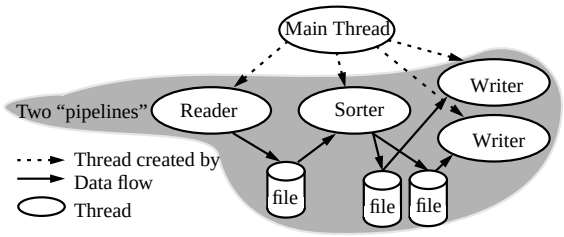


FIG. 17. The organization of the BGw skeleton. In the application used the data were *not* stored on file.

critical path. However, with more than two processors our analysis shows that the Sorter dominates. Ordinary profiling tools, such as Quantify, would indicate Writer as the most beneficial to optimize. Tools that do critical path analysis for an unlimited number of processors would identify Sorter as the most beneficial to optimize. The hump for the Sorter at three to four processors in Fig. 18 is due to the effect of segments with a weight greater than one as previously shown in Fig. 7.

Experimental results on an eight-processor Sun Enterprise 4000 show that Writer is the most beneficial to optimize on one or two processors and Sorter is more beneficial to optimize on more than two processors. This is shown in Fig. 19 where Writer, Sorter, and Reader have been optimized by approximately 20% each. The

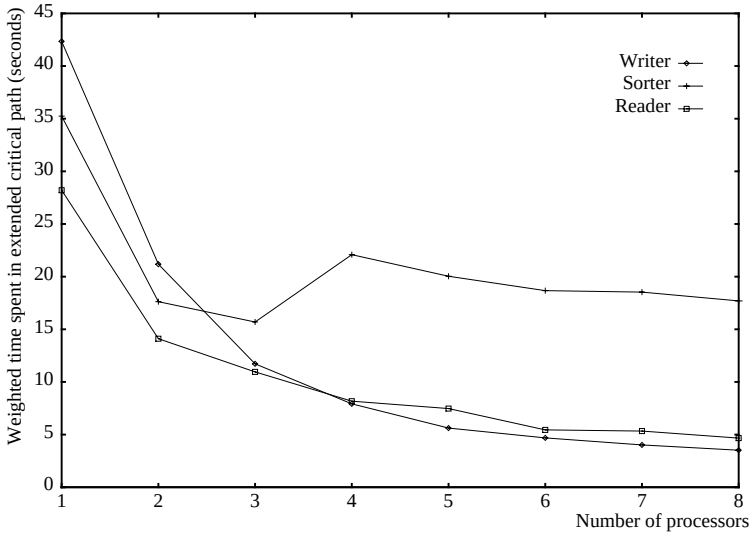


FIG. 18. The time the three function spent in the extended critical path on different number of processors.

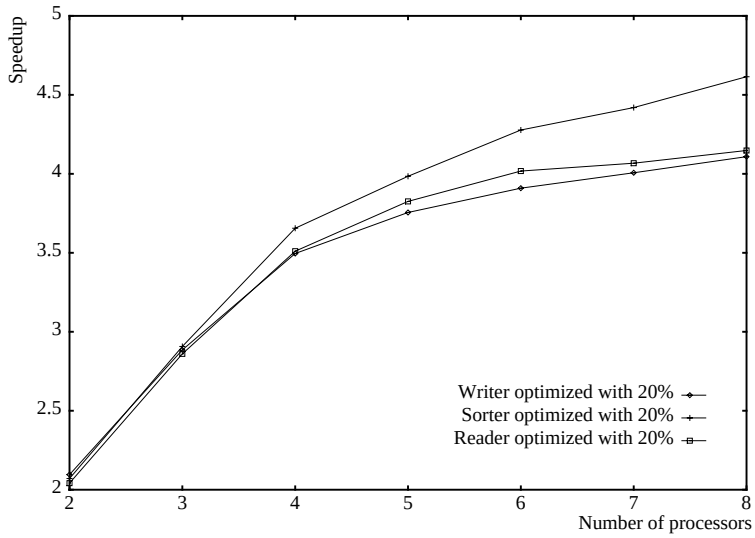


FIG. 19. Execution time for Writer, Sorter, and Reader when optimized with approximately 20% each, normalized to a nonoptimized execution time. The nonoptimized execution and the case with one processor are omitted for readability.

breakpoint when Sorter is more beneficial to optimize than Writer occurs between two and three processors as predicted. We can also see that the crossing point between Reader and Writer is between three and four processors as predicted in Fig. 18.

5. DISCUSSION

The examples in Section 4 show that the function dominating the extended critical path may change when a different number of processors is used for the same application. The extended critical path analysis described here assumes that the scheduling is ideal with infinitely small time slices and no scheduling overhead. Even if this assumption is quite close to the real world [7], it still differs from a real execution. Although not implemented, the tool (VPPB) could be extended with the capability to simulate the execution with one or several functions optimized to any degree. The result will be a simulation that will mimic the scheduling of the Solaris operating system with the optimized application. The complexity of the extended critical path algorithm is manageable, the three examples used for verification in Section 4 took at most 0.2 s to analyze on an ordinary workstation (300 MHz Sun Ultra10).

The extended critical path analysis is developed for CPU intensive applications and I/O has not been addressed. The tool (VPPB) is capable of recording and simulating I/O behavior. A simple approach for adding I/O capability for the extended critical path analysis is to approximate the time the thread waits for the I/O operation with a sleep for the same period of time. The result is then that the thread does not use the processor while waiting for an I/O operation and the extended critical path continues through the I/O operation and continues on the same thread. With this approach, no change in the extended critical path analysis is needed. However, I/O could also be used to (implicitly or explicitly) synchronize between threads. If one thread reads from a file that is empty the thread will be blocked until another thread writes something to the same file. The effect is then a synchronization. By recording the corresponding file descriptor for each I/O operation these dependencies could be solved and I/O will behave more or less as any other lock. The key difference between ordinary locks and the I/O synchronization is, however, that it is hard for the Simulator to know if the file is empty or not at the beginning of execution of the program.

The extended critical path analysis has been addressed for SMPs in this paper. The technique can be modified for message passing applications on distributed memory machines as well. The algorithm must then be modified to manage that a message will take some time to reach its destination. The recording will be local on each node and after execution is done the recorded material will be merged together for all nodes in order to apply the extended critical path algorithm. Message passing applications often use one process–thread per processor and thus the work by Hollingsworth [3] will be appropriate. However, there is an ongoing trend in the message passing community to include thread primitives in the message passing libraries. Then, the situation may occur that there are more threads on one single node than the number of processors on that node, and the technique by Hollingsworth will not be appropriate any longer. Our technique will manage that. For NUMA machines (nonuniform memory architecture) the extended critical path algorithm does not need any modification at all. However, the simulated execution that the algorithm makes use of must be made according to the memory delays for the specific machine. When considering distributed applications, the problem is the same as for a message passing application, but it may be harder to keep a synchronized clock for the distributed machines. A synchronized clock is needed when merging the local log files before applying the extended critical path algorithm.

The extended critical path algorithm can be applied to languages other than C/C++ with Solaris threads. For instance, Java could be used as well. Java includes primitives for critical regions and monitors. Critical regions and monitors can be implemented with the Solaris thread primitives mutex and condition variables. Thus, it is feasible to use the extended critical path analysis for Java as well. It is worth noting that the specification of the Java virtual machine, described in [6] by Lindholm and Yellin, does not say if the Java threads can actually make use of more than one processor. Threads that cannot make use of more than one processor are called green threads, whereas threads that can use more than one processor are called red threads (or native threads). It is the Java virtual machine that decides if a Java thread will be green or red; this cannot be decided in the programming language.

6. RELATED WORK

The critical path analysis is a well-known technique [3, 8, 18]. However, all these approaches originate from the message passing area where the common case is one thread (process) per processor. The works in [3, 8, 18] are based on a PAG (program activity graph) where the critical path is defined as "the longest, time-weighted sequence of events from start of the program to its termination." This means that analysis is done with no limitation on the number of processors available and shows only the points at 16 processors in Fig. 11. However, it is not unusual to have several threads competing for a lesser number of processors in a multithreaded application. In [18] this issue is addressed briefly by using a CPU-based timer when measuring the length of the edges in the PAG. Even if a process is scheduled off the process, the time will be counted to the corresponding edge in the PAG. Thus, when doing the critical path analysis on the PAG the scheduling will be included. The drawback to this solution is that the critical path will not include the process that executed instead. An example with two processes, where process 1 calculates for 12 time units and process 2 acts as a watchdog executing five times for 1 time unit every second time unit, demonstrates this. The example is illustrated in Fig. 20 when executing on a single processor. Given the solution by Yang and Miller in [18] process 1 will have the total length of 17 and thus will be the longest path and be called the critical path. However, optimizing process 2 will also yield a reduced completion time in this example. The critical path by Yang and Miller in [18] will not identify that. Our extended critical path algorithm handles situations like the one in Fig. 20.

In [8], interest is also focused on the second longest path, etc., in the application. This is because if the critical path is optimized to the extent that it will no longer be the longest path, then the second longest path will be the critical path. This technique will not be appropriate for a multithreaded application with more threads executable than processors since the optimization of one function may affect how the application will be scheduled by the operating system.

Other optimization tools for multiprocessors only identify the critical path on one processor. The Solaris program `tha` [15] is designed to work as `prof` [14] with the difference that the information collected is on a per thread basis. `Tha` collects `prof` information per thread and yields correct information per thread. However, there is no information about the execution flow and dependencies between the threads. Simply adding all threads' accumulated execution times for a given function would yield the same information as in Quantify [13]. While `tha` supports better information, it also has some major drawbacks described in [15]; e.g., it is not possible to use the standard C++ I/O primitives.

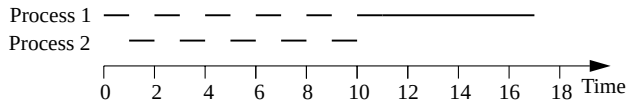


FIG. 20. Two processes executing on one processor.

Another way of presenting the performance problems of multithreaded programs is based on contention, as in Tmon [5]. The contention is based on locks and context switching overhead. Tmon use two single processor workstations, one to execute the multithreaded program and the other to gather the recorded data. The data are analyzed, to some extent, in real time. This setup requires a fast interconnection between the single processors. The applicability of Tmon on multiprocessors is not addressed in [5].

There are other tools that support simulation of a parallel program behavior and then visualize the result [9–12, 17]. However, they do not support (extended) critical path analysis.

7. FUTURE WORK

The extended critical path is shown in the VPPB tool as thicker lines in the Gant diagram. The information about different functions is simply a list of metrics per function. The use of a function call graph found in many conventional profiling tools, e.g., Quantify [13], is not directly applicable, since the extended critical path may move from one thread to another due to synchronizations. The synchronizations may be placed deep down in the functions and thus the extended critical path jumps from a function called deep down in one thread to another function deep down in another thread. The simple call graph will not cope with this kind of jump. An issue for future work is, therefore, to improve the visualization and representation of the extended critical path for the developer.

The simple approach for including I/O in the extended critical path analysis discussed in Section 5 could also be addressed as future work. However, full support for I/O will require further investigations and could be, as such, a future development of the extended critical path analysis.

Another thing that would be appropriate for future work is to identify by how much a function can be optimized before another function will become the function that the most time is spent in. On a single processor it is possible to optimize a function until the total execution time for that function is less than some other function. It is not that simple on a multiprocessor executing more threads than processors, since optimizing one function may alter the scheduling order and another segment may become a part of the extended critical path. Also along those lines, support for elaborating with different optimizations of functions directly in the tool would be interesting. Then it would be possible to allow the developer to specify how much a certain function can be optimized (based on an estimation) and rerun the *simulation* with the "pseudo-optimized" function. After that, the extended critical path analysis can be applied on the rerun simulation.

8. CONCLUSION

Traditional performance optimization is done when the program is written. The main goal is to increase the performance of the application. The existence of a number of commercial profiling tools [13–15] shows the importance of the

optimization task. Multiprocessors are used for the same reason, i.e., to increase the performance. Performance optimization for programs designed for multiprocessors is at least as important as optimization of sequential programs, because high performance requirements are often a major reason for using multiprocessors in the first place.

In the case of a multithreaded program executing on a multiprocessor it is not certain that all executed code segments will add to the completion time. A simple example is a watchdog, which in the single processor case will be a part of the completion time. However, on a multiprocessor, the watchdog may execute on its own processor. The watchdog will then not affect the completion time of the program.

Traditional profilers, such as Quantify [13] and `tha` [15], give misleading information about where in the code to concentrate the optimization efforts. In some cases Quantify will actually give the worst possible indications. The reason these kind of tools give misleading information is that they assume that all executed code segments contribute to the completion time.

To implement efficient performance optimization we must concentrate the efforts on the critical path. The critical path as it is addressed in [3, 8, 18] will not fit our need when more threads are able to execute than there are processors available. Our extended critical path analysis is dependent on the synchronization behavior in the multithreaded program and the number of processors. In this paper an algorithm to find the extended critical path has been presented. The algorithm manages not only an ideal situation when there are as many threads as processors, but also when there are fewer processors than threads.

The possibility for the developer to connect the extended critical path to those functions that are part of the extended critical path is important. We have presented an algorithm that does so on a multiprocessor. The algorithm also shows the amount of time spent in the functions during the extended critical path.

These methods have been implemented in a performance optimization tool called VPPB [1, 2]. The tool pinpoints what parts of the code to optimize without the need of a multiprocessor. The current platform for the tool is the Solaris 2.X operating system; thus, the tool is applicable to a large number of industrial applications.

The usefulness of the method has been demonstrated by using the tool on three parallel programs. The correctness of the predictions was verified on an eight-processor SMP.

ACKNOWLEDGMENT

This work was supported in part by ARTES and the Swedish Foundation for Strategic Research.

REFERENCES

1. 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, IEEE Computer Society, Los Alamitos, CA, 1999.

