General Techniques for Multithreading Algorithms

Perry Fizzano and Steven Swanson
Department of Mathematics and Computer Science
University of Puget Sound
Tacoma, WA, U.S.A.

Abstract *In this paper we develop techniques to multithread two large classes of algorithms. Namely, we consider recursive algorithms and matrix-based dynamic programming algorithms. Along with the general techniques we also developed two optimizations to improve the performance of our techniques in practice. Using the Pthreads library we implemented our techniques on an eight processor Sun Ultra-Enterprise. We report on experimental results comparing our multithreaded algorithms to sequential algorithms. These results show that we achieve a significant performance gain for computationally intensive problems while only a mild improvement for less computationally intensive problems. We also show how our multithreaded algorithms improve as we increase the number of processors.*

Keywords: Multithreading, Symmetric Multiprocessors (SMP), Dynamic Programming, Empirical Analysis

1 Introduction

We are seeing an increase in the availability of desktop multiprocessor machines as the cost of processors goes down. Such machines are commonly known as symmetric multiprocessors (SMP) and are currently available from every major vendor with the number of processors typically ranging from 2 to 16. To maximize the usefulness of such machines for an average user we need to see an increase in the number of standard applications that can take advantage of more than one processor. To this end we have concentrated on developing a set of techniques to transform any of a large class of sequential algorithms into multithreaded algorithms capable of taking advantage of more than one processor.

In order to implement our techniques we need to use a new language or a set of libraries for an existing language that allows us to take advantage of multiple processors. Choices include Pthreads [1] and Cilk [2]. We chose to use the Pthreads library to demonstrate our techniques because Pthreads is available on a wider variety of platforms, and one of our goals is to experiment on numerous architectures.

We have focused on multithreading two classes of sequential algorithms. The first is recursive algorithms and the second is matrix-based dynamic programming algorithms. We chose to study these because of the widespread use of these algorithms in standard applications.

In Section 2 we describe the M-stack, the data structure we use to distribute work among threads. Sections 3 and 4 discuss the algorithms we implemented using our techniques and the experiments we performed. Section 5 contains our conclusions and discusses possibilities for the future.

2 The M-stack

In each of our algorithms we maintain a stack of *contexts* that each correspond to a piece of work that needs to be done (we use the terms "work" and "context" interchangeably). We

refer to this stack as the *M-stack* to differentiate it from the run time stack or a stack used in a particular algorithm.

Our multithreaded algorithms have two parts. The *manager* portion of the algorithm pushes all the work that can be immediately processed onto the M-stack. Then τ threads start running the *thread body*. Each thread body repeatedly checks the M-stack for work, pops some off if there is any, processes it, and pushes any newly available work onto the M-stack. We must protect each access to the stack with a synchronization mechanism since we don't want two different threads processing the same context. This is achieved using mutual exclusion (i.e. mutex) primitives that Pthreads provides. The algorithms terminate when the M-stack is empty, all the threads have completed their work, and none of them are going to place any more contexts on the stack.

In addition to the actual stack of contexts, *work_stack*, an M-stack contains several other variables: A mutex variable, *work_stack_mutex*, protects *work_stack*, and the threads use the condition variable *work_stack_cond* to signal that more work is available. The thread body uses *done_cond* to tell the manager when the algorithm is finished, and the manager uses a flag, *die*, to tell the threads to exit.

Algorithm 2.1
Function MANAGER
  die := **False**.
  Place contexts on *work_stack*.
  Lock *done_mutex*.
  Start τ threads running the THREAD BODY.
  Wait for a signal on *done_cond*.
  die := **True**.
  Broadcast a signal on *work_stack_cond*.
  Release *done_mutex*.
End Function

Algorithm 2.1 is the generic manager. First, it pushes all the work that threads can process immediately onto the stack. Then it starts τ threads running the thread body and waits for a signal on *done_cond*. This signal means the stack is empty and all the threads are idle and waiting for more work to appear. When it receives a signal, MANAGER sets the *die* flag to tell the threads to exit instead of looking for more work. Finally, it signals the threads with *work_stack_cond* and they exit.

The thread body does all the work of the algorithm. Algorithm 2.2 contains the generic thread body and two functions, POP and PUSH, that manage *work_stack*. As long as there is work on *work_stack*, THREAD BODY removes it with POP, processes it, and uses PUSH to push newly available work onto the stack.

