EXECUTING NESTED PARALLEL LOOPS ON SHARED-MEMORY MULTIPROCESSORS

Sadun Anik Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, Illinois 61801

Abstract - - *Cache-coherent, bus-based shared-memory multiprocessors are a cost-effective platform for parallel processing. In scientific parallel applications, most of the computation involves processing of large multidimensional data structures which results in a high degree of data parallelism. This parallelism can be exploited in the form of nested parallel loops. Most existing shared memory multiprocessors exploit this multi-level parallelism at only one level. In this paper, we explore efficient algorithms and models for executing nested parallel loops and present a simulation based performance comparison of different techniques using real application traces. We show that it is possible to exploit the parallelism in nested parallel loops with the use of good scheduling and synchronization algorithms.*

INTRODUCTION

Bus-based shared memory multiprocessors have a moderate number of processors (8-32), and unlike the grand challenge problems to be solved on massively parallel systems, most applications to be run on these multiprocessors are moderate in size. In this study, we examine several compiler parallelized applications from the Perfect Club Benchmark Suite [1] which have nested loop parallelism These programs are parallelized with the KAP parallelizer [2]. Our experience with these applications is that most programs use moderate sized data structures, in the range of 400-10000 elements. When a two dimensional array of 900 elements (30×30) is processed in a doubly nested parallel loop, the parallelism at each level is limited to 30. This demonstrates the need to exploit the parallelism at both loop nest levels efficiently.

One source of overhead in executing parallel loops is due to scheduling tasks (parallel loops) to processors and synchronizing processors at the end of loop execution[3][4]. This overhead becomes significant when the total amount of computation in a parallel loop is small. This is frequently the case for innermost parallel loops due to both finer granularity and relatively small number of iterations.

A component of overhead in parallel loop execution is the time spent by processors waiting to acquire locks, which are used in scheduling algorithms. Most existing high performance lock algorithms, e.g. tournament lock and queuing lock, are blocking algorithms[5][6]. That is, a processor which is trying to acquire a lock is committed until the lock is acquired. This makes it impossible for a processor to utilize the idle time spent waiting for a lock. In the next section we present a non-blocking version of the queuing lock algorithm and describe a task distribution algorithm which uses this non-blocking property to allow processors to wait for multiple task queues simultaneously.

MODELS FOR NESTED PARALLEL LOOP EXECUTION

An application with nested parallel loops can be executed in several ways depending on the available compiler, hardware and run-time system support. The simplest model involves executing the outermost loop in parallel and all the inner parallel loops sequentially. This model is employed in the run-time system of existing shared memory multiprocessors [7]. It has the advantage of being simple but exploits only part of the available parallelism.

Another model for executing nested parallel loops involves exploiting the parallelism at a single level but using compiler transformations to *collapse* the nested loops into a one level loop. Such compiler transformations are a current field of research; study of different transformations and their applicability to particular loop structures is beyond the scope of this paper. We will assume ideal loop restructuring where DOALL loop nests are transformed into single-level DOALL loops. Although such a transformation may not be realistic, it does provide a good reference point in comparing performance of different models.

The third model of execution is to execute the inner loops in parallel on all processors. A blocking barrier is used at the end of each parallel loop, which prevents the overlapping between execution of inner loops. However, the execution of the sequential code after an inner loop can overlapped with that of the next loop at the same nest level.

A refinement of the third model involves relaxing the definition of barrier synchronization by allowing all processors — except for the one which is going to execute the code after the loop — to leave the barrier immediately upon entrance and start executing other parallel loops. This model can improve performance by dynamically overlapping the

Table 1: Execution models for nested parallel loops

execution model	description
1	Inner loops are executed sequentially
2	Nested loops are collapsed (ideal)
3	Nested execution, blocking barriers
4	Nested execution, non-blocking barriers
5	Nested execution, multiple simultaneous access task queues

execution of the barrier with that of another parallel loop.