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, IEEE Computer Society, Los Alamitos, CA, 1998.
3. J. Hollingsworth, Critical path profiling of message passing and shared-memory programs, *IEEE Trans. Parallel Distrib. Systems* **9**, 10 (October 1998), 1029–1040.
4. D. Häggander and L. Lundberg, Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor, *in* "Proc 1998 Int'l Conf. on Parallel Processing," pp. 262–269, IEEE Computer Society, Los Alamitos, CA, 1998.
5. M. Ji, E. Felten, and K. Li, Performance measurements for multithreaded programs, *Perform. Eval. Rev.* **26**, 1 (June 1998), 161–170.
6. T. Lindholm and F. Yellin, "The Java[tm] Virtual Machine Specification," second ed., Addison–Wesley, Reading, MA, 1999.
7. L. Lundberg and M. Roos, Predicting the speedup of multithreaded Solaris programs, *in* "Proc. 4th International Conference on High-Performance Computing," pp. 386–1392, IEEE Computer Society, Los Alamitos, CA, 1997.
8. B. Miller, M. Clark, J. Hollingsworth, S. Kiersead, S.-S. Lim, and T. Torzewski, IPS-2: The second Generation of a Parallel Program Measurement System, *IEEE Trans. Parallel Distrib. Systems* **1**, 2 (April 1990), 206–217.
9. E. Papaefstathiou, D. Kerbyson, G. Nudd, and T. Atherton, An overview of the CHIP^{3}S performance prediction toolset for parallel systems, *in* "Proc. 8th ISCA International Conference on Parallel and Distributed Computing Systems," pp. 527–533, 1995.
10. V. Pillet, J. Laboarta, T. Cortes, and S. Girona, "PARAVER: A Tool to visualize and Analyse Parallel Code," CEPBA/UPC Report RR-95/03, University of Politencia, Catalonia, 1995.
11. S. Sarukkai and D. Gannon, SIEVE: A performance debugging environment for parallel programs, *J. Parallel Distrib. Comput.* **18**, 2 (June 1993), 147–168.
12. Z. Segall and L. Rudolph, PIE: A programming and instrumentation environment for parallel processing, *IEEE Software* **2**, 6 (November 1985), 22–37.
13. Rational, Quantify version 4.2, http://www.rational.com/products/quantify.
14. Sun Man Pages, prof, Sun Microsystems Inc., 1993.
15. Sun Man Pages, tha, Sun Microsystems Inc., 1996.
16. SunSoft, "Solaris Multithreaded Programming Guide," Prentice–Hall, Englewood Cliffs, NJ, 1995.
17. S. Toledo, PERFSIM: A tool for automatic performance analysis of data-parallel Fortran programs, *in* "Proc. 5th Symposium on the Frontiers of Massively Parallel Computation," pp. 396–405, IEEE Computer Society, Los Alamitos, CA, 1994.
18. C.-Q. Yang and B. Miller, Critical path analysis for the execution of parallel and distributed programs, *in* "Proc. 8th International Conference on Distributed Computing Systems," pp. 366–375, IEEE Computer Society Press, Washington, DC, 1988.

Paper VII

A Method for Bounding the Minimal Completion Time in Multiprocessors

Magnus Broberg, Lars Lundberg, and Kamilla Klonowska
Research Report 2002:03, 2002

VII

A Method for Bounding the Minimal Completion Time in Multiprocessors

Magnus Broberg, Lars Lundberg, and Kamilla Klonowska

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Soft Center, S-372 25 Ronneby, Sweden
Phone: +46-(0)457 385822
Fax: +46-(0)457 27125
Magnus.Broberg@bth.se, Lars.Lundberg@bth.se, Kamilla.Klonowska@bth.se

Abstract

The cluster systems used today usually prohibit that a running process on one node is reallocated to another node. A parallel program developer thus has to decide how processes should be allocated to the nodes in the cluster. Finding an allocation that results in minimal completion time is NP-hard and (non-optimal) heuristic algorithms have to be used. One major drawback with heuristics is that we do not know if the result is close to optimal or not.

In this paper we present a method for finding a guaranteed minimal completion time for a given program. The method can be used as a bound that helps the user to determine when it is worth-while to continue the heuristic search. Based on some parameters derived from the program, as well as some parameters describing the hardware platform, the method produces the minimal completion time bound. The method includes an aggressive branch-and-bound algorithm that has been shown to reduce the search space to 0.0004%. A practical demonstration of the method is presented using a tool that automatically derives the necessary program parameters and produces the bound without the need for a multiprocessor. This makes the method accessible for practitioners.

Key words: analytical bounds, minimal completion time, parallel programs, multiprocessors, clusters, processor allocation, branch-and-bound, development tool

1. Introduction

Multiprocessors are often used to increase performance. In order to do this, the program processes have to be distributed over several processors. Finding an efficient allocation of processes to processors can be difficult. Clusters of computers use communication networks to send messages between the processes. In almost all cases it is impossible to (efficiently) move a process from one computer to another, and static allocation of processes is thus essential and an unavoidable aspect of such systems.

Even if we consider shared memory multiprocessors, which are often built using a distributed memory approach, we have to consider the allocation issues. Although the network connecting the processors is of high capacity the time for accessing remote memory is 3 to 10 times longer than accessing local memory [21]. In order to avoid that a process is scheduled on different nodes, and thus make the working set for the process into remote memory accesses, one would like to statically bind/allocate the process to a node in the system. The synchronizations between processes will still be performed remotely.

Finding an allocation of processes to processors that results in minimal completion time is a classic allocation problem that is known to be NP-hard [7]. Therefore, heuristic algorithms have to be used, and this results in solutions that may not be optimal. A major problem with the heuristic algorithms is that we do not know if the result is near or far from optimum, i.e. we do not know if it is worth-while to continue the heuristic search for better allocations.

In this paper we present a method for finding a completion time that, given a certain program, can be achieved. The method can be used as an indicator for the completion time that a good heuristic ought to obtain. The method produces the minimal completion time given some parameters derived from the program as well as some parameters describing the hardware platform. The produced performance bound is optimally tight given the information that we have available about the parallel program and the target multiprocessor.

The result presented here is an extension of previous work [17][18]. The main difference between this result and the previous result is that we now can take network communication time and program granularity into consideration. A practical demonstration of the method is presented at the end of the paper.

2. Definitions and main result

A parallel program consists of a set of sequential processes. The execution of a process is controlled by two synchronization primitives: Wait(Event) and Activate(Event), where Event couples a certain Activate to a certain Wait. When a process executes an Activate on an event, we say that the event has occurred. It may, however, take some time for the event to travel from one processor to another. We call that time the synchronization latency t. If a process executes a Wait on an event which has not yet occurred, that process becomes blocked until another process executes an Activate on the same event and the time t has elapsed. However, if both processes are on the same processor, we assume time for the event to travel to be zero, i.e. zero synchronization latency. A process executing a Wait on an event which has occurred more than t time units before does not become blocked. However, if the event occurred less than t time units ago the process executing a Wait on the event has to block for the remaining part of t, unless both processes reside on the same processor (we assume time for the event to travel to be zero within a processor).

Each process can be represented as a list of sequential segments, which are separated by a Wait or an Activate (see Figure 1). We assume that, for each process, the length and order of the sequential segments are independent of the way processes are scheduled. All processes are created at the start of the execution. Some processes may, however, be initially blocked by a Wait, thus imitating the behaviour that one process creates another process. Under these conditions, the minimal completion time for a program *P*, using a system with k processors, a latency of t and a specific allocation denoted A, is $T(P, k, t, A)$. We further find the minimal completion time for a program *P*, using a system with k processors, a latency of t, as $T(P, k, t) = min_A T(P, k, t, A)$.

The left part of Figure 1 shows a parallel program consisting of three processes (P1, P2, and P3). Sequential processing is represented by a procedure Work, i.e. Work(x) denotes sequential processing for x time units. Process P1 cannot start its execution before P2 has started. This dependency is represented with a Wait on event 1 in P1. The right part shows a graphical representation of P and two schedules resulting in minimum completion time for a computer with three ($T(P, 3, t)$) and two processors ($T(P, 2, t)$), respectively. We assume that each parallel program has a well defined start and end, i.e., that there is some code that is always executed first and some other code that is always executed last. The thin slices in the beginning and end of P2 represent such a well defined start and end of the program, no actual execution is performed since there is no corresponding Work. The left part of Figure 1 (two processors) shows local scheduling, which means that two or more processes share the same processor. The local schedule, i.e., the order in which processes allocated to the same processor are scheduled affects the completion time of the program. We assume optimal local scheduling when calculating $T(P, k, t)$.

```
process P1
begin
   Wait(Event_1);
   Work(1);
   Activate(Event_3);
   Work(1);
   Activate(Event_4);
end P1;
```

```
process P2
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Work(2);
   Wait(Event_4);
   Wait(Event_5);
end P2;
```

```
process P3
begin
   Wait(Event_2);
   Work(2);
   Activate(Event_5);
end P3;
```

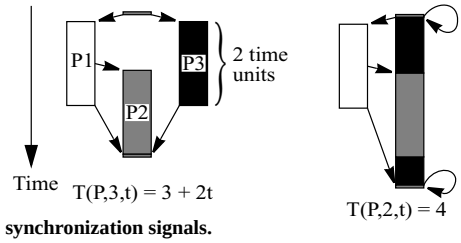


Figure 1: Program *P* with synchronization signals.

For each program P with n processes there is a parallel profile vector V of length n. Entry i in V $(1 \leq i \leq n)$ contains the fraction of the completion time during which there are i active processes, using a schedule with one process per processor and no synchronization latency. The completion time for a program P with n processes, using a schedule with one process per processor and no synchronization latency is denoted $T(P, n, 0)$. $T(P, n, 0)$ is fairly easy to calculate. In Figure 1, the completion time for the parallel program, using a schedule with one process per processor and no synchronization latency is 3 time units, i.e. $T(P, n, 0) = 3$. During time unit one there are two active processes (P1 and P3), during time unit two there are three active processes (P1, P2, and P3), and during time unit three there is one process (P2), i.e. $V = (1/3, 1/3, 1/3)$. Different parallel programs may, obviously, yield the same parallel profile vector.

For each program P there is a granularity, denoted z, that represents the program's synchronization frequency. By adding the work time of all processes in program P, disregarding synchronization, we obtain the total work time of that program. The number of synchronization signals in program P is divided by the total work time for a program in order to get the granularity, z. In the example in Figure 1 the granularity equals $5/6$.

The completion time is affected by the way processes are allocated to processors. Finding an allocation which results in minimal completion time is NP-hard. However, in this paper we will show that a function $p(n, k, t, z, V)$ can be calculated such that for any program P with n processes, granularity z, and a parallel profile vector V: $min_A(T(P, k, t, A)) = T(P, k, t) \leq p(n, k, t, z, V)T(P, n, 0)$. The function $p(n, k, t, z, V)$ is optimal in the sense that for at least some program P, with n processes, granularity z, and a parallel profile vector V: $min_A(T(P, k, t, A)) = T(P, k, t) = p(n, k, t, z, V)T(P, n, 0)$. Consequently, for all programs P with n processes, granularity z, and a parallel profile vector V: $p(n, k, t, z, V) = max_P(T(P, k, t)/T(P, n, 0))$.

The outline for this paper is found in Figure 2. In Section 3 we will show some transformation techniques that allow us to split the program into two parts. The first part includes all the execution time and the other consists of synchronizations only. Then the two parts will be examined resulting in an analytical model for each part in Section 4 and Section 5, respectively. In Section 6 we combine the results from the two parts into a single result that covers the whole program. Following that there is a section where we practically demonstrate the use of this method using a tool. Towards the end we have some discussion and related work. Finally, we have the conclusions.

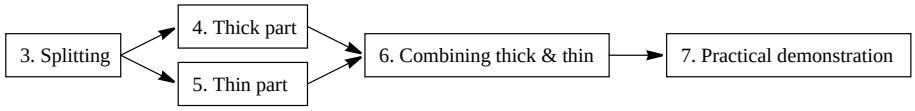


Figure 2: The outline of this paper.

3. Splitting the program into one thick part and one thin part

In this section we will look at techniques, that transform a program P into two parts, one thin part only consisting of synchronizations, and the other part consisting of all the execution time. We will then discuss the thick and thin parts separately in Section 4 and Section 5, respectively.

3.1. Obtaining P' as *m* identical copies of program P

We construct a program P' that is m copies of program P.

Lemma 1: $T(P, k, t)/T(P, n, 0) = T(P', k, t)/T(P', n, 0)$.

Proof: Having m $(m > 1)$ copies of program P, means that we multiply both (P, k, t) and $T(P, n, 0)$ by m: $T(P, k, t)/T(P, n, 0) = mT(P, k, t)/mT(P, n, 0) = T(P', k, t)/T(P', n, 0)$. ■

Figure 3 shows the transformation of the program P into m copies of this program, denote as P'. Program P (left part in the figure) consists of execution time and synchronization signals.

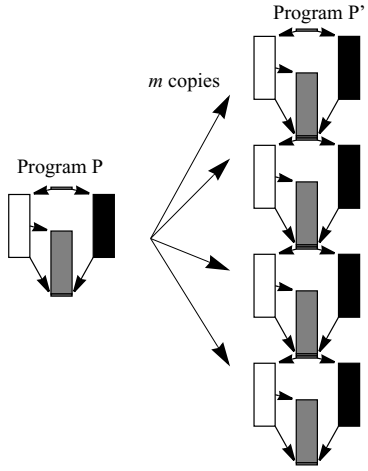


Figure 3: Transformation *P* into *P'*.

3.2. Replacing four copies of a program with three new programs

We prolong each time unit $x, x > 0$ of each process in program P by Δx and get program P'. Program P' is then transformed into P'' in the same way, i.e. after the transformation each time unit $x + \Delta x$ is prolonged with Δx.

In the case with one process per processor and no communication cost the difference of the completion time of $T(P'', n, 0) - T(P', n, 0) = T(P', n, 0) - T(P, n, 0) = \Delta x\, T(P,n,0)/x$.

The situation with synchronization latency is more difficult. Since the synchronization cost in this case is not zero, the differences after prolongation are not always equal, because we do not prolong the synchronizations themselves. Let ΔL denote the difference in length between P and P'. If we denote the length of program P with L, the length of P' will be $L' = L + \Delta L$. In the same way we create P'' with a difference between P' and P'' called $\Delta L'$. The length of P'' will then be the length of $L'' = (L + \Delta L) + \Delta L'$ (see Figure 4).

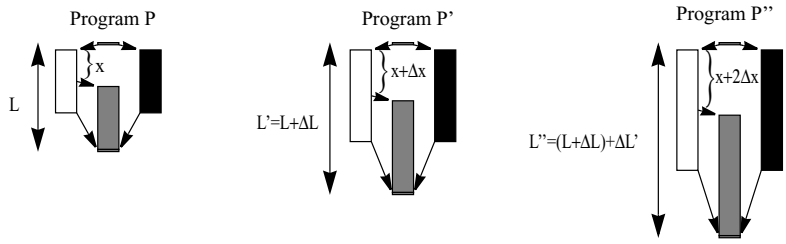


Figure 4: The transformation of the program P by prolongation of the processes.