Algorithm 2.2
Function THREAD BODY
  While True Do
    c := POP.
    Process *c*.
    Use PUSH to place newly available contexts on *work_stack*.
  End While
End Function

Function POP
  Lock *work_stack_mutex*.
  While *work_stack* is empty **Do**
    If all the other threads are waiting on *work_stack_cond* **Then**
      Broadcast a signal on *done_cond*.
      Release *work_stack_mutex* and exit.
    End If
    Wait for a signal on *work_stack_cond*.
    If *die* = **True Then**
      Release *work_stack_mutex* and exit.
    End If
  End While
  Pop a context, *c*, off the *work_stack*.
  Release *work_stack_mutex*.
  Return *c*.
End Function

Function PUSH(*c*)
  Lock *work_stack_mutex*.
  Put *c* on *work_stack*.
  Signal on *work_stack_cond*.
  Release *work_stack_mutex*.
End Function

To place work on the stack, PUSH locks *work_stack_mutex*, pushes the work onto the stack, signals the threads (if any) that found the stack empty that more work is available, and releases *work_stack_mutex*.

If there is work on *work_stack*, POP locks *work_stack_mutex*, pops a context off the stack, releases *work_stack_mutex* and returns the context. If *work_stack* is empty, POP first checks to see if there are any threads that are still processing work. If there are threads still working, POP waits for a signal on *work_stack_cond*. If no other threads are processing work, there is no possibility that any more will appear, and the algorithm is finished. When this happens the thread broadcasts a signal to MANAGER on *done_cond* and exits.

Because PUSH and POP lock *work_stack_mutex* before accessing *work_stack*, only one thread examines or modifies the stack at a time.

2.1 Optimizing the M-stack

Let T_s denote the average amount of time spent maintaining the M-stack per context. Time spent locking mutexes accounts for most of this value, but it also includes the time for push and pop operations on *work_stack* as well as time spent maintaining any algorithm-specific dependency information. Let T_w denote the average amount of time spent actually processing a context. Finally, we define the *stack maintenance coefficient*, $M = \frac{T_s}{T_w}$.

For good performance we want M to be small. We can achieve this by decreasing T_s or increasing T_w. T_s depends on the computing platform and the algorithm in question, and changing its value is difficult. T_w depends on the algorithm as well, but we can affect its value in the following two ways.

The first, *stack buffering*, is universally applicable. In a k-buffered M-stack, the POP function removes k contexts at a time. The thread body then processes these k contexts and places newly available contexts on the stack all at once.

The second method is *context enlargement*. We make the contexts "bigger" so they take longer to process. Unfortunately, using context enlargement depends on the programmer finding a reasonable way to enlarge the contexts for the algorithm under consideration. This may not always be possible, but for many problems it is. For some classes of algorithms, like matrix-based dynamic programming algorithms, there is a context enlargement technique that always works.

The most important difference between stack buffering and context enlargement is how they affect the number of mutex locks the algorithm performs. Stack buffering only reduces the number of times the algorithm locks *stack_mutex*. The algorithms we will examine need to use other mutexes to protect dependency information, so buffering will not significantly reduce the overall number of locks.

With context enlargement the programmer uses problem specific information to enlarge contexts in a logical, appropriate way, hopefully reducing the amount of dependency information involved. This means that context enlargement will reduce the total number of locks more dramatically, and that should make context enlargement more effective in practice.

3 Applications Studied

We implemented three algorithms using our techniques.

The first algorithm we implemented, MATRIX CHAIN, found the optimal ordering for a chain of matrix multiplications [4]. Given an expression $A = A_1 A_2 \ldots A_k$ where the A_i is an $n_i \times n_{i+1}$ matrix for $1 \leq i < k$, the number of scalar multiplications we need to perform in order to compute A depends on how we parenthesize $A_1 A_2 \ldots A_n$. For example, if the matrices A_1, A_2, and A_3 are 5×10, 10×20, and 20×1 respectively, then multiplying $((A_1 A_2) A_3)$ requires 300 multiplications while multiplying $(A_1 (A_2 A_3))$ requires only 250. The objective is to find a parenthesization that minimizes the number of scalar multiplications.

