June 1991

UILU-ENG-91-2231
CRHC-91-21

Center for Reliable and High-Performance Computing

PERFORMANCE IMPLICATIONS OF SYNCHRONIZATION SUPPORT FOR PARALLEL FORTRAN PROGRAMS

Sadun Anik
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2231 (CRHC-91-21)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (*If applicable*) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research AMD, NCR, NSF, and NASA ICLAS
6c. ADDRESS (*City, State, and ZIP Code*) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (*City, State, and ZIP Code*) Arlington, VA Dayton, Ohio California Hampton VA Washington, DC
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014- 90-J-1270 NASA NAG 1-613
8c. ADDRESS (*City, State, and ZIP Code*) 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

Performance Implications of Synchronization Support for Parallel FORTRAN Programs (unclassifi

12. PERSONAL AUTHOR(S)

Anik, Sadun and Hwu, Wen-Mei

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1991 June 17	45

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	synchronization, shared memory, scheduling, parallel processing

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This paper studies the performance implications of architectural synchronization support for automatically parallelized numerical programs. As the basis for this work, we analyze the needs for synchronization in automatically parallelized numerical programs. The needs are due to task management, loop scheduling, barriers, and data dependency handling. We present synchronization algorithms for efficient execution of programs with nested parallel loops. Next, we identify how various hardware synchronization primitives can be used to satisfy these software synchronization needs. The synchronization primitives studied are *test & set*, *fetch & add*, *exchange-byte* and synchronization bus implementation of *lock/unlock* operations. Lastly, we ran experiments to quantify the impact of various architectural support on the performance of a bus-based shared memory multiprocessor running automatically parallelized numerical programs. We found that supporting an atomic *fetch & add* primitive in shared memory is as effective as supporting *lock/unlock* operations with a synchronization bus. Both achieve substantial performance improvement over the cases where atomic *test & set* and *exchange-byte* operations are supported in shared memory.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

Performance Implications of Synchronization Support for Parallel FORTRAN Programs

Sadun Anik and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing

Coordinated Science Laboratory

University of Illinois, Urbana-Champaign

1101 W. Springfield Ave.

Urbana, IL 61801

Correspondent: Wen-mei W. Hwu

Tel: (217) 244-8270

Email: hwu@crhc.uiuc.edu

Abstract

This paper studies the performance implications of architectural synchronization support for automatically parallelized numerical programs. As the basis for this work, we analyze the needs for synchronization in automatically parallelized numerical programs. The needs are due to task management, loop scheduling, barriers, and data dependency handling. We present synchronization algorithms for efficient execution of programs with nested parallel loops. Next, we identify how various hardware synchronization primitives can be used to satisfy these software synchronization needs. The synchronization primitives studied are *test&set*, *fetch&add*, *exchange-byte* and synchronization bus implementation of *lock/unlock* operations. Lastly, we ran experiments to quantify the impact of various architectural support on the performance of a bus-based shared memory multiprocessor running automatically parallelized numerical programs. We found that supporting an atomic *fetch&add* primitive in shared memory is as effective as supporting *lock/unlock* operations with a synchronization bus. Both achieve substantial performance improvement over the cases where atomic *test&set* and *exchange-byte* operations are supported in shared memory.

1 Introduction

Automatically parallelized numerical programs represent an important class of parallel applications in high-performance multiprocessors. These programs are used to solve problems in many engineering and science disciplines such as Civil Engineering, Mechanical Engineering, Electrical Engineering, Chemistry, Physics, and Life Sciences. In response to the popular

demand, parallelizing FORTRAN compilers have been developed for commercial and experimental multiprocessor systems to support these applications [2][12][1][8][11]. With maturing application and support software, the time has come to study the architecture support required to achieve high performance for these parallel programs.

Synchronization overhead has been recognized as an important source of performance degredation in the execution of parallel programs. Many hardware and software techniques have been proposed to reduce the synchronization cost in multiprocessor systems [13][20][19][3][14][15][16]. Instead of proposing new synchronization techniques, we address a simple question in this paper: does architecture support for synchronization substantially affect the performance of automatically parallelized numerical programs?