In order to discuss the effects of local scheduling separately, we will assume that there is only one process per processor. We will relax this restriction at the end of this section (Section 3.2). The *critical path* is defined as the longest path from the start to the end of the program following the synchronizations. In the case when two (or more) paths are the longest, the path with the minimum number of arrows (synchronizations) is the critical path.

Let $arr(P)$ be a number of arrows in the critical path in the program P, and $arr(P')$ be the number of arrows in the critical path in P' (i.e. P after the prolongation).

Lemma 2: $arr(P) \geq arr(P')$.

Proof: Suppose that $arr(P) = x, (x \geq 0)$ and that it in program P there is another path that consists of more than x arrows. Because we prolong the processes, the path with more arrows (and thus less execution) increases slower than the critical path. Consequently, the path with more arrows never can be longer than the critical path.

■

Then of course: $arr(P') \geq arr(P'')$.

When we prolong a program the critical path may change its way (see Figure 5). This happens when path two is longer than path one, and path two has less execution (and thus more synchronizations) than path one. As a consequence of the prolongation, path one will grow faster than path two. At a prolongation of a given Δx the resulting program P' with the paths one and two will have the same length. Adding yet a Δx will yield program P'' where path one is the longest path.

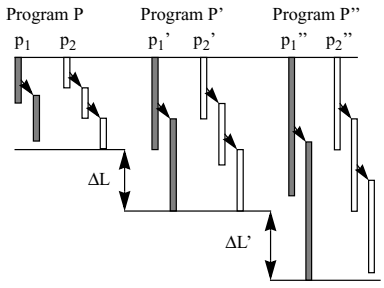


Figure 5: Path p_1 grows faster than p_2 when we add Δx.

Theorem 1: $\Delta L \leq \Delta L'$.

Proof: Let $E_1 = y_1 + arr(E_1)t$ be the length of path one, where y_1 is the sum of the lengths of the segments in path one, $arr(E_1)$ is the number of arrows with the communication cost t. Let $E_2 = y_2 + arr(E_2)t$ be the corresponding length for path two. Further, let $y_1 > y_2$ and path two be the critical path, i.e. $E_1 < E_2$. Then let $E_1' = y_1\frac{x + \Delta x}{x} + arr(E_1)t$ and $E_2' = y_2\frac{x + \Delta x}{x} + arr(E_2)t$. Let also $E_1'' = y_1\frac{x + 2\Delta x}{x} + arr(E_1)t$ and $E_2'' = y_2\frac{x + 2\Delta x}{x} + arr(E_2)t$. There are three possible alternatives (provided that there are only these two paths in the system):

- $E_1' < E_2'$ and $E_1'' < E_2''$. In this case $\Delta L = \Delta L'$, the critical path does not change.
- $E_1' < E_2'$ and $E_1'' \geq E_2''$. In this case $\Delta L < \Delta L'$, since path one grows faster than path two when we add Δx.
- $E_1' \geq E_2'$ and $E_1'' > E_2''$. In this case $\Delta L < \Delta L'$, since path one grows faster than path two when we add Δx.

If there are more than two paths in the program, we may have another path (besides path one and two) that becomes the critical path in *P''*. In this case we get $\Delta L < \Delta L'$, since the new path must contain more processing (and less synchronizations) and thus grow faster than paths one and two when we add Δx.

■

According to Theorem 1 we have:

Theorem 2: $2L' \leq L + L''$.

Proof: $2L' = L + \Delta L + L + \Delta L \leq L + \Delta L + L + \Delta L' = L + (L + \Delta L + \Delta L') = L + L''$.

■

This means that the length of two copies of P' is less than or equal to the length of P plus the length of P''.

We will now look at the case where there can be more than one process on each processor. Let programs *P, P'* and *P''* be programs where there are more than one process on some processors; *P, P'* and *P''* are identical except that each work-time in *P''* is twice the corresponding work-time in *P'*, and all work-times are zero in *P.* Consider an execution of *P''* using allocation *A* and optimal local scheduling. Let *Q''* be a program where we have merged all processes executing on the same processor into one process. Figure 6 shows how processes *P*2'' and *P*3'' are merged into process *Q*2''. Let *Q'* be the program which is identical to *Q''* with the exception that each work-time is divided by two. Let *Q* be the program which is identical to *Q''* and *Q'* except that all work-times are zero.

From Theorem 2 we know that $2T(Q', k, t, A) \leq T(Q, k, t, A) + T(Q'', k, t, A)$. We use the same allocation *A* for both *P"* and *Q"*. However, since there are less processes in *Q"* we ignore the allocation of non-existing processes in *Q"*, i.e. each process in *Q"* is allocated to a processor of its own. From the definition of *Q"* we know that *T*(*P"*, *k*, *t*, *A*) = *T*(*Q"*, *k*, *t*, *A*). Since the optimal order in which the processes allocated to the same processor (i.e. the optimal local schedule) may not be the same for *P'* and *P"* we know that $T(P', k, t, A) \leq T(Q', k, t, A)$, since *Q'* by definition is created with the optimal local scheduling of *P"*.

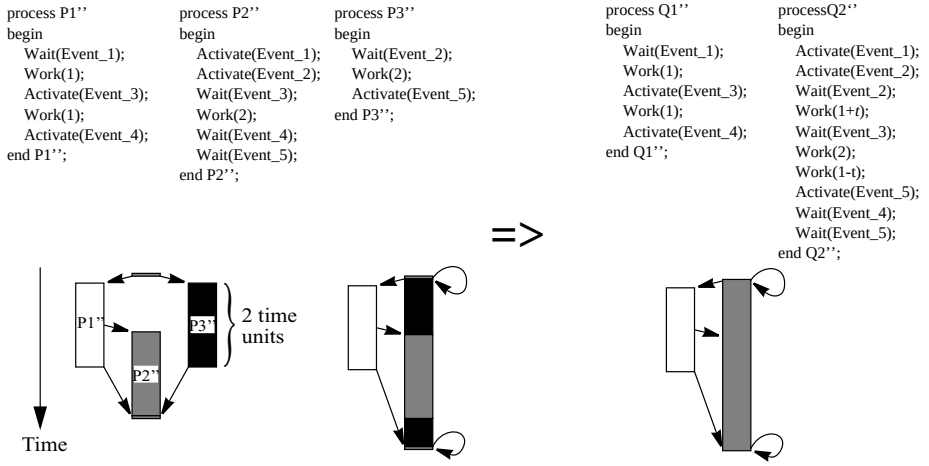


Figure 6: Transforming a program *P"*, allocated to two processors, into a program *Q"* with one process per processor.

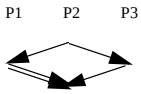
Consider now a program *R*, such that the number of processes in *R* is equal to the number of processes in *P*, and such that *R* also has zero work-time (i.e. *R* contains only synchronizations, just like *P*). The number of synchronizations between processes R*i* and R*j* in program *R* is twice the number of synchronizations between P*i* and P*j* in program *P*, i.e. there is always an even number of synchronizations between any pair of processes in *R*. All synchronizations in *R* must be executed in sequence (see Figure 7). We know that it is always possible to form such a sequence, since there is an even number of synchronizations between any pair of processes.

Sequential execution of synchronizations obviously represents the worst case, and local scheduling does not affect the execution time of a sequential program. We thus know that $2T(P, k, t, A) \leq T(R, k, t, A)$ and $2T(Q, k, t, A) \leq T(R, k, t, A)$.

Consequently,

$4T(P', k, t, A) \leq 4T(Q', k, t, A) \leq 2T(Q'', k, t, A) + 2T(Q, k, t, A) \leq 2T(P'', k, t, A) + T(R, k, t, A)$.

```
process P1
begin
   Wait(Event_1);
   Activate(Event_3);
   Activate(Event_4);
end P1;

process P2
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Wait(Event_4);
   Wait(Event_5);
end P2;

process P3
begin
   Wait(Event_2);
   Activate(Event_5);
end P3;

process R1
begin
   Wait(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Activate(Event_4);
   Wait(Event_5);
   Activate(Event_6);
end P1;

process R2
begin
   Activate(Event_1);
   Wait(Event_2);
   Activate(Event_3);
   Wait(Event_4);
   Activate(Event_5);
   Wait(Event_6);
   Activate(Event_7);
   Wait(Event_8);
   Activate(Event_9);
   Wait(Event_0);
end P2;

process R3
begin
   Wait(Event_7);
   Activate(Event_8);
   Wait(Event_9);
   Activate(Event_0);
end P3;
```



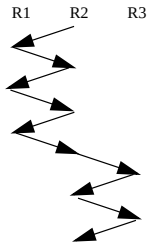


Figure 7: Transforming program *P* into program *R*.

3.3. Transforming program *P* into a program with a thick and a thin section

In this section we describe how to transform an arbitrary program into a program with one part consisting of synchronization only and the other part with all the execution time. We start with an arbitrary program *P'*. First we create *m* copies of *P'*, where $m = 2^x$, for some integer x ($x \geq 2$). We then combine the *m* copies in groups of four and transform the four copies of *P'* to two programs *P"* and one program *R*. From the discussion above we know that $4T(P', k, t, A) \leq 2T(P'', k, t, A) + T(R, k, t, A)$ for any allocation *A*, i.e. $4T(P', k, t) \leq 2T(P'', k, t) + T(R, k, t)$. Note that $4T(P', n, 0) = 2T(P'', n, 0) + T(R, n, 0)$, and that the parameters *n*, *V* and *z* are invariant in this transformation. We now end up with 2^{x-1} programs *P"*. Again we combine these 2^{x-1} *P"* programs in groups of four and use the same technique and end up with 2^{x-2} programs *P"'* (with twice the execution time compared to *P"*) and one program *R*. We repeat this technique until there are $2^{x-1} - 1$ thin *R* programs and two very thick programs (i.e. all the execution time is concentrated to two copies of the original program). By selecting a large enough m, we can neglect the synchronization times in these two copies. This is illustrated in Figure 8, where we demonstrate it for $m = 4$. Note that the execution time using k processors may increase due to this transformation, whereas $T(P, n, 0)$ is unaffected.

Thus, we are able to transform a program into two parts: a thin part consisting only of synchronizations, and the other part consisting of all the execution time. We will then discuss the thick and thin parts separately in Section 4 and Section 5, respectively.

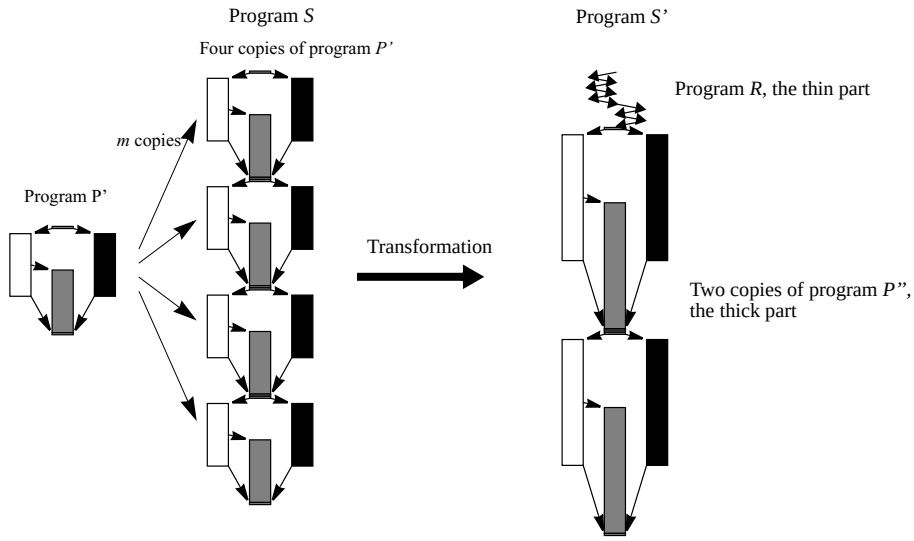


Figure 8: The transformation of *m* copies of the program P into a thick and a thin part. The transformation guarantees that $T(P', k, t)/T(P', n, 0) = T(S, k, t)/T(S, n, 0) \leq T(S', k, t)/T(S', n, 0)$.

4. The thick part

In this section we will deal with the thick part of the program. The thick part has the nice property that we can assume that the synchronization time (t) is arbitrarily small, i.e. we can assume $t = 0$. We can do this since selecting a large enough m in Section 3 will cause the synchronization time in the thick part to be negligible because virtually all synchronizations will be in the thin part. We are thus also free to introduce a constant number of new synchronizations without any problems.

These properties of the thick part will allow us to first transform the thick part into a matrix consisting of zeros and ones. Using this matrix representation we can then find the worst possible program of all programs with n processes and a parallel profile V. Finally, we will show a formula that we can use when calculating the execution time of this worst case program given an allocation. More elaborate proof and discussion concerning the transformations in this section can be found in [11] and [12].

4.1. Transforming *P* into *Q*

We consider $T(P, n, 0)$. This execution is partitioned into m equally sized time slots in such a way that process synchronizations always occur at the end of a time slot. In order to obtain Q we add new synchronizations at the end of each time slot. These synchronizations guarantee that no processing done in slot $r, (1 < r \leq m)$, can be done unless all processing in slot $r - 1$, has been completed. The net effect of this is that a barrier synchronization is introduced at the end of each time slot. Figure 9 shows how a program P is transformed into a new program Q by this technique.

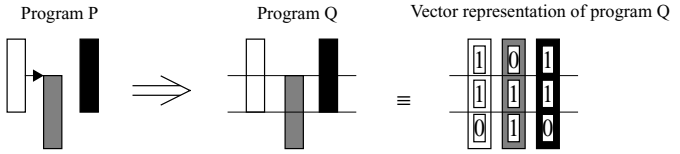


Figure 9: The transformation of program P

The synchronizations in Q form a superset of the synchronizations in P, i.e. $T(P, k, 0) \le T(Q, k, 0)$ (in the thick part of the program we assume that $t = 0$). Consequently, $T(P, k, 0)/T(P, n, 0) \le T(Q, k, 0)/T(Q, n, 0)$. It is important to note that we obtain the same parallel profile vector for both P and Q.

In order to simplify the discussion we introduce an equivalent representation of Q, called the vector representation (see Figure 9). In this representation, each process is represented as a binary vector of length m, where m is the number of time slots in Q, i.e. a parallel program is represented as n binary vectors. In some situations we treat these vectors as one binary $m \times n$ matrix, where each column corresponds to a vector and each row to a time slot.

From now on we assume unit time slot length, i.e. $T(Q, n, 0) = m$. However, $T(Q, k, 0) \ge m$, because if the number of active processes exceeds the number of processors on some processor during some time slot, the execution of that time slot will take more than one time unit.

