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.

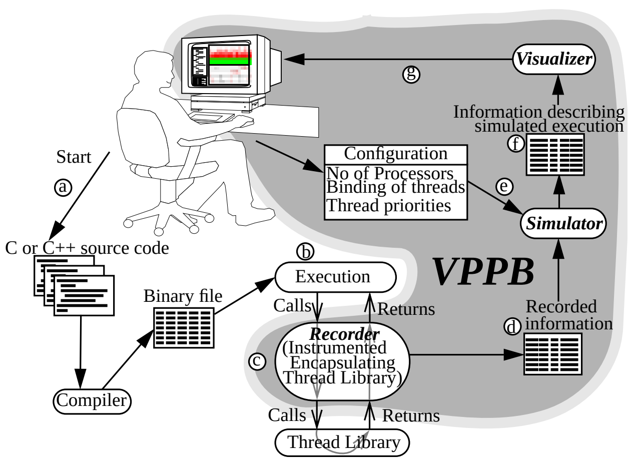

Figure 1: A schematic flowchart of the VPPB system.

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 $l » \varepsilon$. 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, $l » \varepsilon$	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.

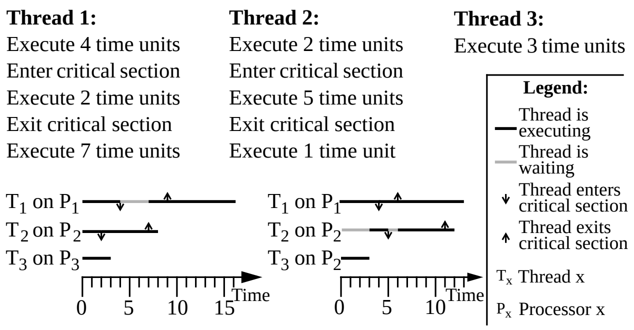
Thread 1:	**Thread 2:**	**Thread 3:**
Execute 4 time units	Execute 2 time units	Execute 3 time units
Enter critical section	Enter critical section	
Execute 2 time units	Execute 5 time units	
Exit critical section	Exit critical section	
Execute 7 time units	Execute 1 time unit	

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.

$$\text{Thread number: } \overbrace{1 \quad 2 \quad \ldots \quad P \quad P{+}1 \quad P{+}2 \quad \ldots \quad T}^{T}$$

$$\text{Number of tests: } \underbrace{\cancel{1} + 2 + \ldots + P}_{P} + \underbrace{P + P + \ldots + P}_{T-P} =$$

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

$$T \geq P \text{ and } P > 1$$

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.

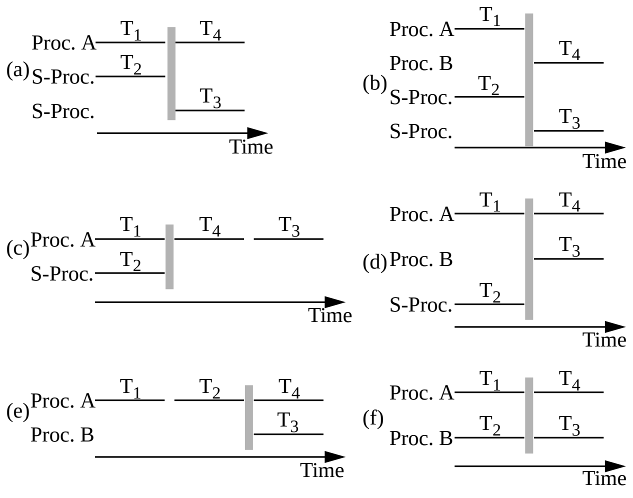

Figure 4: The different steps in the SGA algorithm for assigning the four threads in Section 3. A shadow processor is named S-Proc. The synchronization is the vertical bar, analogous to a barrier. The final assignment is the one in (f).

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.