To answer this question, we start with analyzing the needs of synchronization in parallelized FORTRAN programs in Section 2. Due to the mechanical nature of parallelizing compilers, parallelism is expressed in only a few structured forms. This parallel programming style allows us to systematically cover all the synchronization needs in automatically parallelized programs. Synchronization issues arise in task management, loop scheduling, barriers and data dependence handling. A set of algorithms are presented which use generic lock()/unlock() and increment() operations. We then identify how several hardware synchronization primitives can be used to implement these generic synchronization operations. These synchronization primitives are *test&set*, *fetch&add*, *exchange-byte*, and *lock/unlock* operations . Since these primitives differ in functionality, the algorithms for synchronization in parallel programs are implemented with varying efficiency.

Section 3 describes the experimental procedure and the scope of our experiments. In Section 4, the issue of loop scheduling overhead is addressed in the context of hardware synchronization support. We present an anlytical model for the effect of loop scheduling overhead and loop granularity on execution time. Furthermore we measure loop scheduling overhead for different synchronization primitives with simulation.

Synchronization needs of a parallel application depend on the numerical algorithms and the effectiveness of the parallelization process, therefore the performance implications of architectural synchronization support can only be quantified with experimentation. Section 5 addresses the issues of granularity and lock locality in real applications. Using programs selected from the Perfect Club [6] benchmark set, we evaluate the impact of various architectural support on the performance of a bus-based shared-memory multiprocessor architecture in Section 6. We conclude that architectural support for synchronization has a profound impact on the performance of the benchmark programs.

Finally, the related work is presented in Section 7 and Section 8 includes the concluding remarks.

2 Background

In this section, we first describe how parallelism is expressed in parallel FORTRAN programs. We then analyze the synchronization needs in the execution of these programs. Most importantly, we show how architectural support for synchronization can affect their performance.

```
      DOALL 30 J=1,J1
      X(II1+J) = X(II1+J) * SC1
      Y(II1+J) = Y(II1+J) * SC1
      Z(II1+J) = Z(II1+J) * SC1
30    CONTINUE
```

Figure 1: A DOALL loop

2.1 Parallel FORTRAN Programs

The application programs used in this study are selected from the Perfect Club benchmark set [6]. The Perfect Club is a collection of numerical programs for benchmarking supercomputers. The programs were written in FORTRAN. For our experiments, they were parallelized by the Cedar source-to-source parallelizer [11] which generates a parallel FORTRAN dialect, Cedar FORTRAN. This process exploits parallelism at the loop level and loop level parallelization has been shown to capture most of the available parallelism for these programs [7]. Cedar FORTRAN has two major constructs to express loop level parallelism: DOALL and DOACROSS loops. A DOALL loop is a parallel DO loop where there is no dependence between the iterations. The iterations can be executed in parallel in arbitrary order. Figure 1 shows an example of a DOALL loop.

In a DOACROSS loop [9], there is a dependence relation across the iterations. A DOACROSS loop has the restriction that iteration i can only depend on iterations j where $j < i$. Because of this property, even a simple loop scheduling scheme can guarantee deadlock free allocation of DOACROSS loop iterations to processors. In Cedar FORTRAN, dependences between loop iterations are enforced by Advance/Await synchronization statements

[2]. An example of a DOACROSS loop is shown in Figure 2. The first argument of Advance and Await statements is the name of the synchronization variable to be used. The second argument of an Await statement is the data dependence distance in terms of iterations. In this example, when iteration i is executing this Await statement, it is waiting for iteration $i - 3$ to execute its Advance statement. The third argument of Await is used to enforce sequential consistency in Cedar architecture [11]. The third argument implies that upon the completion of synchronization, the value of `X(I-3)` should be read from shared memory. Similarly, the second argument of Advance statement implies that writing the value `X(I)` to shared memory should be completed before Advance statement is executed.

```
      DOACROSS 40 I=4,IL
      :
      AWAIT(1, 3, X(I-3))
      X(I) = Y(I) + X(I-3)
      ADVANCE (1, X(I))
      :
30    CONTINUE
```