The second algorithm we implemented solved the Longest Common Subsequence problem [4]. A *subsequence* of a string, $S = s_1 s_2 \ldots s_n$, is simply S with some (possibly all or none) of its elements left out. Given two strings S and T, the longest common subsequence problem is to find the longest subsequence of S that is also a subsequence of T. For example, the longest common subsequence of bcdadab and dbcacab is bcaab.

The solutions to these two problems use dynamic programming [4]. Specifically, they work by calculating all the entries in a matrix, M. We call algorithms of this type *matrix-based dynamic programming* algorithms. Multithreading matrix-based dynamic programming algorithms using MANAGER and THREAD BODY is simple.

For example, Algorithm 3.1 contains psuedocode for the thread body of our multithreaded version of LONGEST COMMON SUBSEQUENCE. It operates on two sequences, S and T, and we denote the entry of M in the ith row and jth column as $M_{i,j}$. There are two differences between this algorithm and the generic thread body. The first is addition of the nested **If** statement after the call to POP. This statement is the core of the sequential version of LONGEST COMMON SUBSEQUENCE.

The second is the **Foreach** loop. It ensures that the entries of M are computed in a correct order. In this case, a correct order is any order that ensures that $M_{i,j}$ is not computed before $M_{i-1,j}$, $M_{i,j-1}$, and $M_{i-1,j-1}$ are finished. This ordering is important because $M_{i,j}$ depends on those three entries. To maintain the dependency information the algorithm uses a counter for each entry, $c(M_{i,j})$, and protects it with a mutex, $\text{mut}(M_{j,i})$. At the beginning of the algorithm MANAGER sets $c(M_{i,j})$ to the number of entries that $M_{i,j}$ depends on. When the thread body computes $M_{i,j}$ it decrements the counter for each entry that depends on $M_{i,j}$. When a counter reaches 0, the algorithm pushes the corresponding entry onto the M-stack.

Algorithm 3.1
Function LONGEST COMMON SUBSEQUENCE
  While True Do
    $M_{i,j} =$ POP.
    If $t_j = s_i$ **Then**
      $M_{i,j} := M_{i-1,j-1} + 1$
    Else
      If $M_{i-1,j} \geq M_{i,j-1}$ **Then**
        $M_{i,j} := M_{i-1,j}$
      Else
        $M_{i,j} := M_{i,j-1}$
      End If
    End If
    Foreach $M_{l,k}$ that depends on $M_{i,j}$ **Do**
      Lock $\text{mut}(M_{l,k})$.
      $c(M_{k,l}) := c(M_{k,l}) - 1$.
      If $c(M_{k,l}) = 0$ **Then**
        Release $\text{mut}(M_{l,k})$.
        PUSH$(M_{l,k})$.
      Else
        Release $\text{mut}(M_{l,k})$.
      End If
    End Foreach
  End While
End Function

We devised a general method for enlarging contexts for matrix-based dynamic programming. We divide M into sub-matrices and each thread computes an entire sub-matrix before going back to the M-stack for more work. We can think of the sub-matrices of M as entries in a new matrix M'. It is easy to see that we can process the entries of M' in the same order as we processed the entries of M.

The third algorithm we implemented was quicksort, a well known recursive sorting algorithm [3]. It performs very well in practice and is the fastest general purpose sorting algorithms available.

Multithreading recursive algorithms is more complicated than multithreading matrix-based dynamic programing algorithms. First, we need to handle algorithms that make a series of recursive calls, process the results, and possibly make another series of calls. Second, we need to address recursive algorithms that use more than one function (e.g. doubly recursive

algorithms). Finally, we need to be able to deal with return values.

To handle multiple sets of recursive calls we divide each function into *phases*. Each phase contains a set of recursive calls that are mutually independent. We add space for a phase value to each context, c, and denote it phase(c). When a thread removes a context from the stack, it checks its phase and processes it accordingly, producing contexts for that phase's recursive calls, if any. We say that c is the *parent* of these contexts. The thread pushes the new contexts onto *work_stack*. The thread also sets a counter in c, F(c), to track how many of its recursive calls are uncompleted. A mutex variable, $F_{mut}(c)$, protects F(c).