The final model is a further refinement of the fourth model. This execution model increases the task scheduling throughput by using multiple task queues. All the processors can access all the task queues, and a task, i.e., parallel loop, can be scheduled to multiple processors. A summary of these five models are shown in Table 1

Although multiple task queues have been proposed before [3] to increase task scheduling throughput, this method does not require an assignment of processors to task queues therefore does not introduce a load balancing problem. Access to task queues are controlled by non-blocking locks and each processor tries to gain exclusive access to all the queues simultaneously. A processor checks the locks that it is waiting for in a round robin fashion. When a processor acquires the lock of a task queue, it stops trying to gain access to the remaining task queue locks. We present a non-blocking queuing lock algorithm whose operation is similar to the queuing lock while allowing a processor to abort a lock access. Unlike the previous low-contention, high-performance lock algorithms such as queuing lock, this new algorithm does not require the processors to commit themselves upon a lock operation [5],[6].

The non-blocking queuing lock algorithm shown in Figure 1 uses the *fetch&add* primitive to set up a lock queue. The atomic *fetch&add* operation is used to obtain a unique number to set up a queue. The array a is used for mapping these unique numbers to the processors. The algorithm assumes that on a P processor system, the processors are numbered 0 to P-1, and the variable me[lock_id] is initialized to this number. A processor entering the lock queue reads array a for the id of its predecessor in the queue.

A processor in the lock queue waits for its predecessor to release the lock. The array b is used for signaling between the processors. A processor leaving the lock queue also uses array b to pass the id of the processor ahead to the processor behind for proper operation of the queue. While the lock is held by a processor, other processors can enter and leave the lock queue many times. The algorithm includes several checks to prevent race conditions and to ensure the proper use of arrays a and b.

The task scheduling algorithm we use for model 5 to increase task distribution throughput is a straight forward extension of centralized task scheduling. A fixed number of task queues are used for distributing tasks. Each processor tries to gain exclusive access to the queues which are not empty. Upon gaining access to a task queue, it leaves the other lock queues.

Figure 2 illustrates the simultaneous use of multiple task queues. In the first snapshot (left side) processors P2 and P4 own the locks for accessing the the two task queues. Processors P1, P3 and P5 are waiting for both locks. In the second snapshot (right side), P2 releases the lock of task queue 1 and the next processor in the queue of lock 1, P3, acquires lock 1. After acquiring lock 1, P3 leaves the queue of lock 2. Therefore P1 and P5 advance in both lock queues.

Balancing the utilization of each queue is important for the throughput of task scheduling. In the case where all loops are distributed from one of the task queues, the processors execute the loops as in model 4. To balance the number of loops scheduled from task queues, processors inserting tasks to the queues start from a random queue and visit queues in round-robin fashion for each new parallel loop.

EXPERIMENTAL RESULTS

To compare the performance of different models for executing nested parallel loops, we use traces from three parallelized Perfect Club applications: FLO52, DYFESM and ADM. Of the thirteen programs in the Perfect Club set, four of them are parallelized to the extend that at least half of the computation is done in parallel loops. Out of these four applications, the parallel loops in the BDNA program are not nested. This leaves us with the three applications listed above for this study.

Among the three programs, FLO52 has the highest level of parallelization. Only 1% of the dynamic instructions in the program trace are in sequential sections. The percentage of sequential instructions in the trace is considerably higher for the other two programs; around 25% for DYFESM and 32% for ADM. The programs have varying levels of granularity and parallelism for the innermost parallel loops. FLO52 has an average parallelism of 58 iterations per innermost parallel loop and a granularity of 39 instructions per inner loop iteration. DYFESM has an average parallelism of 14 and a granularity of 112 at innermost parallel loops, and these numbers for ADM are 11 and 48 respectively.

The performance results are obtained using a trace driven, bus transaction level shared-memory multiprocessor simulator. The simulated multiprocessor supports an atomic *fetch&add* operation and this operation is used to implement iteration self-scheduling and linear barrier algorithms. We would like to note that exploitation of high degree of parallelism in a single level DOALL loop may make it possible to efficiently use different types of iteration scheduling algorithms other than self-scheduling [8].