Figure 2: A DOACROSS loop

2.2 Synchronization Needs

In executing parallel FORTRAN programs, the need for synchronization arises in four contexts: task management, loop scheduling, barrier synchronization, and Advance/Await. Task management is used for starting the execution of a parallel loop on multiple processors. In this study, task management is implemented by a global task queue. The processor which

```
put_task() {
    new_loop->number_of_processors = 0 ;
    new_loop->number_of_iterations = number of iterations in loop;
    new_loop->barrier_counter = 0 ;
    new_loop->iteration_counter = 0 ;
    lock(task_queue) ;
    insert_task_queue(new_loop) ;
    task_queue_status = NOT_EMPTY ;
    unlock(task_queue) ;
}
```

Figure 3: Producer algorithm for loop distribution

executes a DOALL or DOACROSS statement places the loop descriptor in the global task queue. All idle processors receive this loop descriptor and start the execution of the loop iterations. The accesses to the task queue by the processors are mutually exclusive. In our implementation, we use a task queue lock to enforce mutual exclusion. Figures 3 and 4 show the algorithms for the processor which executes the parallel DO statement and for the idle processors respectively. The removal of the loop descriptor from the task queue is addressed in the discussion of the barrier synchronization algorithm.

The implementation issues for the functions `lock()`, `unlock()`, and `increment()` with different primitives is presented in the next section. By definition `lock()` and `unlock()` operations are atomic. Whenever underlined in an algorithm, the `increment()` operation is also atomic and can be implemented with a sequence of lock, read-modify-write, unlock operations.

During the execution of a loop, each processor is assigned with different iterations. This is called loop scheduling. We used the processor self-scheduling algorithm [18] to implement

```
read_task() {
    while(task_queue_status = EMPTY) ;
    lock(task_queue) ;
    current_loop = read_task_queue_head() ;
            /* Doesn't remove the loop from the queue */
    increment(current_loop->number_of_processors) ;
    unlock(task_queue) ;
}
```

Figure 4: Consumer algorithm for loop distribution

```
schedule_iteration() {
    last_iteration = increment(current_loop->iteration_counter) ;
    if (last_iteration >= current_loop->number_of_iterations) {
        barrier synchronization ;
    }
    else {
        execute (last_iteration + 1)th iteration of loop;
    }
}
```

Figure 5: Self scheduling algorithm for loop iterations

loop scheduling. In processor self-scheduling, each processor executes the self-scheduling code before executing a parallel loop iteration. The self-scheduling algorithm shown in Figure 5 is executed at the beginning of each loop iteration and it uses an atomic increment operation on a shared counter. Unless the multiprocessor supports an atomic fetch&add operation, a lock is required to enforce the mutually exclusive accesses to the shared counter.

After all iterations of a loop are executed, processors synchronize at a barrier. For barrier synchronization, we used a non-blocking linear barrier algorithm which is implemented with

```
barrier_synchronization() {
    if (current_loop->barrier_counter == 0) {
        lock(task_queue) ;
        if (current_loop == read_task_queue_head()) {
            delete_task_queue_head() ;
            if (task_queue_empty() == TRUE) task_queue_status = EMPTY ;
        }
        unlock(task_queue) ;
    }
    if (increment(current_loop->barrier_counter) ==
            current_loop->number_of_processors - 1) {
        resume executing program from the end of this loop ;
    }
    else read_task() ;
}
```

Figure 6: Barrier synchronization algorithm

a shared counter (see Figure 6). After all iterations of a parallel loop have been executed, each processor reads and increments the barrier counter associated with the loop. The last processor to increment the counter completes the execution of the barrier. As in the case of loop self-scheduling, unless the multiprocessor system supports an atomic fetch&add operation, the mutually exclusive accesses to the shared counter are enforced by a lock.