With P_{nv} we denote the set of all programs with n processes and a parallel profile V. From the definition of the parallel profile vector, V, we know that if $Q \in P_{nv}$, then $T(P, n, 0)$ must be a multiple of a certain minimal value m_v, i.e. $V = \left(\frac{x_1}{m_v}, \frac{x_2}{m_v}, \ldots, \frac{x_n}{m_v}\right)$ where $T(P, 1, 0) = xm_v$, for some positive integer x.

Programs in P_{nv} for which $T(Q, n, 0) = m_v$ are referred to as minimal programs. For instance, the program in Figure 1 is a minimal program for the parallel profile vector $V = (1/3, 1/3, 1/3)$.

We are now able to handle programs as a binary matrix. Each column in the matrix represent one process, and the rows are independent of each others.

4.2. Transforming *Q* into *Q'*

We start by creating $n!$ copies of Q. The vectors in each copy are reordered in such a way that each copy corresponds to one of the $n!$ possible permutations of the n vectors. Vector number v $(1 \le v \le n)$ in copy number c $(1 \le c \le n!)$ is concatenated with vector number v in copy $c + 1$, thus forming a new program Q' with n vectors of length $n!m$. The execution time from slot $1 + (c - 1)m$ to cm $(1 \le c \le n!)$ cannot be less than $T(Q, k, 0)$, using k processors. Consequently, $T(Q', k, 0)$ cannot be less than $n!T(Q, k, 0)$. Since, $T(Q', n, 0) = n!m$, we know that $T(Q, k, 0)/T(Q, n, 0) = T(Q, k, 0)/m = n!T(Q, k, 0)/(n!m) \le T(Q', k, 0)/T(Q', n, 0)$. It is important to note that we obtain the same parallel profile for both Q' and Q.

The n vectors in Q' can be considered as columns in a $n!m \times n$ matrix. Reordering the rows in this matrix affects neither $T(Q', k, 0)$ nor $T(Q', n, 0)$. The rows in Q' can be reordered into n groups, where all rows in the same group contain the same number of ones (some groups may be empty). Figure 10 shows how program Q is transformed into a new program Q'.

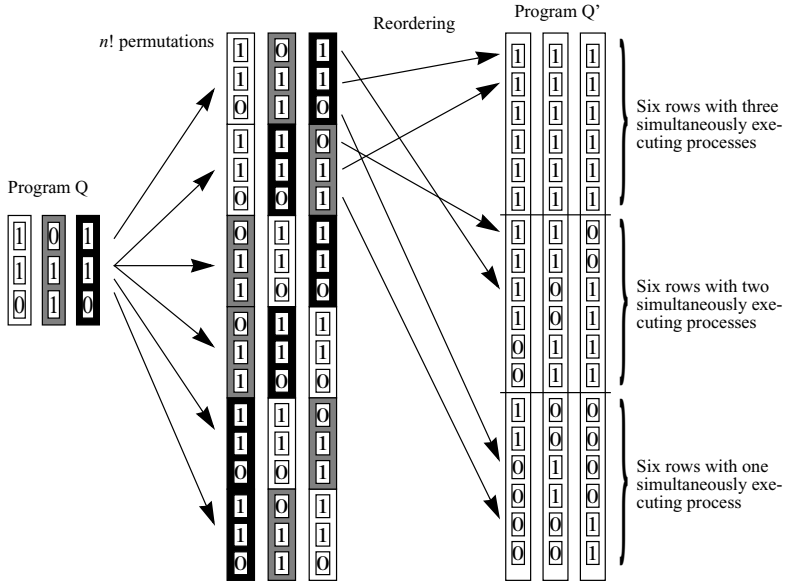


Figure 10: The transformation of vector representation of program Q into Q'.

Due to the definition of minimal programs, we know that all minimal program Q result in the same program Q_m'. In Figure 10 we have a situation where $Q_m' = Q'$. If $Q2$ $(Q2 \in P_{nv})$ is a program for which $T(Q2, n, 0) = xm_v$, $(x > 1)$, then the corresponding program $Q2'$ is identical with Q_m', except that each row in Q_m' is duplicated x times. Creating x identical copies of each row does not affect the ration $T(Q', k, 0) / T(Q', n, 0)$. Consequently, all programs Q' in P_{nv} can be mapped onto the same Q_m'. We have now found the worst possible program Q_m' ($Q_m' \in P_{nv}$).

4.3. Allocation properties of the thick part

The completion time of Q_m' is not affected by the identity of the processes allocated to different processors, because if vector p_1 is allocated to processor A and vector p_2 is allocated to processor B, then moving p_1 to B and p_2 to A is equivalent to reordering the rows in Q_m'. Obviously, reordering the rows does not affect the completion time. Consequently, the completion time of Q_m' is affected only by the number of vectors allocated to the different processors.

Theorem 3: If there are n_1 vectors allocated to processor A and n_2 vectors to processor B and $n_1 < n_2$, the completion time cannot increase if we move one vector from processor B to processor A.

Proof: We order the vectors in Q_m' in such a way the vectors 1 to n_1 are allocated to processor A and vectors $n_1 + 1$ to $n_1 + n_2$ are allocated to processor B, then we prove that moving vector $n_1 + 1$ to processor A does not increase the completion time.

If vector $n_1 + 1$ has a zero in slot r $(1 \le r \le n!m_v)$, the contribution to the completion time from row r will not be affected by moving vector $n_1 + 1$ from processor B to processor A. Consequently, we have only to consider rows for which the corresponding slot is one in vector $n_1 + 1$. Moreover, if the number of ones in positions 1 to n_1 is smaller than the number of ones in positions $n_1 + 1$ to $n_1 + n_2$, the contribution to the completion time from row r does not increase.

Q_m' contains all permutations. Consequently, for each row r such that position $n_1 + 1$ contains a one, and there are at least as many ones in positions 1 to n_1 as in positions $n_1 + 1$ to $n_1 + n_2$, there exists an (n_1, n_2)-permutation r'. An (n_1, n_2)-permutation of a row is obtained by switching items i and $n_1 + n_2 + 1 - i$ $(1 \le i \le n_1)$, see Figure 11.

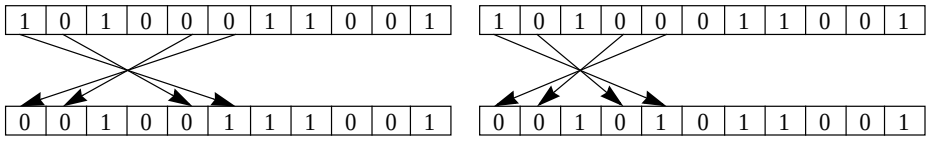


Figure 11: One (2,4)-permutation and one (2,3)-permutation of the same row.

If the contribution to the completion time from row r increases from h to $h + 1$ when we move vector $n_1 + 1$ from processor B to processor A, there must be h ones in position 1 to n_1 in row r. In that case we know that the symmetry of the (n_1, n_2)-permutation guarantees that there are at least $h + 1$ ones in positions $n_1 + 1$ to $n_1 + n_2$ in row r'. Therefore, the contribution to the completion time from r' will decrease with one when moving vector $n_1 + 1$ from processor B to processor A. Consequently, the completion time of Q_m' cannot increase if we move one vector from processor B to processor A.

■

As a consequence, an allocation of Q_m' results in shorter completion time the more evenly the processes are spread out on the processors. Also the identity of the processes is of no importance, and we can thus order the n vectors in some arbitrary order. In fact, the minimal completion time is obtained by allocating vector number $c + ik$ to processor c $(1 \le c \le k, 0 \le i \le \lfloor n/k \rfloor)$.

4.4. Calculating the thick section

The most obvious way of calculating the thick part is to generate the $n!m_v \times n$ matrix containing all permutations. This is however extremely inefficient. We will in this section show how to calculate the thick part in an analytical and generic way for any allocation of processes to processors.

The actual allocation is specified using an ordered set. The notation used for an ordered set A is that the number of elements in the ordered set is a, thus $|A| = a$. Also, the *i:th* element of the ordered set is denoted a_i, thus $A = (a_1, \ldots, a_a)$.

First we will show how we handle the case when the number of processes executing simultaneously changes during run-time. Then we will look at how to calculate the execution time of the case with a fixed number of processes executing simultaneously. In order to perform those calculation we use another function. At the end of this section we will demonstrate the most fundamental parts of the functions below using a small example.

4.4.1 *s(A, V)*

This function is used to handle the case when the number of processes executing simultaneously changes during run-time, indicated by the parallel profile V. This is illustrated by Q' in Figure 10. In this figure we have a parallel profile $V = (6/18, 6/18, 6/18) = (1/3, 1/3, 1/3)$. For each entry in this vector we count the ones (using function f described in Section 4.4.2) and use the parallel profile (V) as weight when adding them together. As found in the previous section the allocation A is independent of the process identity, thus the allocation A only shows the *number* of processes allocated to each processor, where A is decreasing, i.e. $a_i \geq a_{i+1}$ for $1 \leq i < a$. Note that we have the parameters n and k implicit in the vectors, $n = v = \sum_{i=1}^{a} a_i$ and $k = a$.

The binary matrix in Figure 10 can be divided into n parts such that the number of ones in each row is the same for each part. The relative proportion between the different parts is determined by the parallel profile vector. The function *s(A,V)* is then obtained as the weighted sum of these parts, i.e., $s(A, V) = \sum_{q=1}^{n} \frac{v_q f(A, q, 0)}{\binom{n}{q}}$.

4.4.2 *f(A, q, l)*

This function calculates how long it takes to execute a program with n processes and $\binom{n}{q}$ rows where q processes, and only q, are executing simultaneously using an allocation A, compared to executing the program using one processor for each process. This is done by counting the maximum number of ones for each processor for all rows. The function f is recursively defined and divides the allocation A into smaller sets, where each set is handled during one (recursive) call of the function. The function uses another function later found in Section 4.4.3. The function f will be discussed in detail in Section 4.4.4 using an example.

$f(A, q, l)$ uses q for the number of ones in a row. The l denotes the maximum number of ones in any previous set in the recursion. Note that we have the parameters n and k implicit in the vector, $n = \sum_{i=1}^{a} a_i$ and $k = a$. From the start $l = 0$. We also have $w = a_1$, $d = |\{a_x, \ldots\}|$ $(1 \leq x \leq a$ and $a_x = a_1)$ and $b = n - wd$.

If $b = 0$ then $f(A, q, l) = \sum_{l_1 = max(0, \lceil q/d \rceil)}^{min(q, w)} \pi(d, w, q, l_1) max(l_1, l)$, otherwise:

$$f(A, q, l) = \sum_{l_1 = max(0, \left\lceil \frac{q-b}{d} \right\rceil)}^{min(q, w)} \sum_{i = max(l_1, q-b)}^{min(l_1 d, q)} \pi(d, w, i, l_1) f(A - \{a_1, \ldots, a_d\}, q - i, max(l_1, l)).$$

4.4.3 $\pi(k, w, q, l)$

$\pi(k, w, q, l)$ [11] is a help function to f and denotes the number of permutations of q ones in kw slots, which are divided into k sets with w slots in each, such that the set with maximum number of ones has exactly l ones. $\pi(k, w, q, l) = 0$ if $q < l$ or if $q > kl$, otherwise if $k = 1$ then $\pi(k, w, q, l) = \binom{w}{l}$, otherwise it is given by:

$$\pi(k, w, q, l) = \binom{w}{l} \sum_I \binom{w}{i_1} \cdot \ldots \cdot \binom{w}{i_{k-1}} \frac{k!}{b(\{l, i_1, \ldots, i_{k-1}\})} \cdot \prod_{j=1} a(\{l, i_1, \ldots, i_{k-1}\}, j)!$$

Here, the sum is taken over all sequences of non negative integers $I = \{i_1, \ldots, i_{k-1}\}$, which are decreasing, i.e. $i_j > i_{j+1}$ for all $j = 1, \ldots, k-2$, and for which $i_1 \le min(w, l)$ and $\sum_{j=1}^{k-1} i_j = q - l$. The functions $a(I, j)$ and $b(I)$ are defined in the following way:

$a(I, j)$ = the number of occurrences of the *j:th* distinct integer in I.
$b(I)$ = the number of distinct integers in I.

More detailed proofs and discussions for how the functions in this subsection (Section 4.4.3) are obtained can be found in [11] and [12].

4.4.4 A guide through the most fundamental formula in the thick part

In this section we will focus on the function f, since function s is quite simple and the function π is already described in [11] and [12]. The example we will use have the following parameters $A = (2, 2, 1)$ and $q = 3$. This means that we have five processes of which three, and only three, are running simultaneously. We also have three processors, two which are assigned two processes each, and one processor with one process assigned to it. All possible permutations of this system is found in Table 4, where a one indicates that the process is running and a zero that the process is blocked, thus we have the vector representation found in Figure 9. What is more included in the table is the execution time required to execute each row (the left-most column). This column is calculated by finding the processor with the largest number of ones. For example, in row 1 processor A has to execute both process 1 and 2, which means that it will spend 2 time units of execution. Processor B has only process 3 to execute (process 4 has a zero), thus it will spend 1 time unit. Thus the time for executing this row is two time units and is determined by processor A. Another case is row 5, where all three processors only have a single one each, thus the time for

executing that row is one time unit. The total time for executing the program is given by adding the execution time per each row, resulting in 16 time units.

Table 4: The permutations of the system $A = (2, 2, 1)$ and $q = 3$.

Row number	Processor A		Processor B		Processor C	Execution time per row
	Process 1	Process 2	Process 3	Process 4	Process 5	
1	1	1	1	0	0	2
2	1	1	0	1	0	2
3	1	1	0	0	1	2
4	1	0	1	1	0	2
5	1	0	1	0	1	1
6	1	0	0	1	1	1
7	0	1	1	1	0	2
8	0	1	1	0	1	1
9	0	1	0	1	1	1
10	0	0	1	1	1	2

We will now briefly describe the basic structure of function f. The allocation ($A = (2, 2, 1)$) is split into sets of processors with equal number of processes assigned to them. Remember that A is decreasing. In our example we have the sets of $(2, 2)$ and (1). Each such set fits into the function π and will be handled separately through the recursion. The variable d is the length of such a set and w is the number of processes in each element in the set, thus dw is the number of processes in the set. The variable b is the number of processes in the remaining part of the allocation, thus representing the maximum number of ones that possibly can be fit into the remaining part of the allocation. The variable l_1 iterates over all possible number of ones that can fit in each processor. Also the variable i iterates over all possible number of ones that can be contained in the whole set. The variable l holds the maximum number of l_1 through the recursion and all the sets. The variable l is set to zero for the first call to function f for the only reason that it will not be larger than any of the l_1 in the sets.

