# 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,

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.

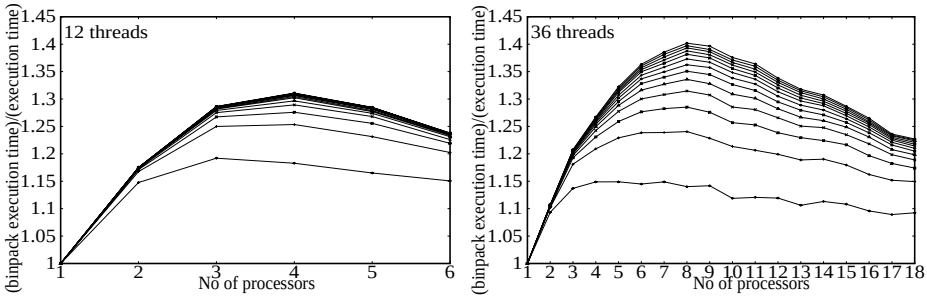


**Fig. 2.** The average gain by using the Greedy Algorithm as compared to binpacking

### 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.

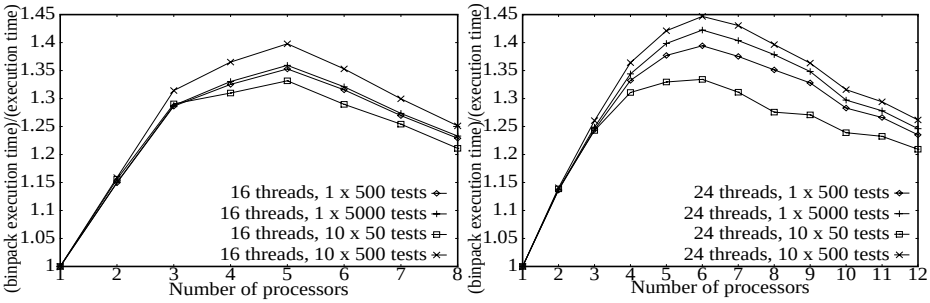


**Fig. 3.** The difference between 1 x 500 vs. 10 x 50 tests, and 1 x 5000 vs. 10 x 500 tests

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)