Bounding the Minimal Completion Time of Static Mappings of Multithreaded Solaris Programs

Lars Lundberg
Department of Computer Science, University of Karlskrona/Ronneby,
Soft Center, S-372 25 Ronneby, Sweden, email: Lars.Lundberg@ide.hk-r.se

1 Introduction

A multithreaded Solaris program can be executed in parallel on a multiprocessor. Previous experience show that, using the default scheduling algorithm, threads are frequently relocated from one processor to another [5]. After each such relocation, the code and data associated with the relocated thread have to be moved to the cache of the new processor. In order to avoid this problem, one can map threads to processors using the *processor_bind* directive [2]. The major problem with such static mappings is that one can easily end up with an unbalanced load. The problem of finding a mapping of threads to processors which results in minimum completion time is NP-hard [1]. It is even difficult to determine if a certain mapping is close the optimal case or if it is worth-while to look for other mappings.

Previous results [4] show that, based on certain information about the program, one can obtain a tight bound on the minimal completion time using static mappings, i.e. it is always possible to find a mapping with a completion time less than or equal to the bound. This makes it possible to determine if a certain mapping is close to the optimal case or if it is worth-while to look for other mappings. In this paper, we present a set of tools which make it possible to obtain such a bound, using an ordinary uni-processor workstation.

2 Method Overview

Figure 1 shows the steps used for bounding the minimum completion time of a static mapping of a multithreaded Solaris program P using a multiprocessor with k processors. Bold boxes indicate the parts developed by us. The routines in the Solaris thread library are overloaded with an instrumented thread library. For each call to a thread routine, a number of values are recorded, e.g. the identity of the thread making the call, the identity of the routine (e.g. *thr_create, thr_exit* and *sema_wait*) and the local process time when the call is made.

In order to bound the minimum completion time for a multiprocessor with k processors, we use an ordinary uni-processor workstation. The uni-processor execution of the multithreaded program, using the instrumented thread library, results in a list of recorded values. This list is then restructured into the dynamic program behaviour format, which is the information needed in order to calculate the bound. The bound calculation tool takes the dynamic program behaviour of the monitored program and the number of processors (k) in the multiprocessor system for which we are going to calculate the performance bound.

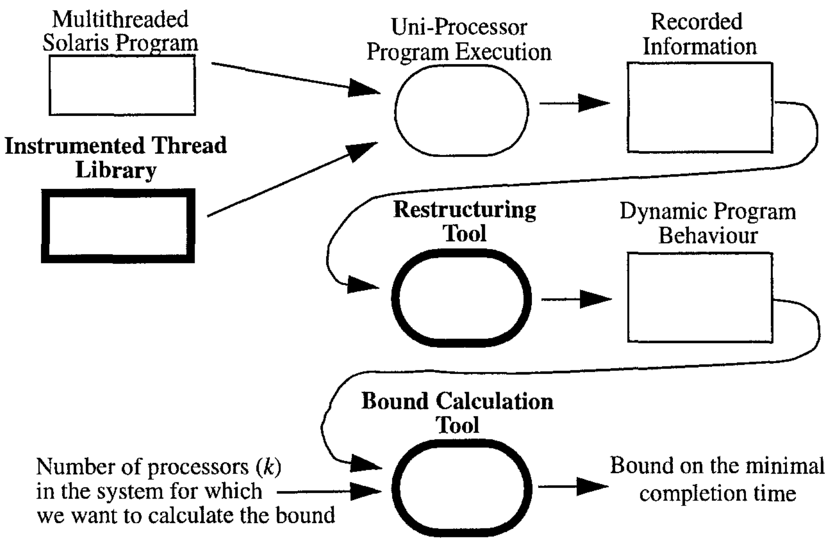

Multithreaded Solaris Program
Uni-Processor Program Execution
Recorded Information
Instrumented Thread Library
Restructuring Tool
Dynamic Program Behaviour
Bound Calculation Tool
Number of processors (k) in the system for which we want to calculate the bound
Bound on the minimal completion time

Figure 1: The tools which enable us to calculate a bound on the minimal completion time of any static mapping of a multithreaded Solaris program.