In the following sections we will go through the simple example starting in Section 4.4.5 with the first call to function f with the initial parameters.

4.4.5 *f*(*A*=(2, 2, 1), *q*=3, *l*=0)

From the incoming parameters we can conclude that $n = 5$, $w = 2$, $d = 2$, and $b = 1$. Since $b \neq 0$ the second part of the function is used. Further the variable $l_1 = 1 \ldots 2$. For each iteration over l_1 we have:

- $l_1 = 1$: The variable $i = 2 \ldots 2 = 2$, thus we have the call to function $\pi(2, 2, 2, 1) = 4$ and the (recursive) call to $f((1), 1, 1) = 1$ (see Section 4.4.6 for how this is calculated). The resulting value is then $4 \cdot 1 = 4$.

- $l_1 = 2$: The variable $i = 2 \ldots 3$, for each iteration over i we have:
  - $i = 2$: The call to function $\pi(2, 2, 2, 2) = 2$ and the (recursive) call to $f((1), 1, 2) = 2$ (see Section 4.4.7 for how this is calculated). The resulting value of this iteration is then $2 \cdot 2 = 4$.
  - $i = 3$: The call to function $\pi(2, 2, 3, 2) = 4$ and the (recursive) call to $f((1), 1, 2) = 2$ (see Section 4.4.7 for how this is calculated). The resulting value of this iteration is then $4 \cdot 2 = 8$.

In total $f((2, 2, 1), 3, 0) = 4 + 4 + 8 = 16$. This is what we concluded by hand in Table 4.

4.4.6 *f*(*A*=(1), *q*=1, *l*=1)

From the incoming parameters we can conclude that $n = 1$, $w = 1$, $d = 1$, and $b = 0$. Since $b = 0$ the first part of the function is used. Further the variable $l_1 = 1 \ldots 1$. For $l_1 = 1$ we have the call to function $\pi(1, 1, 1, 1) = 1$ and $max(l_1, l) = max(1, 1) = 1$. The resulting value of this iteration is then $1 \cdot 1 = 1$. In total $f((1), 1, 1) = 1$.

4.4.7 *f*(*A*=(1), *q*=1, *l*=2)

From the incoming parameters we can conclude that $n = 1$, $w = 1$, $d = 1$, and $b = 0$. Since $b = 0$ the first part of the function is used. Further the variable $l_1 = 1 \ldots 1$. For $l_1 = 1$ we have the call to function $\pi(1, 1, 1, 1) = 1$ and $max(l_1, l) = max(1, 2) = 2$. The resulting value of this iteration is then $1 \cdot 2 = 2$. In total $f((1), 1, 2) = 2$.

5. The thin part

From Section 3 we know that the thin part of a program for which the ratio $T(P, k, t)/T(P, n, 0)$ is maximized consists of a sequence of synchronizations, i.e. program *R* in Section 3. From Section 4 we know that the identity of the processes allocated to a certain processor does not affect the execution time of the thick part. In the thin part we would like to allocate processes that communicate frequently to the same processor.

Let y_i be a vector of length *i*-1. Entry *j* in this vector indicates the number of synchronizations between processes *i* and *j*. Consider an allocation *A* such that the number of processes allocated to processor *x* is a_x. The optimal way of binding processes to processors under these condition is a binding that maximizes the number of synchronizations within the same processor.

Consider also a copy of the thin part of the program where we have swapped the communication frequency such that process *j* now has the same communication frequency as process *i* had previously and process *i* has the same communication frequency as process *j* had previously. In Figure 12 the original communication vector for P3 is (6,2), meaning that there are six synchronizations between P3 and P1 and two synchronizations between P2 and P3. In the copy we have swapped the communication frequencies of P2 and P3 and we thus have six synchronizations between P2 and P1 and two synchronizations between P2 and P3.

It is clear that the minimum execution time of the copy is the same as the minimum execution time of the original version. The mapping of processes to processors that achieves this execution time may however, not be the same. For instance, if we have two processors and an allocation *A* = (2,1), we obtain minimum completion time for the original version when P1 and P3 are allocated to the same processor, whereas the minimum for the copy is obtained when P1 and P2 share the same processor.

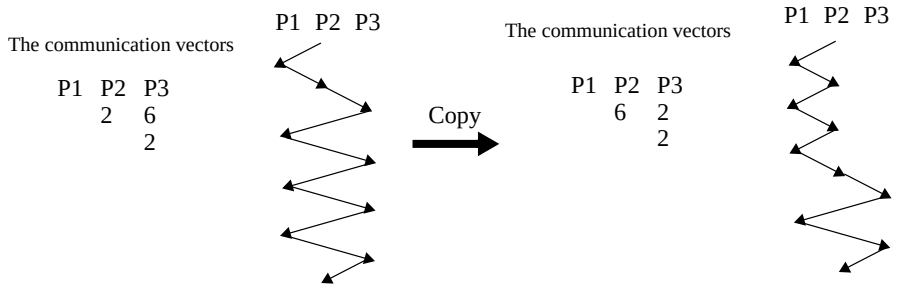


Figure 12: Making a copy where we have swapped the communication frequencies of processes P2 and P3.

If we concatenate the original (which we can call *P*) and the copy (which we can call *Q*) we get a new program *P'* such that $T(P', k, t, A) \geq T(P, k, t, A) + T(Q, k, t, A)$ for any allocation A (all thin programs take zero execution time using a system with one process per processor and no communication delay). By generalizing this argument we obtain the kind of permutation as we had for the thick part (see Figure 10).

These transformations show that the worst case for the thin part occurs when all synchronization signals are sent from all n processes $n-1$ times - each time to a different process. All possible synchronization signals for n processes equals $n(n-1)$ and all possible synchronization signals between the processes allocated to processor i equals $a_i(a_i-1)$. Because some processes are executed on the same processor, the communication cost for them equals zero. That means that the number of synchronization signals in the worst-case of program (regarding communication cost) is equal to $n(n-1) - \sum_{i=1}^{k} a_i(a_i-1)$.

Note that we have the parameters n and k implicit in the vectors, $n = v = \sum_{i=1}^{a} a_i$ and $k = a$.

If $n = 1$ then $r(A, t, V) = 0$, otherwise $r(A, t, V) = \dfrac{n(n-1) - \sum_{i=1}^{k} a_i(a_i-1)}{n(n-1)} t \sum_{q=1}^{n} (qv_q)$.

6. Combining the thick and thin sections

Combining the thick and the thin section is done by adding the function $s(A, V)$ and $r(A, t, V)$, weighted by the granularity, z. Thus, we end up with a function $p(n, k, t, z, V) = min_A(s(A, V) + zr(A, t, V))$. It should be noted that the allocation A implicitly includes the parameters $n = \sum_{i=1}^{a} a_i$ and $k = a = |A|$. As we can see in the formula we use the granularity, z, as a weight between the thick part and the thin part. In a program with high granularity, i.e. high synchronization frequency, the thin part has larger impact than in a program with low granularity.

As previously shown the thick section is optimal when evenly distributed over the processors, whereas the thin section is optimal when all processes reside on the same processor. The algorithm for finding the minimum allocation A is based on the knowledge about the optimal allocation for the thick and thin part respectively.

6.1. Finding the optimal allocation using allocation classes

The basic idea is to create *classes* of allocations and evaluate them. An allocation class consists of three parts:

- A common allocation for both the thick and thin parts, the allocation here shows the number of assigned processes for the n first processors. The allocation must be decreasing, i.e. the number of assigned processes to processor i must be greater than or equal to the number of assigned processes to processor $i + 1$.
- The allocations for the remaining processors for the thick part, the allocations are evenly distributed. The highest number of processes assigned to a processor must be equal or less than the least number of assigned processes for any processor in the common assignment.
- The allocations for the remaining processors for the thin part, the allocations make use of as few of the remaining processors as possible and with as many processes as possible on each processor. The highest number of processes assigned to a processor must be equal or less than the least number of assigned processes for any processor in the common assignment.

An example of an allocation class with the common part consisting of one processor, $n = 9$ and $k = 4$, is shown in Figure 13(a) where the first part is the common allocation, the upper part is the thick allocation for the remaining 3 processors and the lower part is the thin allocation for the remaining 3 processors. In Figure 13(b) we have the first allocation, with no common allocation at all. In fact this first allocation consider the thick part only and, thus is quite inaccurate. In Figure 13(c) are all the possible allocation classes for this example. By calculating the thick part using the common and thick allocation and the thin part using the common and thin allocation we will get a result that is better than (or equal to) any allocation within that class. This is because we will get an over-optimal result, due to the fact that we have different allocations for the thick and the thin part, which in practice is impossible. Then we take the minimum value of all the classes in Figure 13(c) for the over-optimal allocation.

6.2. Branch-and-bound algorithm

The algorithm for finding an optimal allocation is a classical branch-and-bound algorithm [1]. The allocations can then be further divided, by choosing the class that gave the minimum value and add one processor in the common allocation, lets say in this example (in Figure 13) the class with 5 processors in the common allocation was the minimum. The subclasses are shown in Figure 13(d). All the subclasses will have a higher (or equal) value than the previous class. The classes are organized as a tree structure and the minimum is now calculated over the *leaves* in the tree and a value closer to the optimal is given. By repeatedly selecting the leaf with minimum value and create its subclasses we will reach a situation where the minimum leaf no longer has any subclasses, then we know that we have found the optimal allocation. If we assume that the classes in Figure 13(e) and (f) gives the minimum values we have reached an optimal allocation. When calculating a class we use the common allocation concatenated with the thick allocation as input for the function $s(A_{thick}, V)$ and $r(A_{thin}, t, V)$, where A_{thick} is the common allocation concatenated with the thick allocation and A_{thin} is the common allocation concatenated with the thin allocation. By adding the results (weighted by z) from the two functions we get the value of that leaf ($s(A_{thick}, V) + zr(A_{thin}, t, V)$).

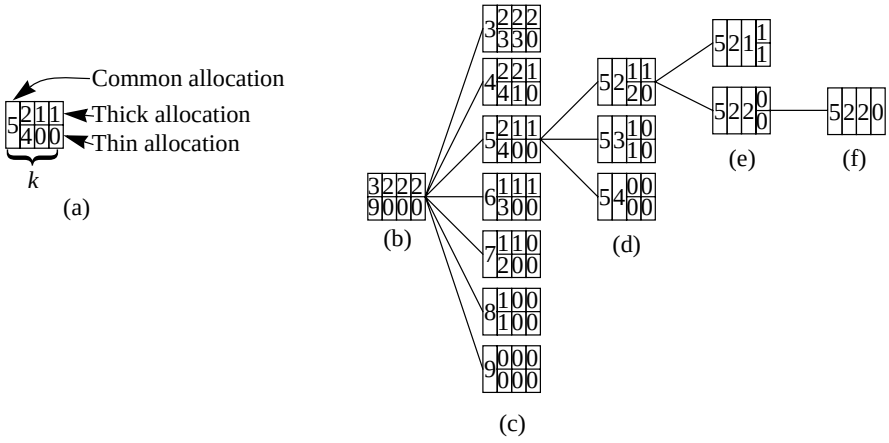


Figure 13: Allocation classes.

6.3. Lazy evaluation

We can further reduce the number of calculations by adopting a lazy evaluation approach. This approach is based on the observation that the value for the thick part (function s) in Figure 13(c) increases as we calculate it for more and more processes on the first processor. This is also shown in Figure 14 for an application with $n = 79, k = 16$. We also know that the value of the thin part is decreasing as we increase the number of processes on the first processor, also shown in Figure 14. The value of an allocation class is, as stated above, the sum of the thick part and the thin part weighted by z, which is shown in Figure 14 for some z.

The lazy evaluation is based on the fact that the thick part is increasing and the thin part is decreasing. We can conclude from Figure 14 that the minimum of the thick and thin curve is at eight processes on the first processor, thus when we reach 31 processes on the first processor we note that the thick part alone is larger than the minimum at 8 processes. We then know that this allocation (and further) can never be the minimum (since the thick part is increasing) regardless how much the thin part decreases. In this case the thin part may become zero since the allocation (9) yields a thin part of zero. We mark all allocations with more than 31 processes on the first processor to have the value of *at least* the value of the *thick part* of 31 processors on the first processor *and* the *thin part* of allocation (9) (which is zero). If we now apply the same technique in the subclasses found in Figure 13(d), we know that the least value of the thin part is found in the allocation $(5, 4)$. This means that the thin part is never lower than in this allocation class, thus we can stop evaluating when the value of the thick part of the current allocation and the thin part of allocation $(5, 4)$ (the allocation with the least thin part) reaches above the minimum. We then mark the remaining allocations with the value of at least the thin part of the current allocation and the thin part of allocation $(5, 4)$.

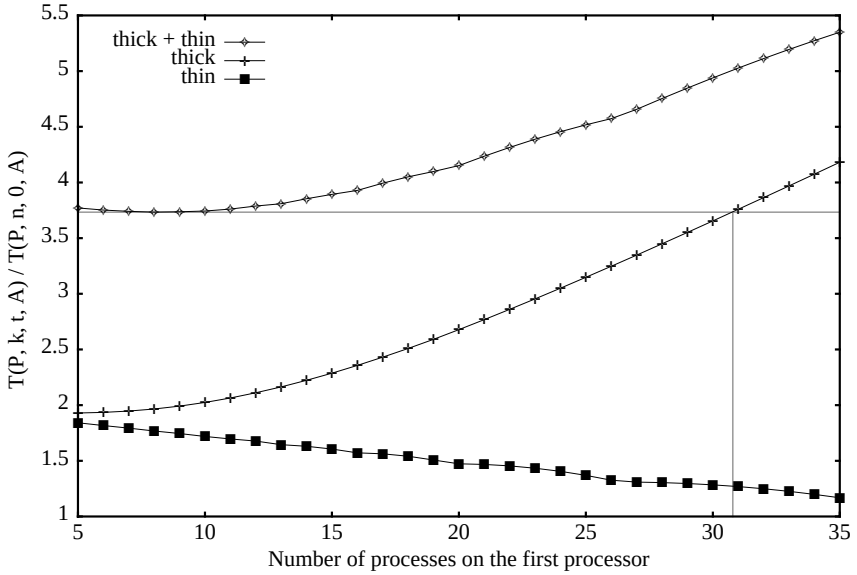


Figure 14: An example showing that the thick part increases and the thin part decreases as the number of processes on the first processor increases.