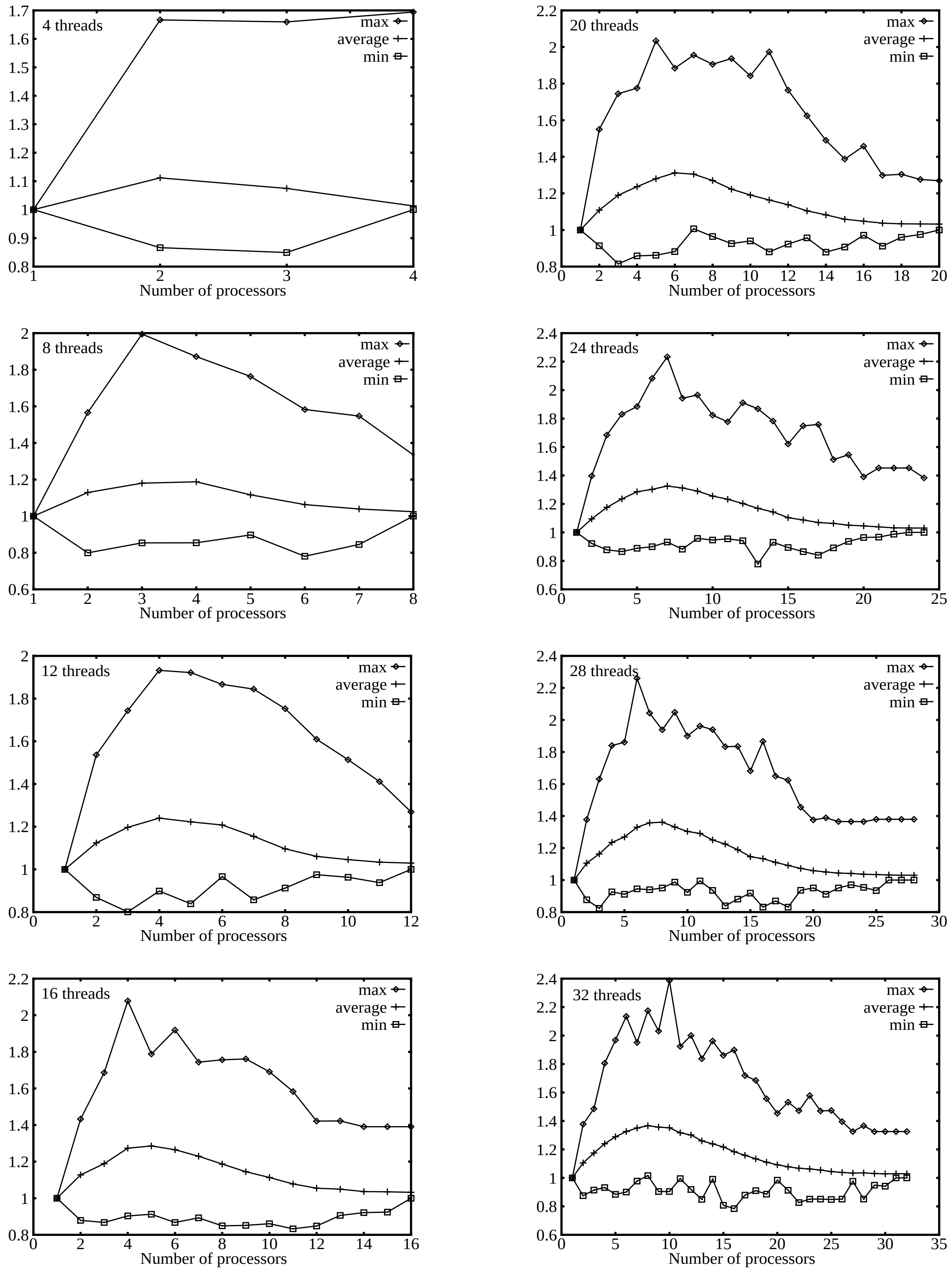

Figure 5: Simulation results for SGA. The Y-axis shows (Binpack execution time)/(SGA execution time).