We deal with multiple functions by adding a *type* field to each context. We denote it type(c). Each type corresponds to a different function. Different types of context can have different numbers of phases. As with the phase information, the threads check the context's type before processing it.

To allow functions to return values we define a new function RETURN (see Algorithm 3.2). RETURN decrements the number of unprocessed children that belong to c's parent, parent(c), and pushes parent(c) onto the stack if all its children are finished. The value that the function returns must be stored in c so the threads can access it when processing parent(c).

Algorithm 3.2
Function RETURN
  If c has a parent **Then**
    Lock F_{mut}(parent(c)).
    Decrement F(parent(c)).
    If F(parent(c)) $= 0$ **Then**
      Release F_{mut}(parent(c)).
      PUSH(parent(c)).
    Else
      Release F_{mut}(parent(c)).
    End If
  End If
End Function

Although we report results for experiments with quicksort, we use merge sort to demonstrate our technique. This is because quicksort has only one phase and does not need to call RETURN. As a result, applying our technique is not very interesting. Multithreading quicksort is left as an easy exercise.

Algorithm 3.3 contains the code to replace "Process c" in Algorithm 2.2 in a multithreaded version of merge sort. The code operates on a segment of an array, A, with end points $p(c)$ and $q(c)$ where c is the current context. MERGE SORT has a single type of context with two phases.

Algorithm 3.3
Case phase(c) **Of**
  1:
    /* child contexts are c_1 and c_2. */
    phase(c_1) := phase(c_2) := 1.
    type(c_1) := type(c_2) := 1.
    parent(c_1) := parent(c_2) := c.
    $p(c_1) := p(c)$.
    $q(c_1) := \lfloor \frac{p(c)+q(c)}{2} \rfloor$.
    $p(c_2) := \lfloor \frac{p(c)+q(c)}{2} \rfloor + 1$.
    $q(c_2) := q(c)$.
    F(c) := 2.
    phase(c) := 2.
    PUSH(c_1).
    PUSH(c_2).
  2:
    MERGE(A, $p(c_1)$, $q(c_2)$).
    RETURN.
End Case

For our multithreaded version of QUICK SORT we can enlarge contexts by having each thread recurse several times before placing children on the stack. The first context would correspond to the first k levels of recursion, and it would produce about 2^k contexts (depending on the evenness of the partitions), each of which would represent k additional levels of recursion. We call this technique *vertical breaking*. Vertical breaking is only applicable to algorithms like QUICK SORT that do not do work after their recursive calls. For example, vertical breaking does not work with MERGE SORT,

because it needs to merge the two arrays after it sorts them.

4 Experiments

We implemented our algorithms using the Pthreads library on an 8-processor Sun Ultra-Enterprise at Oregon State University running Solaris version 5.6.

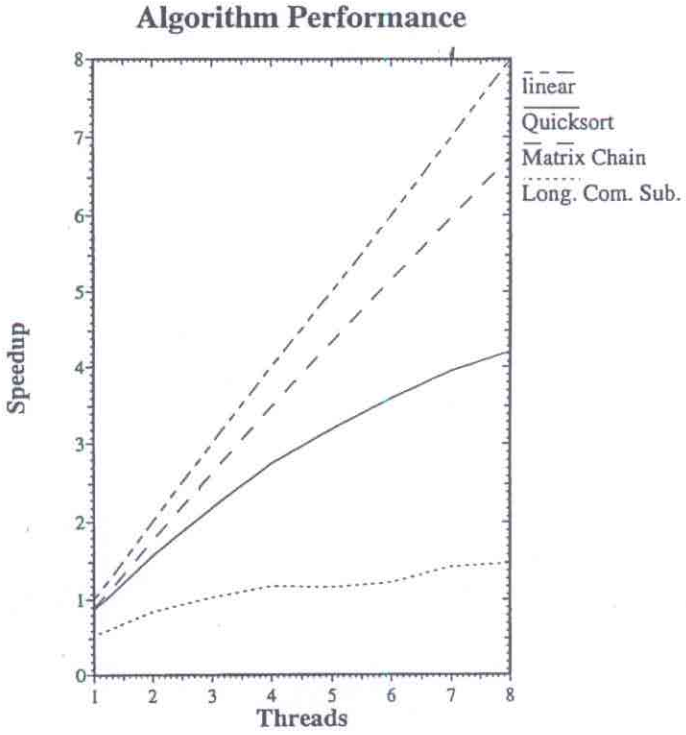
Our data collection on the Sun had two phases. In the first phase we found the optimal amount of stack buffering and context enlargement for each algorithm for both four and eight threads.