When we continue (in Figure 14) to make the allocation subclasses to the class with eight processes on the first processor the minimum may increase. At some point the minimum may increase above the value of the thick part at 31 processes on the first processor, then we have to continue to calculate the case with 32 processes and so on until we once again find a thick part that is larger than the minimum. The benefit with this lazy evaluation is that we will not always have to evaluate all allocation classes as later shown in Section 7.2.

The drawback with this algorithm is that it (theoretically) can search through all possible nodes in the tree before finding the optimal solution. However, our practical experience shows that only very few nodes needs to be investigated until the optimal leaf is found, see Section 7.2.

7. Method demonstration

In this section we will demonstrate how the method in this paper can be used in a practical case. Previously we have developed a tool for monitoring multithreaded programs on a uni-processor workstation and then, by simulation, predict the execution of the program on a multiprocessor with any number of processors. The tool is called VPPB (Visualization of Parallel Program Behaviour) [2, 3]. We have modified this tool to extract the program parameters required by our method (n, V, and z). The tool can be given information about the hardware in order to simulate synchronization latency (t) between the k (simulated) processors. We use that ability to compare the results from our method with the simulations.

7.1. Overview of the 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.

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 15. The developer writes the multi-

threaded program (a) in Figure 15, compiles it and an executable binary file is obtained. After that, the program is executed on a single processor workstation. 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.

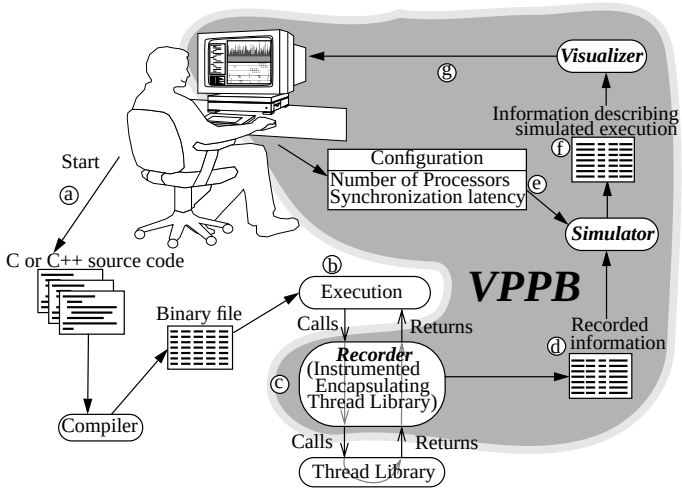


Figure 15: A schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 15. 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 can count the number of threads in the program, the parameter n. We can also count the number of synchronizations in the recorded information, as well as accumulate all execution times recorded in order to get the execution time when using one processor. The later is used together with the number of synchronizations to get the granularity, parameter z. Also the parallel profile vector, parameter V, can be obtained using the simulator. The simulator simulates the recorded information using as many processors as there are threads, then the simulator is able to calculate the parallel profile vector. The simulator was already able to simulate more loosely coupled systems, thus, to set a latency time for the synchronizations between processors [3]. This functionality can be used when comparing a real allocation (predicted by the simulator) with the values given from the function p.

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. The result from the algorithm is shown as a vertical bar, indicating that this is longest time the program has to run if appropriately allocated. With these facilities the developer may detect problems in the program and can modify the source code (a) or

change the allocation. 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 [9] and/or POSIX threads [6] on the Solaris 2.X operating system.

7.2. Example - prime number generator

We will demonstrate the described technique (including the tool described in Section 7.1) by bounding the speed-up of a parallel C-program (with Solaris threads) for generating all prime numbers smaller than or equal to a number X. The algorithm generates the primes numbers in increasing order starting with prime number 2 and ending with the largest prime number smaller than or equal to X. A number Y is a prime number if it is not divisible by any prime number smaller than Y. Such divisibility tests are carried out by filter threads; each filter thread has a prime number assigned to it. This is shown in Figure 16. In the front (the first process) there is a number generator feeding the chain with numbers. When a filter receives a number it will test it for divisibility with the filter's prime number. If the number is divisible it will be thrown away. If the number is not divisible it will be sent to the next filter. In the case that there is no next filter, a new filter is created with the number as the new filter's prime number.

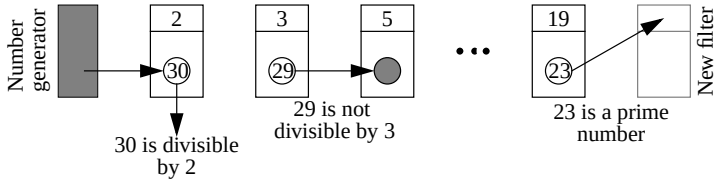


Figure 16: The algorithm for generating prime number.

The demonstration was performed with a number generator that stops by generating number 397, which is the 78th prime number. This means the program will contain 79 threads (including the number generator). The VPPB tool uses the thread programming model which means that the target environment is a (distributed) shared memory machine. The program's speed-up has been evaluated using three kinds of networks; *fast* network with no extra time for a (remote) synchronization, *ordinary* network with a synchronization time of four microseconds, and a *slow* network with eight microseconds latency. The value of the ordinary network has been defined by the remote memory latency found in three distributed shared memory machines: SGI's Origin 2000 [10], Sequent's NUMA-Q [14], and Sun's WildFire [8]. The remote memory latency varies between 1.3 to 2.5 micro seconds [21] and a synchronization must at least include two memory accesses (the thread releasing the thread issues a write and the receiving threads issues a read). The slow network is simply twice the ordinary network. The results are compared to the predictions of the tool in Section 7.1 when applying the corresponding cost for synchronization. The allocation algorithm used in the simulated case is simple. The first n/k processes (process one is the number generator, process two is the first filter, and so on) are allocated to the first processor, the next n/k processes to the second processor and so on. In the case when n is not divisible with k, the first n modulus k processors are allocated $\lceil n/k \rceil$ processes and the remaining processors are allocated $\lfloor n/k \rfloor$ processors. This is a common way to allocate chain-structured programs [4].

As can be seen in Figure 17 (zero latency) the simple allocation scheme performs worse than function p, and we can thus conclude that we for sure are able to find a better allocation. In Figure 18 (latency is four micro seconds) the simple allocation scheme performs worse than function p for less than eight processors, and we can thus conclude that we for sure are able to find a better allocation when we have less than eight processors. In the case with eight processors or more we can not conclude if there is a better allocation or not. However, it is important to know

that the fact that we are above the bound does not guarantee that the simple allocation is the optimal. In Figure 19 (latency is eight micro seconds) the simple allocation scheme performs worse than function p for four (or less) processors, and we can thus conclude that we for sure are able to find a better allocation when we have less than five processors. In the case with five processors or more we can not conclude if there is a better allocation or not.

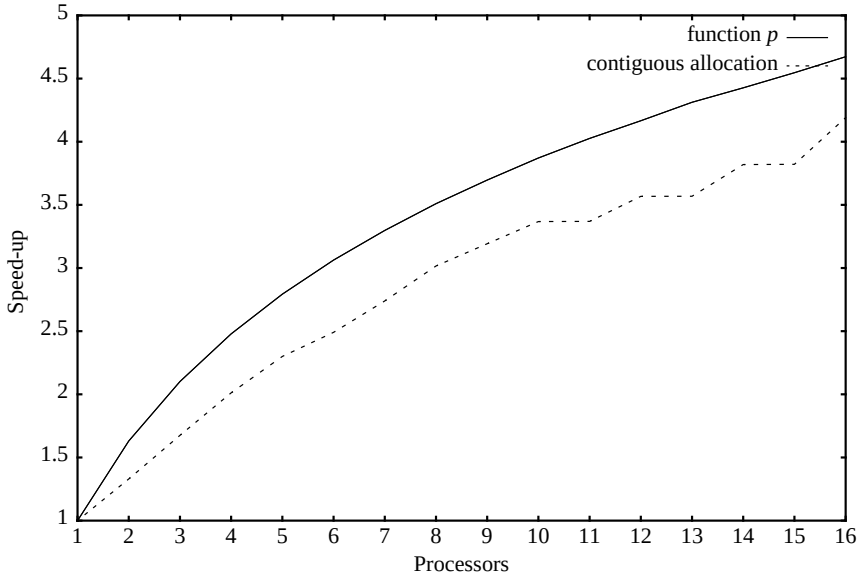


Figure 17: The speed-up for the prime number program on a fast network (latency = 0 μs).

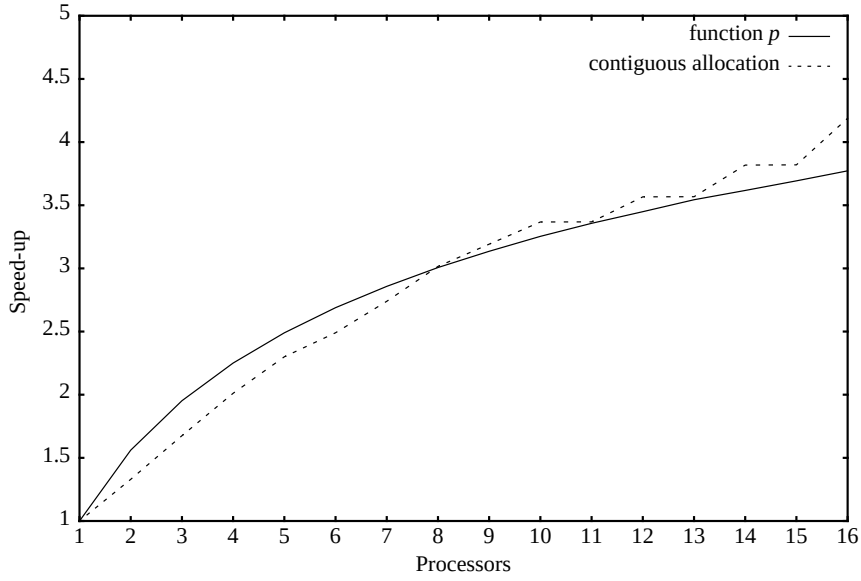


Figure 18: The speed-up for the prime number program on an ordinary network (latency = 4 μs).

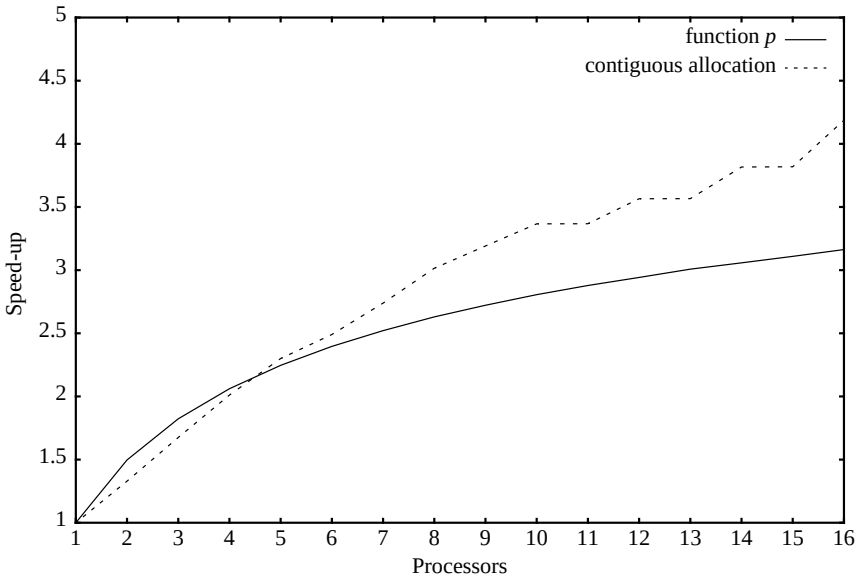


Figure 19: The speed-up for the prime number program on a slow network (latency = 8 μ s).

If we keep the number of processors constant, say at eight, we see how the functions and allocations are affected by the network latency time, shown in Figure 20. The speed-up will decrease as the latency increases. This is clearly shown for function *p* . The simple allocation is latency tolerant and the speed-up decreases, although hard to see. This is due to allocating the first processes (which also communicated the most) to the same processor, and so on.

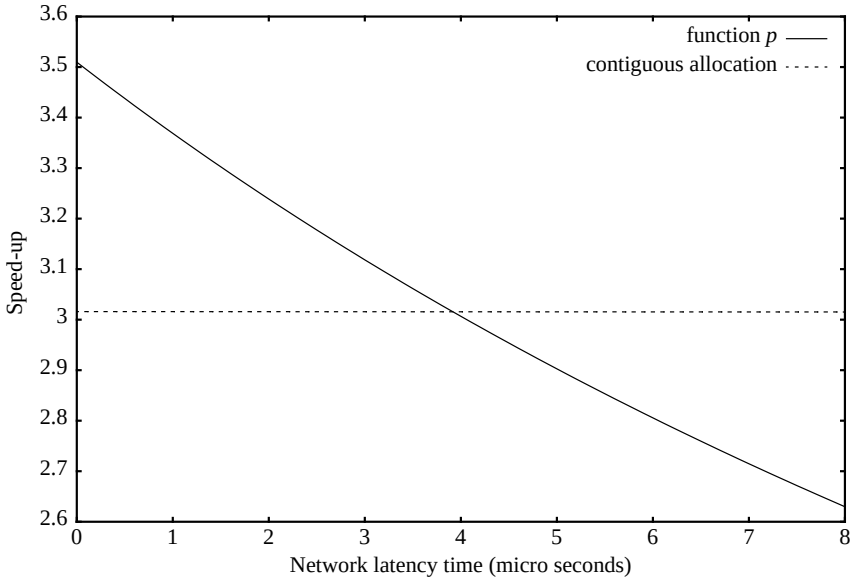


Figure 20: The speed-up for the prime number program on eight processors with various networks latencies.

As indicated earlier, the function *p* gives a bound. The intended usage of this bound is to indicate that there is a better allocation strategy if the current allocation strategy gives results below the bound, as in the case in Figure 17 (the prime program with no latency). We now look at other allocation strategies and tries a round robin strategy that assigns (on a multiprocessor with *k* processors) thread number *i* to processor (*i* mod *k*)+1. In Figure 21 we see that the round robin strategy gives better result than the bound. However, a better result than the bound does not automatically means that we have found an optimal result.