The speedup figures for the three programs, FLO52, DYFESM, and ADM are shown in Figures 3, 4, and 5 respectively. These figures show that nested parallel loop exe-

```
void initialize(lock_id)
{  for (i=0;i<P+1;i++) a[lock_id][i] = EMPTY ;
   a[lock_id][P+1] = 2*P
   for (i=0;i<2*P;i++) b[lock_id][i] = BUSY ;
   b[lock_id][2*P] = FREE ;
   lock_counter = 0 ;
}
int lock(lock_id)
{  myturn = fetch&add(lock_counter,1) % (P+2) ;
   myturn_minus_one = myturn+P+1 % (P+2)
   my_id = me[lock_id] ;
   me[lock_id] = (me[lock_id] + P) % (2*P) ;
   while(a[lock_id][myturn_minus_one] == EMPTY) ;
   ahead_of_me[lock_id]=a[lock_id][myturn_minus_one];
   while(b[lock_id][my_id] != BUSY) ;
   a[lock_id][myturn] = my_id ;
   a[lock_id][myturn_minus_one] = EMPTY ;
   else return(check_lock(lock_id)) ;
}
int check_lock(lock_id)
{  lock_status=b[lock_id][ahead_of_me[lock_id]] ;
   if(lock_status == BUSY) return(LOCK_BUSY) ;
   else if(lock_status == FREE) {
      b[lock_id][ahead_of_me[lock_id]] = BUSY ;
      return(LOCK_OWNED) ;
   } else {
      b[lock_id][ahead_of_me[lock_id]] = BUSY ;
      ahead_of_me[lock_id] = lock_status ;
      return(check_lock(lock_id) ;
   }
}
void release_lock(lock_id)
{  b[lock_id][my_id] = FREE ;
}
void leave_lock(lock_id)
{  check_lock(lock_id) ;
   b[lock_id][my_id] = ahead_of_me[lock_id] ;
}
```

Figure 1: Non-blocking queuing lock algorithm with *fetch&add* primitive

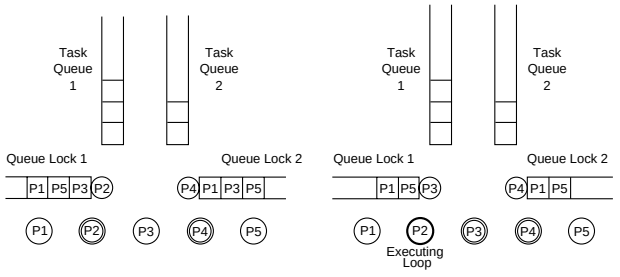

Figure 2: Operation of task distribution algorithm

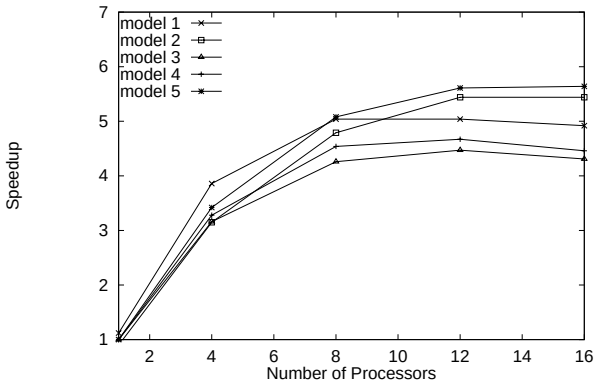

Figure 3: Speedup of FLO52 program for different execution models

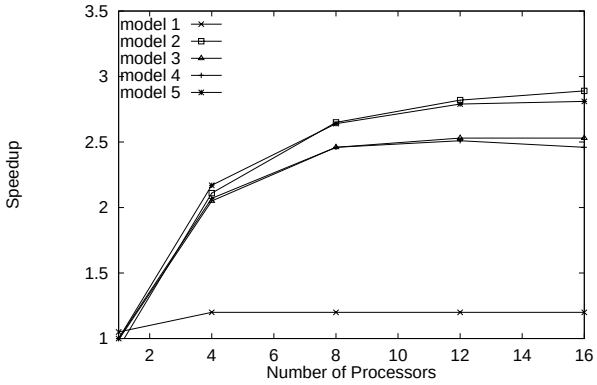

Figure 4: Speedup of DYFESM program for different execution models

cution with non-blocking barriers and multiple task queues (model 5) and perfect loop collapsing (model 2) perform consistently better than other execution models when the number of processors is large. Furthermore these two models result in similar performance. This demonstrates that for the architecture model we used, performance gains of perfect loop collapsing by a compiler can be achieved by executing nested parallel loops with efficient run-time scheduling and synchronization algorithms.