In phase two we determined how the performance of the optimized and unoptimized algorithms compared. Table 1 shows the running time for each algorithm with and without optimizations. Notice that without optimizations, only MATRIX CHAIN's performance improves with 8 threads instead of 1. This is because the other algorithms spend a large amount of time maintaining the stack. For instance, the unoptimized version of LONGEST COMMON SUBSEQUENCE performed around 4,000,000 mutex locks but with context enlargement it needed to perform only 10,000. Clearly, the optimizations can make a substantial difference.

We also measured how much faster the threaded algorithms are compared to an efficient unthreaded version. If thread overhead takes too much time, our techniques might not provide a significant, real performance gain. Figure 1 shows the speedup for all three threaded algorithms relative to their unthreaded counterparts. MATRIX CHAIN, with its naturally large contexts, achieves nearly linear speedup for up to eight processors. The performance of LONGEST COMMON SUBSEQUENCE and QUICK SORT improved as the number of threads increased but the improvement was not as dramatic as for MATRIX CHAIN.

While it is reassuring that most of our algorithms performed very well, the reasons that some of our optimized algorithms did not perform equally well are also interesting.

Figure 1: Speedup for Optimized Algorithms Compared to a Efficient Sequential Algorithms



In particular LONGEST COMMON SUBSEQUENCE's performance is disappointing. The most likely explanation for this is that LONGEST COMMON SUBSEQUENCE's contexts are too small, and we simply cannot make them big enough to significantly lower the stack maintenance coefficient. This points out a fundamental limitation of our techniques: If an algorithm lacks sufficient complexity it will be very difficult to significantly improve its performance with multithreading.

These data also suggests that mutex locking time increasing with the number of threads running. This would explain why running time increased with more threads for the unoptimized versions of QUICK SORT and LONGEST COMMON SUBSEQUENCE. Turning off optimizations dramatically increases the number of contexts that need to be processed and would make any increase in mutex locking overhead more prominent.

Table 1: Running times and speedups for optimized and unoptimized algorithms. Times are in seconds.

Algorithm	Unoptimized			Optimized		
	$\tau = 1$	$\tau = 8$	Speedup	$\tau = 1$	$\tau = 8$	Speedup
MATRIX CHAIN	207.0	28.5	7.3	201.1	26.5	7.6
LONGEST COM SUBSEQ	50.4	67.9	0.7	6.5	2.3	2.8
QUICK SORT	23.2	37.9	0.6	11.6	2.4	4.9

5 Conclusions and Future Work

We developed general techniques for creating portable, multithreaded versions of conventional recursive and matrix-based dynamic programming algorithms. The effectiveness of the techniques varies from algorithm to algorithm and is primarily dependent on the fraction of time each spends maintaining the M-stack and dependency information. In almost all cases some optimization in the form of context enlargement or stack buffering is needed to achieve good performance. For the optimized versions we achieved speedup of between 1.6 and 6.7 by using eight threads instead of using a sequential algorithm.

We would like to extend this work by testing our algorithms on a larger number of processors. We do not expect the speedups to continue to increase indefinitely, but we would like to empirically measure the point where they level off. We would also like to apply these techniques to other real applications like the diff utility in UNIX. We believe that a utility like diff (which can be solved using matrix-based dynamic programming) would have a low stack maintenance coefficient and that we would realize good performance gains by multithreading it with the techniques presented here. We would also like to measure performance on other platforms. We do know that the code works on several other platforms, but we are still working on testing on these platforms.

Acknowledgments

We would like to thank Professor Mike Quinn and the Department of Computer Science at Oregon State University for providing us with dedicated access to their eight processor Sun Ultra-Enterprise.

References

[1] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming.* O'reilly & Associates, Sebastopol, CA, 1996.

[2] C. F. Joerg. *The Cilk System for Parallel Multitheaded Computing.* PhD thesis, MIT, Cambridge, MA, January 1996.

[3] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.