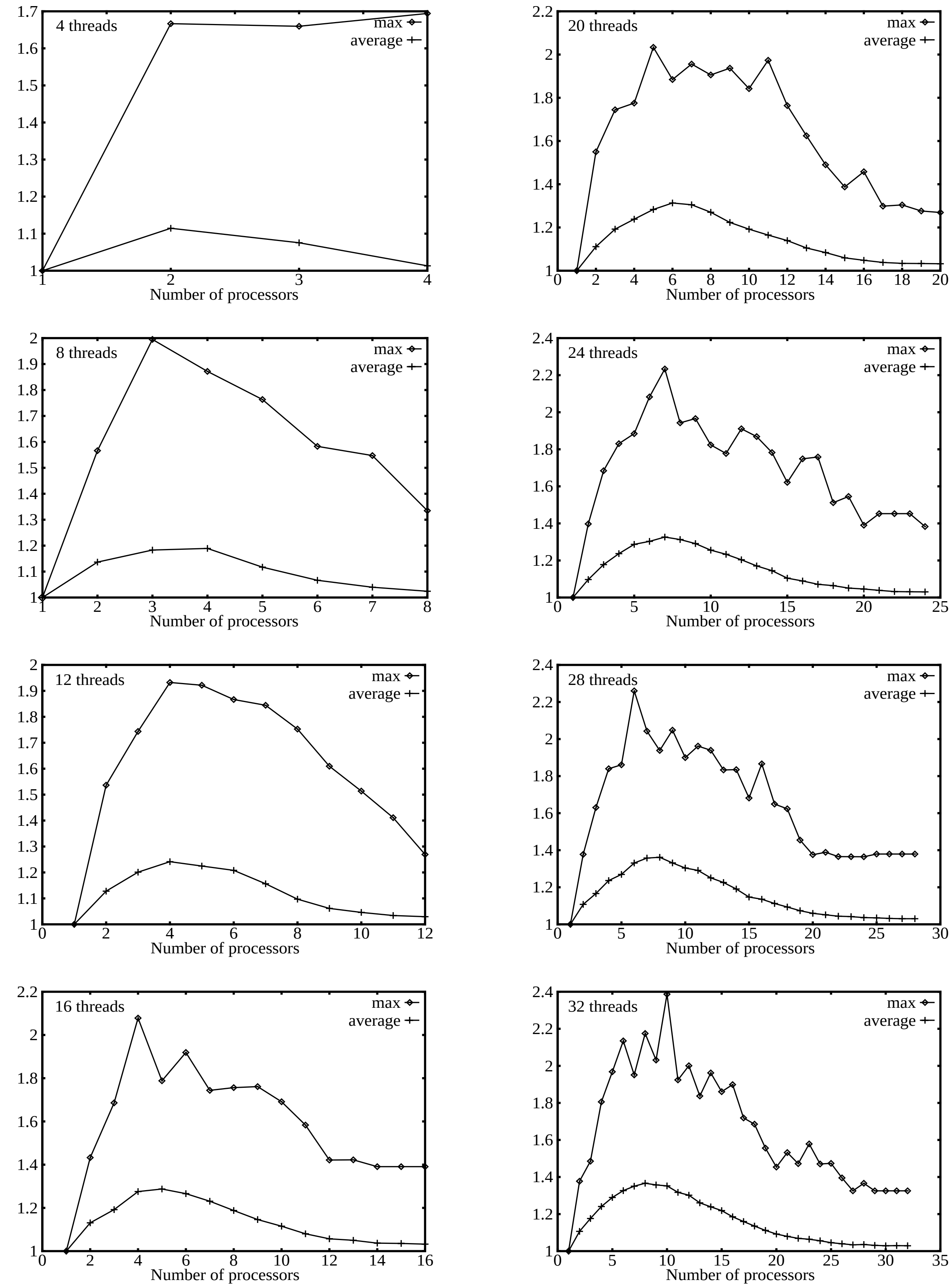

Figure 6: Simulation results for C-SGA. The Y-axis shows (execution time for binpacking) / (execution time for C-SGA).

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.