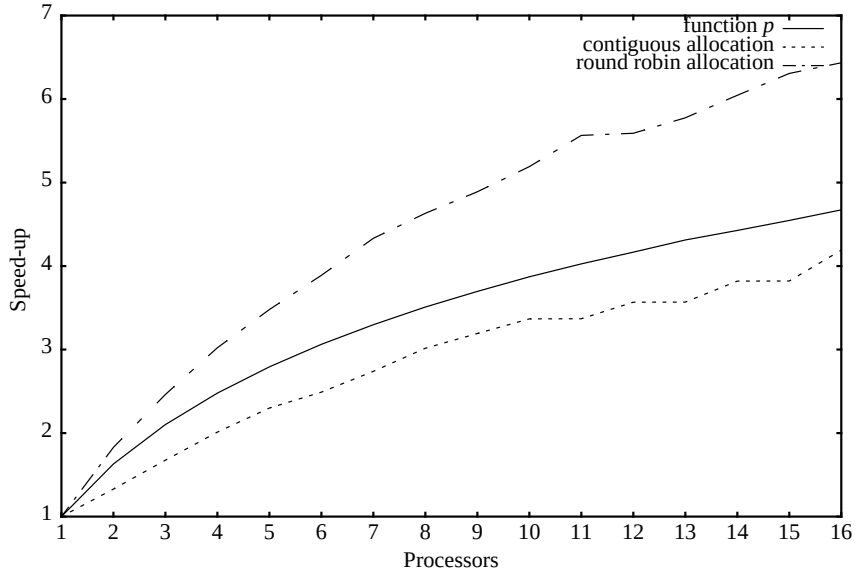


Figure 21: The speed-up using round robin for the prime number program (latency = 0 μ s).

In Figure 22 the previous graphs are summarized. The figure shows the function *p* for a latency between 0 and 8 micro seconds and the number of processors between 1 and 16.

It took in average 102 seconds to calculate a data point in the graph in Figure 22 on a 300 MHz Sun Ultra 10 workstation. The reason for finding the optimal solution so fast is the use of allocation classes and the branch-and-bound algorithm combined with the lazy evaluation, described in Section 6. One might think that if we desire to find the optimal solution, all the calculations with allocation classes will be waisted since they are all (except the last) unrealistic allocations (different allocations for the thin and thick part). On the contrary, by traversing the tree from the top we are able very efficiently ignore several branches. The reason for this is that an allocation class will always have worse (or equal) results in its sub-classes. We are thus able reduce the search dramatically. We are able to reduce some of the calculation costs as well. The major calculation cost is for the thick part and in Figure 13(d) we can see that the top allocation class has the same thick part allocation as the previous allocation class in Figure 13(c). Then we can re-use the already calculated value of the thick part, however the two remaining allocations in Figure 13(d) have to be calculated. Note that we always calculate the thin part, since it is very fast to calculate.

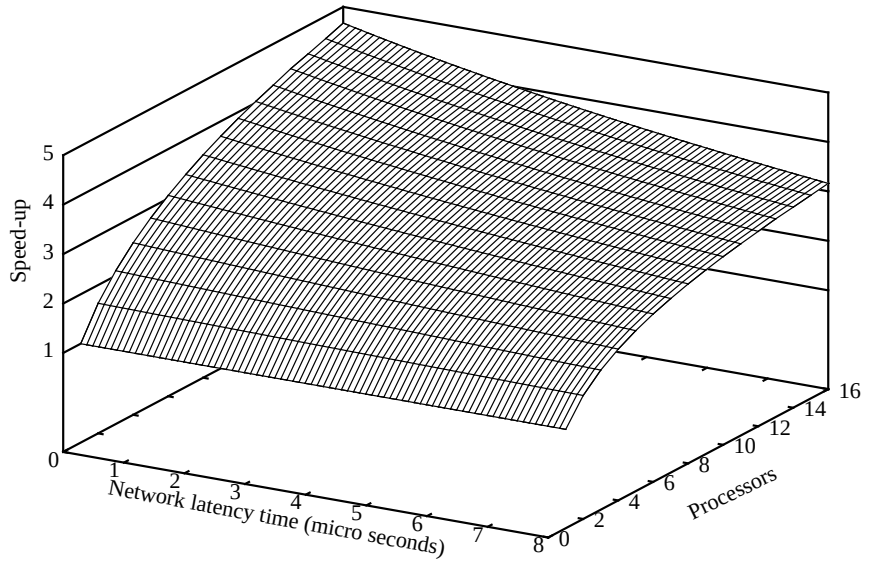


Figure 22: The speed-up for the prime number program by applying function *p*, varying number of processors and networks latencies.

In order to illustrate the benefits of our algorithm we consider the prime number program on 16 processors. In total we can find 6,158,681 unique allocations of the 79 processes on 16 processors. The number of allocations that needs to be evaluated using our approach is very small, in Figure 23 the fraction of allocations calculated using branch-and-bound compared to extensive search is shown. As can be seen, roughly only one allocation out of 80,000 needs to be evaluated even without the lazy evaluation. If we add the lazy evaluation we have to evaluate roughly one allocation out of 270,000. The time to calculate the optimal allocation for the prime number program ($k = 16, t = 8\mu s$) is five minutes on a 300MHz Ultra 10. Without the allocation classes, branch-and-bound technique and lazy evaluation the same calculation would require two and a half year.

8. Discussion and Related Work

The parameters used by the method do not require such an advanced tool as the one used in Section 7. However, in order to get the programs parameters we still need some kind of tool. In order to calculate the granularity, z, we need the programs total execution time (on one processor) which can be obtained by executing the program on a single processor workstation and some time-command to measure the time. The number of synchronizations must, however, be measured in some other way. One way could be to use the (unsupported) tool from Sun called tnfview [22] which graphically shows all threads and synchronizations. Although possible to calculate the number of synchronizations by hand a short script would do the work. The tnfview also shows the number of threads, i.e. the parameter n.

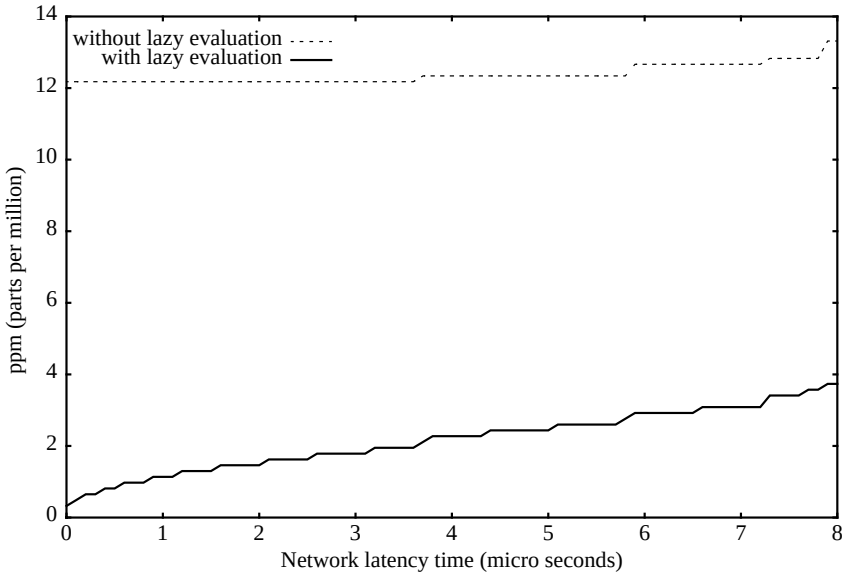


Figure 23: The fraction of allocations calculated using allocation classes and branch-and-bound with and without the lazy evaluation compared to extensive search, expressed in ppm (parts per million).

The remaining parameter, the parallel profile vector V, is somewhat harder to obtain than n and z. Here we need to find the amount of parallelism in the program when there are one processor for each process. For many systems the number of processors in a multiprocessor is less than the number of threads in the program. We can use tnfview (with some limitations as discussed and solved for our tool in [2]) in order to capture the synchronizations between threads and calculate the sequential work time between the synchronizations. With this information we can build a weighted directed graph where each node is a sequential segment (with the work time as weight) and where an edge is a synchronization. With this graph we can obtain the parallel profile vector V. The length of the longest path in the graph will represent $T(P, n, 0)$.

Further development of the method could be to adjust the formulas to include that several processors may be situated on the same node, sharing the same physical memory, thus the latency is then between processors on different nodes and not between processor on the same node. Processes may also be dynamically scheduled from one processor to another within the same node.

The basic approach in the current work is to obtain a performance bound, which we know is possible to achieve. This approach has been used for a long time in real time systems, where it is well known that a set of n tasks is schedulable if the sum of processor utilization is less than $n(2^{(1/n)} - 1)$ [5], similar results also exist for real-time process sets which operates under what is called an age constraint [19][20].

The proof techniques described in Section 4 in this paper has previously been used for obtaining performance bounds in a number of application areas besides multiprocessor scheduling, e.g. cache memory systems [13], parallel accesses to multiprocessor memory systems [16], and message scheduling on a number of communication links [15].

9. Conclusion

Cluster systems do not usually allow processes to dynamically migrate from one node to another. It is thus essential to find an efficient allocation of processes to processors. Even in the case of machines with distributed shared memory such allocations can be beneficial. Unfortunately, finding the optimal allocation is NP-hard and we have to resort to heuristic algorithms to find a good allocation. One problem with heuristic algorithms is that we do not know whether the result is close to optimal or if it is worth-while to continue the search for a better result.

In this paper we find a way to analytically calculate a bound on the minimal completion time for a given parallel program. The program is specified by the parameters n (the number of processes), V (the parallel profile vector), and z (the synchronization frequency); different programs may yield the same n, V, and z. Further, the hardware is specified by the parameters k (number of processors) and t (the synchronization latency). The bound on the minimal completion time is optimally tight. This means that there exists a program (with the given parameters n, V, and z) for which the minimal completion time is equal to the bound. The parameters z and t give a more realistic model than previous work [17][18]. The bound makes it possible for the user to determine when it is worth-while to continue the heuristic search, or perhaps apply another heuristic algorithm. Analytical performance bounds, like the one presented here, have successfully been used when engineering real-time systems, and techniques similar to the one described here has been used for network and memory systems. Our method includes an aggressive branch-and-bound algorithm that, together with lazy evaluation, has been shown to reduce the search space to only 0.0004% for finding the optimal bound.

The method has been implemented in a practical tool. The tool is able to automatically obtain the values of n, V, and z for a program and calculate the minimal completion bound with the given hardware parameters. Based on this practical tool we have demonstrated the usability of the method. One feature of the tool is that it can be totally operated on a single processor workstation. This is of practical importance since no multiprocessor is then needed when applying the method. Finally, the tool makes the presented analytical bound on the minimal completion time easily accessible for practitioners.

Acknowledgment

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] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, “Scheduling Computer and Manufacturing Processes,” Springer-Verlag, ISBN 3-540-61496-6, 1996.

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

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

[4] S. H. Bokhari, “Partitioning Problems in Parallel, Pipelined, and Distributed Computing,” IEEE transactions on Computers, Vol. 37, No. 1, 1988.

[5] A. Burns and A. Wellings, “Real-Time Systems and Programming Languages,” Addison-Wesley, 2001, ISBN 0-201-72988-1.

[6] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, ISBN 0-20-163392-2, 1997.

[7] M. Garey and D. Johnson, “Computers and Intractability,” W. H. Freeman and Company, 1979.

[8] E. Hagersten, and M. Koster, “WildFire: A Scalable Path for SMPs,” in Proceedings of the The Fifth International Symposium on High Performance Computer Architecture, pp. 172-181, 1999.

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

[10] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA Highly Scalable Server,” in Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 241-251, 1997.

[11] H. Lennerstad and L. Lundberg, “An optimal Execution Time Estimate of Static versus Dynamic Allocation in Multiprocessor Systems,” SIAM Journal of Computing, Vol 24, no 4, pp. 751-764, 1995.

[12] H. Lennerstad and L. Lundberg, “Optimal Combinatorial Functions Comparing Multiprocess Allocation Performance in Multiprocessor Systems,” SIAM Journal of Computing, Vol. 29, No. 6, pp. 1816-1838, 2000.

[13] H. Lennerstad and L. Lundberg, “Optimal Worst Case Formulas Comparing Cache Memory Associativity,” SIAM Journal of Computing, Vol. 30, No. 3, pp. 872-905, 2000.

[14] T. Lovett and R. Clapp, “STiNG: A cc-NUMA Computer System for the Commercial Marketplace,” ACM/IEEE International Symposium on Computer Architecture (ISCA), pp. 308-317, 1996.

[15] L. Lundberg, H. Lennerstad, “Optimal Bound on the Gain of Permitting Dynamical Allocation of Communication Channels in Distributed Processing,” Acta Informatica, Vol. 36, No. 6, pp. 425-446, 1999.

[16] L. Lundberg, H. Lennerstad, “Bounding the Gain of Changing the Number of Memory Modules in Shared Memory Multiprocessor,” Nordic Journal of Computing, Vol. 4, No. 3, 1997, pp. 233-258.

[17] L. Lundberg, H. Lennerstad, “Using Recorded Values for Bounding the Minimum Completion Time in Multiprocessors,” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 4, pp. 346-358, 1998.

[18] L. Lundberg, H. Lennerstad, “An Optimal Upper Bound on the Minimal Completion Time in Distributed Supercomputing,” in Proceedings of the 1994 ACM International Conference on Supercomputing, pp. 196-203, 1994.

[19] L. Lundberg, “Fixed Priority Scheduling of Age Constraint Processes,” in Proceedings of the 4th International Euro-Par Conference, pp. 288-296, 1998.

[20] L. Lundberg, “Utilization Based Schedulability Bounds for Age Constraint Process Sets in Real-Time Systems,” Real-Time Systems, to appear.

[21] L. Noordergraaf and R. van der Pas, “Performance Experiences on Sun’s WildFire Prototype,” in Proceedings of the Super Computing Conference 1999, CD-ROM, 1999.

[22] SunSoft, “tnfview Demo Guide,” 1995. The documentation is included in ftp://ftp.sunsite.kth.se/ftp/www/sun/sunsite-sun-info/tnftools/tnfdemo.tar.gz, (site visited November 20th, 2001).

Paper VIII

Selecting Simulation Models when Predicting Parallel Program Behaviour

Magnus Broberg, Lars Lundberg, and Håkan Grahn
Research Report 2002:04, 2002

VIII

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.