In this algorithm, the first processor to enter the barrier removes the completed loop from the task queue. Using this barrier synchronization algorithm, the processors entering the barrier do not wait for the barrier exit signal and can start executing another parallel loop whose descriptor is in the task queue. In contrast to the compile time scheduling of "fuzzy barrier" [15], this algorithm allows dynamic scheduling of loops to the processors in a barrier.

In its presented form, the last processor to enter the barrier executes the *continuation* of the parallel loop — the code in the sequential FORTRAN program that is executed after all iterations of the current loop are completed[1].

This combination of task scheduling, iteration self scheduling and non-blocking barrier synchronization algorithms allows deadlock free execution of nested parallel loops with the restriction that DOACROSS loops appear only at the deepest nesting level [18].

The overhead associated with task management and barrier synchronization depends on the number of participating processors. When an P processor system is executing a parallel loop with N iterations, this task can be distributed to at most P processors. Therefore, at most P processors synchronize at a given barrier. Using processor self-scheduling, the N iterations are distributed to processors one at a time.

For the last type of synchronization, the ADVANCE/AWAIT statements are implemented by a vector for each synchronization point. In executing a DOACROSS loop, iteration `i`, waiting for iteration `j` to reach synchronization point `synch_pt`, busy waits on location `V[synch_pt][j]`. Upon reaching point `synch_pt`, iteration `j` sets location `V[synch_pt][j]`. This implementation, as shown in Figure 7, uses regular memory read and write operations, thus does not require atomic synchronization primitives. This implementation assumes a coherent and sequentially consistent memory system. In the presence of a memory system with weak ordering, an AWAIT statement can be executed only after the previous memory write operations complete execution. For a multiprocessor with software controlled cahce

[1] By using a semaphore, the processor which executed the corresponding DOALL/DOACROSS statement can be made to wait for the barrier exit to execute the continuation of the loop.

```
initialization(synch_pt) {
    for (i = 1 ; i < number_of_iterations ; i++) V[synch_pt][i] = 0 ;
}

advance(synch_pt) {
    V[synch_pt][iteration_number] = 1 ;
}

await(synch_pt, dependence_distance) {
    if(iteration_number <= dependence_distance) return() ;
    else while (V[synch_pt][iteration_number - dependendence_distance] == 0) ;
}
```

Figure 7: Algorithm for ADVANCE/AWAIT operations

coherency protocols, Cedar FORTRAN ADVANCE/AWAIT statements include the list of variables whose values should be read from/written to shared memory before their execution. The implementation details of these statements in multiprocessors with weakly ordered memory models or software controlled cache coherency protocols are beyond the scope of this paper.

In a multiprocessor which does not support an atomic fetch&add operation, lock accesses play an important role in the execution of scheduling and barrier synchronization algorithms. The next section discusses the implementation issues of lock accesses in the presence of different synchronization primitives.

2.3 Locks and Hardware Synchronization Primitives

In executing numeric parallel programs, locks are frequently used in synchronization and scheduling operations. In the task scheduling algorithm (See Figures 3 and 4), the use of locks enforces mutually exclusive access of processors to the task queue. Locks are also used to ensure correct modification of shared counters when an atomic *fetch&add* operation is not supported by the architecture. Such shared counters are used both by loop scheduling (See Figure 5) and barrier synchronization (See Figure 6) algorithms.

There are several algorithms for implementing lock accesses in cache coherent multiprocessors using hardware synchronization primitives[3][14]. These algortihms have different dynamic characteristics. Consider the execution of a linear barrier, which is implemented by a shared barrier counter. In the case where all processors arrive at the barrier at the same time, a simple spin lock algorithm to enforce exclusive access to the counter will cause excessive bus traffic. This would slow down the execution of the barrier. However, in the best case, processors would arrive at the barrier at different times, causing no lock contention. In this case, the latency of a lock operation is important for the overhead in entering a barrier. An important tradeoff in lock algorithms is their performance under heavy load versus the latency of lock operations. In Section 4 we analyze the implications of using different lock algorithms on the performance of loop scheduling.