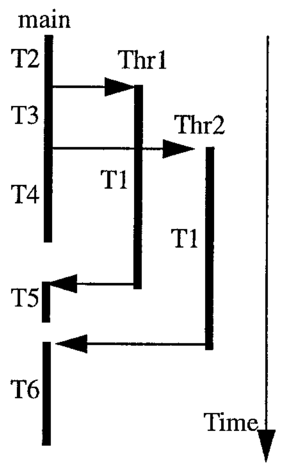
3 Calculating the Bound

The upper left part of figure 2 shows a multithreaded Solaris program, containing three threads: main, Thrl and Thr2. The upper right part of the figure shows a graphical representation of the execution of this program using one processor for each thread. The lower part of figure 2 shows a textual representation corresponding to the graphical representation; *work(t)* denotes sequential processing for *t* time units. Each thread is represented as a list of synchronization events separated by periods of sequential processing. These lists represent the dynamic program behaviour (see figure 1). The lists correspond to the sequence of events during the monitored execution. Therefore, the lists are completely deterministic. Indeterministic events, e.g. *mutex_trylock* are modelled by the corresponding deterministic events. If a *mutex_trylock* succeeds in the monitored execution, this is modelled as a *mutex_lock,* otherwise the event is not modelled at all.

The table below shows the recorded information generated during the monitored uni-processor execution of the multithreaded program in figure 2.

Process local time	**Thread Id**	**Thread routine**	**Parameters**
T2	main	thr_create	Thrl
T2+T3	main	thr_create	Thr2
T2+T3+T4	main	thr_join	Thrl
T2+T3+T4+T1	Thrl	thr_exit	--
T2+T3+T4+T1+T1	Thr2	thr_exit	--
T2+T3+T4+T1+T1+T5	main	thr_join	Thr2
T2+T3+T4+T1+T1+T5+T6	main	thr_exit	--

```
...
void thread_code(void *parameters) {
 ... // sequential processing for T1 time units
} //end thread_code
...
int main() {
 thread_t Thr1, Thr2;
 ... // sequential processing for T2 time units
 thr_create(0, 0, thread_code, param, 0, &Thr1);
 ... // sequential processing for T3 time units
 thr_create(0, 0, thread_code, param, 0, &Thr2);
 ... // sequential processing for T4 time units
 thr_join(Thr1, NULL, NULL);
 ... // sequential processing for T5 time units
 thr_join(Thr2, NULL, NULL);
 ... // sequential processing for T6 time units
} // end main
```



```
main               Thr1          Thr2
begin              begin         begin
 work(T2)           work(T1)      work(T1)
 thr_create(Thr1)  end           end
 work(T3)
 thr_create(Thr2)
 work(T4)
 thr_join(Thr1)
 work(T5)
 thr_join(Thr2)
 work(T6)
end
```

Dynamic Program Behaviour

Figure 2: Three representations of a multithreaded Solaris program. Upper part: the source code and a graphical representation of the execution using one processor for each thread. Lower part: a set of lists corresponding to the behaviour of the program.

The monitored execution of the multithreaded program is done on a uni-processor system using one LWP (Light Weight Process) for the entire program. The scheduling of threads to LWPs is non-preemptive, i.e. if there is only one LWP, a running thread cannot be preempted by another thread in the same program. This means that thread switching only occurs at a call to a routine in the thread library, e.g. *thr_exit, sema_wait* and *mutex_lock.* We monitor all these calls and record the time when they occur. Therefore, it is possible to restructure the recorded information into the lists (one list for each thread) representing the dynamic program behaviour.

4 Method Example

We are going to bound the completion time of a parallel implementation of an algorithm for generating prime numbers. A number of filters form a line with a number generator feeding numbers into the line. There is a prime number associated with each filter. Each filter filters out numbers which are divisible by its prime number, e.g. the first one filters out all even numbers. If a filter cannot divide a number, that number is forwarded to the next filter. Therefore, prime numbers reach the end of the line. When a prime number reaches the end of the line a new filter is created. The number generator stops by generating the number 3,000,000. The filter chain is cut up into contiguous subchains containing 500 filters each. Each subchain is executed by a Solaris thread. The maximum number of such subchains, which is reached at the end of the execution, is 44, i.e. $n = 44$.