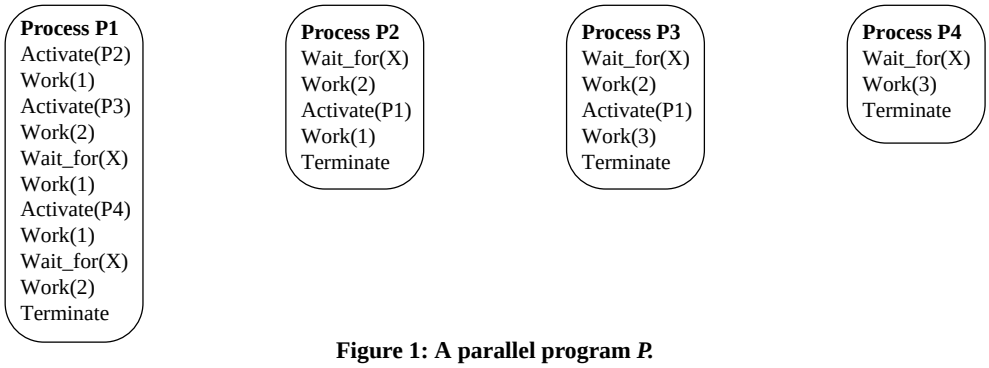


Figure 1: A parallel program *P*.

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.

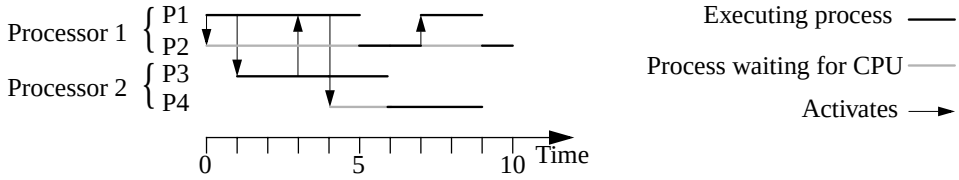


Figure 2: Execution flow of program *P* using two processors.

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.

```
Process P1
Activate(P2)
Work(1)
Activate(P3)
Work(1)
for (i = 1, i <= 2, i++) {
    Wait_for(X)
    Work(2)
    if X == P2
        Activate(P2)
    else
        Activate(P3)
    Work(1)
}
Terminate
```

```
Process P2
Wait_for(X)
Work(7)
Activate(P1)
Work(1)
Wait_for(X)
Work(1)
Terminate
```

```
Process P3
Wait_for(X)
Work(2)
Activate(P1)
Work(1)
Wait_for(X)
Work(1)
Terminate
```

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

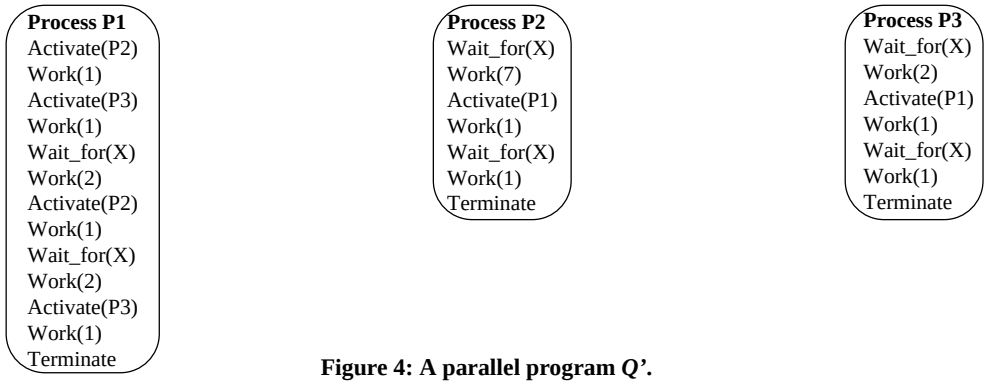


Figure 4: A parallel program *Q'*.

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.

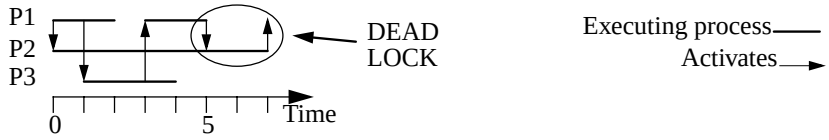


Figure 5: Execution flow of program *Q'* when using one processor per process.

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.

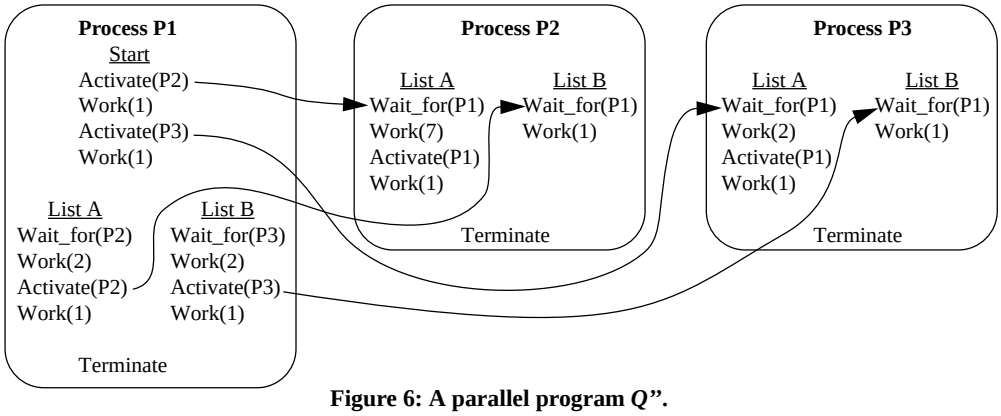


Figure 6: A parallel program *Q"*.

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.

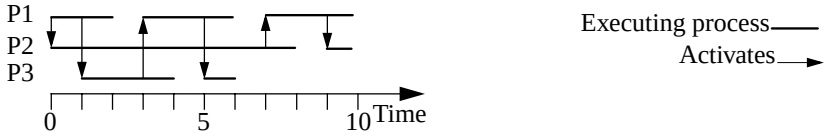


Figure 7: Execution flow of program *Q"* when using one processor per process.

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*.

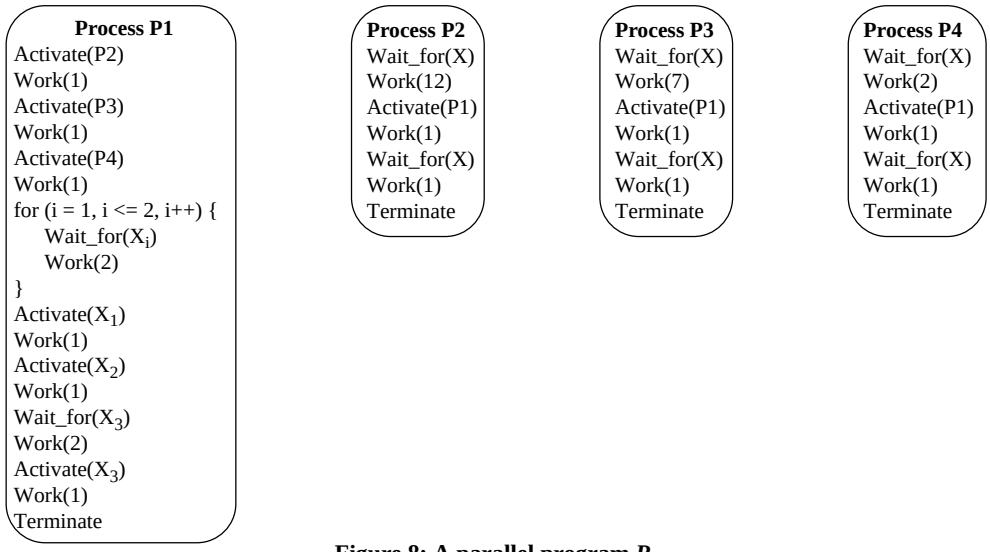


Figure 8: A parallel program *R*.

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

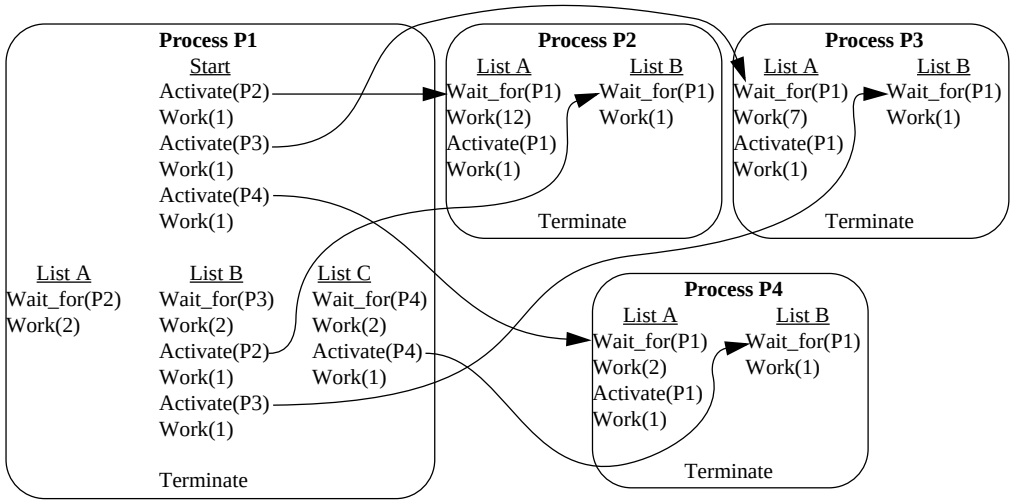


Figure 9: A parallel program *R”*.

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.

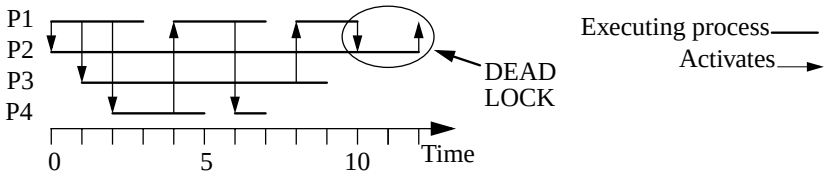


Figure 10: Execution flow of *R”*.

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'''*.

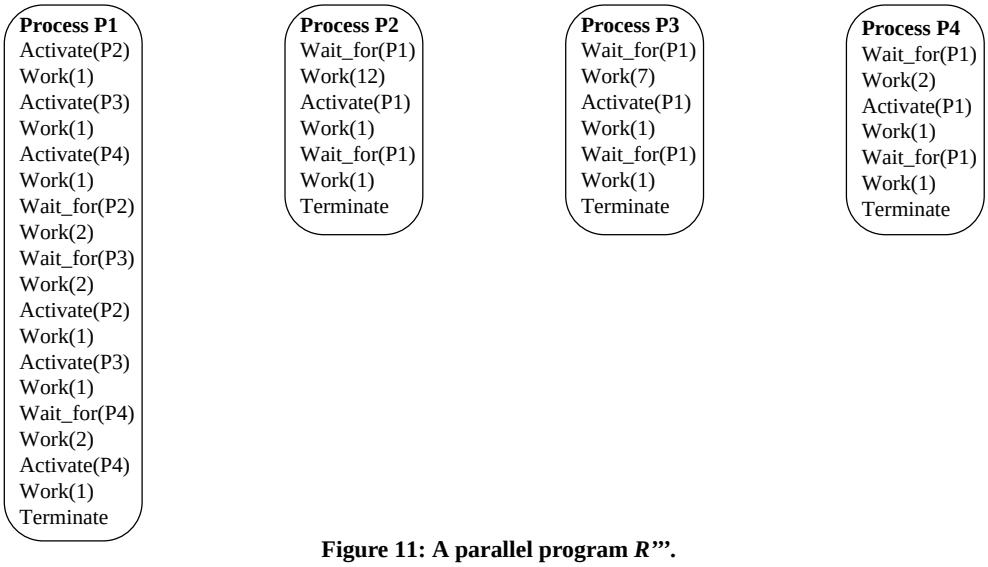


Figure 11: A parallel program *R'''*.

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).

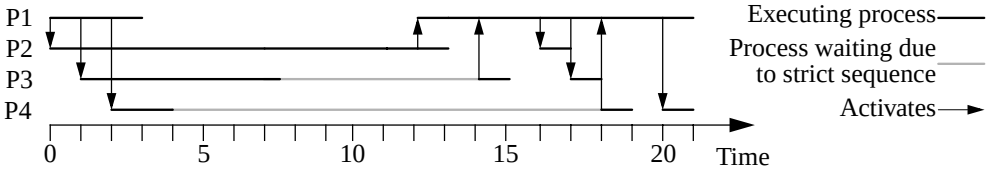


Figure 12: Execution flow of program *R'''*.

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.

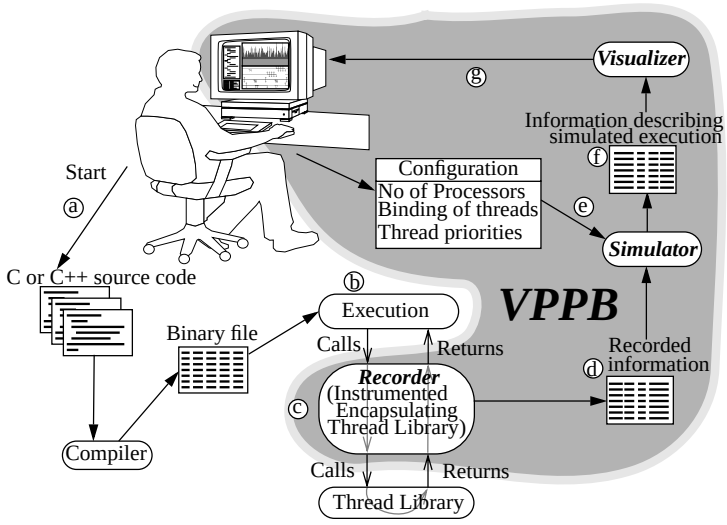


Figure 13: A schematic flowchart of the VPPB system.

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)	7.67	7.24 (5.6)	7.05 (8.1)	6.90 (10.0)

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 2.8% error, Client-Server Model has 15.9% and the Strict Sequence Model has 22.1%. 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.

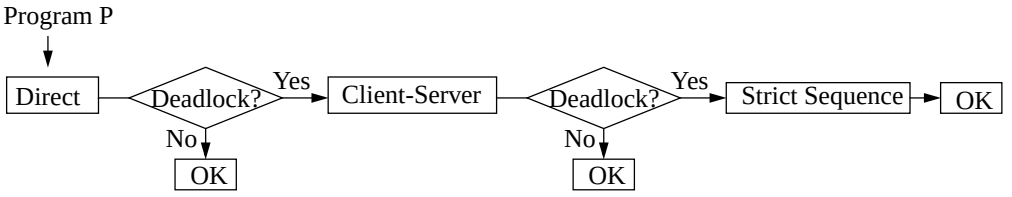


Figure 14: Practical scheme showing how to use the three simulation models.

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.