All existing multiprocessor architectures provide some hardware support for atomic synchronization operations. Functionally, any synchronization primitive can be used to satisfy the high level synchronization needs of a parallel program. In practice, different primitives

may result in very different performance levels. For example, a queuing lock algorithm [3][14] can be implemented efficiently with an *exchange-byte* or a *fetch&add* primitive but a *test&set* implementation is more complicated and may be inefficient.

The *exchange-byte* version of the queuing lock algorithm is shown in Figure 8. In this implementation, the *exchange-byte* primitive is used to construct a logical queue of processors which contend for a lock. The variable `my_id` is assumed to be set at the start of the program such that it is value for the ith processor is i. The variable `queue_tail` holds the I.D. of the last processor which tried to acquire this lock. A processor which tries to access the lock receives the I.D. of the processor which preceded it and writes its own I.D. into the variable `queue_tail`. This algorithm constructs a queue of processors waiting for a lock where each processor waits only on its predecessor for the release of the lock. By mapping the elements of synchronization vector `flags[]` to non-conflicting cache lines, the memory accesses in the `while` loop of this algorithm can be confined to individual caches of processors. When a processor releases the lock, only the cache line read by its successor in the queue is invalidated.

In implementing the queuing lock algorithm with the *test&set* primitive, because of the functional limitations of this primitive, the queue of processors contending for a lock can not be constructed with a single atomic operation. This introduces a critical section into the implementation of queuing operation, which requires the use of an auxiliary lock. When *test&set* is used to emulate the *exchange-byte* primitive in the algorithm in Figure 8, the queuing operation becomes the sequence of operations

```
initialization() {
    flags[0] = FREE ;
    flags[1...P] = BUSY ;
    queue_tail = 0 ;
}

lock() {
    queue_last = exchange-byte(my_id, queue_tail) ;
    while(flags[queue_last] == BUSY) ;
    flags[queue_last] = BUSY ;
}
unlock() {
    flags[my_id] = FREE ;
}
```

Figure 8: Queuing lock algorithm for lock accesses

lock(auxilary_lock) ; read(queue_tail) ; write(my_id) ; unlock(auxilary_lock).

Clearly, the lock operations used in the implementation of the queuing lock algorithm need to be implemented with a different algorithm like *test&test&set* (see Figure 9).

In the synchronization algorithms presented in Section 2.2, most of the lock operations are used to implement atomic increment operations. The critical section involved in an atomic increment operation consists of one memory-read, one addition, and one memory-write instruction, which is similar to the critical section used in emulating an exchange-byte operation (one memory-read and one memory-write instruction). Therefore, the overhead of constructing the lock queue in the *test&set* implementation of a queuing lock would be similar to the overhead in using the *test&test&set* algorithm to implement lock operations for accessing shared counters. Therefore, whenever the architecture supports only the *test&set*

```
lock() {
    while(lock == BUSY || test&set(lock) == BUSY) ;
}

unlock() {
    lock = CLEAR ;
}
```

Figure 9: Test&test&set algorithm for lock accesses

primitive, a plain *test&test&set* algorithm is used in this study to implement all lock operations [2]. For an architecture with the *exchange-byte* primitive, the queuing lock algorithm is used for lock operations.

Due to the emphasis on atomic increment operations, supporting a *fetch&add* operation in hardware can significantly decrease the need for lock accesses in the synchronization algorithms. When *fetch&add* is the main synchronization primitive of a system, we used a *fetch&add* implementation of *test&test&set* algorithm to support the lock/unlock operations in task management. The performance implications of supporting a *fetch&add* primitive on loop scheduling algorithm will be presented in Section 4.