The program was executed on a uni-processor version of a Sun Sparc Center 1000, using the instrumented thread library. Based on recorded values we calculate the bound. Figure 3a shows the bound in the interval ($1 \leq k \leq 8$).

Using the *processor_bind* routine, a simple mapping was implemented. The first $\lfloor n/k \rfloor$ threads are executed by processor zero (thread zero is the first subchain, thread one is the second, and so on). Similarly the next $\lfloor n/k \rfloor$ threads are executed by processor one. The last $n - (k - 1) \times \lfloor n/k \rfloor$ threads are executed by processor k-1. We refer to this as contiguous mapping, since the chain of threads is cut up into k contiguous subchains, and each subchain is mapped to a processor.

Figure 3b shows the completion time using contiguous mapping and the upper bound. The figure shows that the completion time using contiguous mapping is above the upper bound when the number of processors (k) is less than 6. Consequently, in this interval we know that contiguous mapping is not optimal, and it is thus worth-while to look for other mappings.

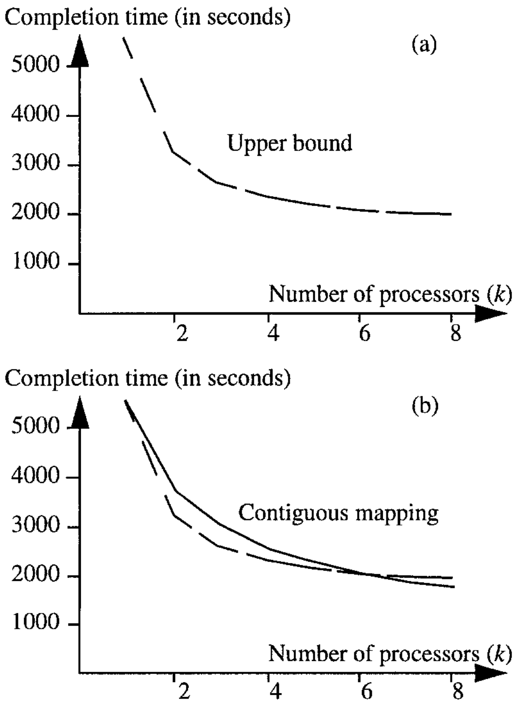


Figure 3: Comparing the completion using contiguous and round-robin mapping with the upper bound.

5 Conclusions

The technique is based on the assumption that the behaviour of the multithreaded program is (almost) deterministic. There are programs for which this assumption is reasonable, e.g. programs which multiply matrices of a certain size and programs which perform parallel image processing by splitting the image into fixed sized pieces.

The idea of using information from monitored uni-processor executions in order to estimate the multiprocessor performance of a parallel program is not new. The same idea has been used for parallel Ada and Fortran programs [3][6]. However, in these cases the goal was to predict the speedup of a the parallel program, i.e. no performance bounds were calculated in these projects.

The technique and tools described in this paper show that it is possible to integrate a theoretical performance bound in a real parallel programming environment, making the bound accessible to practitioners. The applicability of the result has been demonstrated by comparing the completion time of a real multithreaded Solaris program with the completion time bound. The bound itself, which was obtained in a previous study [4], is optimal in the sense that it cannot be improved within the definition of the problem. The tool used for obtaining the necessary information require no modification of the source code. Except for a minimal recording overhead, the behaviour of the multithreaded program is not affected by the recordings.

References

[1] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman and Company, 1979.

[2] B. Lewis and D. J. Berg, *Threads Primer*, Prentice Hall, 1996.

[3] L. Lundberg, *Predicting the Speedup of Parallel Ada Programs*, in Proceedings of the Ada-Europe 1992 Conference, Amsterdam, June 1992, Springer-Verlag, pp. 257-274.

[4] L. Lundberg and H. Lennerstad, *An Optimal Upper Bound on the Minimal Completion Time in Distributed Supercomputing*, in Proceedings of the 1994 ACM International Conference on Supercomputing, July, 1994, Manchester, England, pp. 196-203.

[5] L. Lundberg, *Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications*, in Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, Dijon, France, September, 1996, pp. 225-231.

[6] K. So, A. S. Bolmarcich, F. Darema and V. A. Norton, *A Speedup Analyser for Parallel Programs*, in Proceedings of the 1988 International Conference on Parallel Processing, August 1988, pp. 126-129.