Executing the outermost loop in parallel and inner loops sequentially (model 1) results in different execution behavior among the three programs. In FLO52, where parallelization is the most successful, the outer loop parallelism is sufficient for achieving good speedup. It actually performs slightly better than models 3 and 4, which are simpler models for executing the innermost loop in parallel. The poor performance of these two models for this program can be attributed to the low granularity of innermost loops — hence high overhead for iteration scheduling. For DYFESM, model 1 results in the loss of almost all the avail-

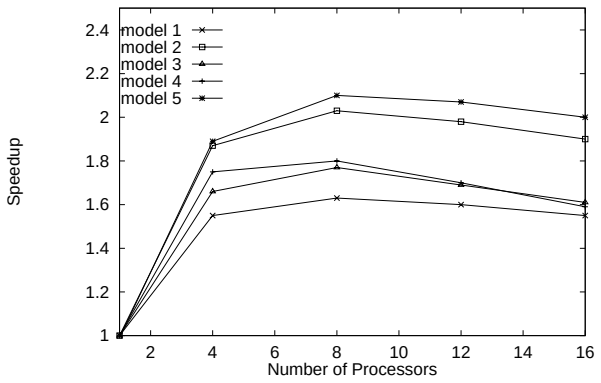

Figure 5: Speedup of ADM program for different execution models

able parallelism. This model also results in the worst performance for program ADM though the speedup was close to those obtained from the models 3 and 4.

Finally, we observe that use of non-blocking barriers alone do not contribute significantly to performance. The performance effects of the task distribution algorithm are much more significant than those of the barrier synchronization algorithm.

CONCLUDING REMARKS

In this paper, we examine several practical aspects of nested parallel loop execution. We use five different models for executing nested parallel loops. We conclude that when parallelism is exploited only at the outermost loop level, performance can degrade due to loss of parallelism. Exploiting parallelism of a nested loop structure at a single level through compiler loop collapsing can be very effective to decrease task scheduling overhead.

We explore three variations of true nested parallel loop execution. In the first one, model 3, innermost loops are executed one at a time where synchronization is done with a conventional barrier. We then use a non-blocking barrier version of this model, model 4, where processor idle time in a barrier is minimized. However, this does not result in any significant performance difference. As a refinement to this model, in model 5 we use a non-blocking queuing lock algorithm and a multiple task queue based task distribution algorithm to decrease task scheduling overhead. Our simulation results show that this model can achieve the performance of a perfect loop collapsing transformation.

Acknowledgements

This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, NCR, AMD, Matsushita, Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS.

REFERENCES

[1] M. Berry and et al., "The perfect club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD Rpt. No. 827, Center for Supercomputing Research and Development, University of Illinois, 1989.

[2] Kuck & Associates, Inc., *KAP User's Guide*, version 6 ed., 1988.

[3] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance of thread management alternatives for shared memory multiprocessors," *Proceedings of SIGMETRICS*, pp. 49–60, 1989.

[4] C. D. Polychronopoulos, "The impact of run-time overhead on usable parallelism," *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 108–112, August 1988.

[5] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *Transactions on Parallel and Distributed Systems*, vol. 1, No. 1, pp. 6–16, 1990.

[6] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *Computer*, pp. 60–69, June 1990.

[7] Alliant Computer Systems Corp., *Alliant FX/Series Architecture Manual*, 1986.

[8] C. D. Polychronopoulos, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *Transactions on Computers*, vol. C-36, No. 12, pp. 1425–1439, December 1987.