The functionality and sophistication of hardware synchronization support increase the cost of a system. In the Alliant FX/8, a separate synchronization bus and a Concurrency Control Unit is provided [2] and which can improve parallel program performance by reducing the and latency of lock accesses and the memory contention caused by them. Therefore in

[2]However, We would like to point out that in an environment where critical sections of algorithms involve many instructions and memory accesses, a *test&set* implementation of a queuing lock may enhance performance.

our analysis in Section 4, we also consider the case where a synchronization bus is used to implement lock operations. Finally, the cost performance tradeoffs for synchronization support can only be decided by evaluating the performance implications of different schemes for real parallel applications. These experiments are presented in Section 6.

3 Experimental Method

To evaluate the performance of several parallel processor architectures, we used a high-level trace driven simulator. In our approach, we used an abstract model of parallel program execution which presents a simplified view of the application program while allowing a detailed evaluation of synchronization support.

Execution of a sequential program on one of the processors of a shared memory MIMD machine can be modeled by partitioning the program execution time into two sections: the execution that is local to the processor (execution of instructions in the CPU and memory accesses to the local cache) and the time spent in handling memory requests to the shared memory. For computationally intensive numeric applications where I/O and system calls account for a small fraction of execution time, this simplistic model can be used to approximate sequential program execution time.

Using a RISC based processor model where instruction execution times are defined by the architecture, the local execution time of a program can be calculated directly from its dynamic instruction count. On the other hand, the time to service the accesses to shared memory depends not only on the data and instruction access patterns of the local processor

but also on the activities of other processors in the system.

Extending this model to parallel processing, a parallel FORTRAN program can be modeled as a collection of program segments, that we will call *task-pieces*, where each task-piece is a sequential part of the application which executes on a single processor. The dependences among task-pieces are enforced by *events*. The two special events program_start and program_end mark the beginning and the completion of a program. With this model, the execution of a sequential program consists of a program_start event followed by a task-piece which covers the whole of the program execution, and a program_end event. For parallel FORTRAN programs, the additional events are: execution of DOALL and DOACROSS statements, beginning and end of parallel loop iterations, barriers, and execution of Advance/Await statements.

A *high level trace* is the record of events that took place during execution and the information about task-pieces executed between pairs of events. In this study, the high level traces are collected by manual source code instrumentation of parallelized applications. In the trace, each event is identified by its type (DOALL, iteration start, barrier etc.) and applicable arguments (e.g., the synchronization point and the iteration number for an Await event). The task pieces are represented by the dynamic count of read and write accesses to shared data and the approximate number of dynamic instructions executed.

An event driven, bus based shared memory multiprocessor simulator is used to calculate the program execution time from a high level trace. The simulator implements the task management and synchronization algorithms for different synchronization primitives by using

Table 1: Relative timing parameters for simulations

component/operation	cycle time
processor (base)	1
memory bus	1
memory module	3
test&set	6
exchange-byte	6
fetch&add	6
lock/unlock (synchronization bus)	1
fetch&add (synchronization bus)	1

the algorithms described in Section2. In our experiments, the atomic operations *test&set*, *exchange-byte* and *fetch&add* are implemented in shared memory. The processor memory interconnection is a decoupled access bus whose cycle time is equal to the processor cycle time. We assumed that shared memory is 8-way interleaved where an access to a module takes 3 bus cycles. Atomic operations that are implemented in memory take two memory cycles (e.g. a *test&set* operation takes 6 bus cycles to execute in memory). When support of a synchronization bus is evaluated, a single cycle access synchronization bus model is used. A summary of the timings parameters is shown in Table 1.

In the simulation model, an invalidation based write-back cache coherence scheme is used. The atomic operations implemented in shared memory are assumed to be write-through and result in a single word bus transaction. This allows caching of synchronization variables (a necessity for efficiency of spin-locks) with reduced bus traffic for atomic operations. The event in a parallel program are simulated at the level of individual bus transactions, taking into account the contention at the bus and memory module access conflicts.

Table 2: Assumptions for memory traffic

parameter	value
memory/instruction ratio	0.20
shared data cache miss rate	0.80
non-shared data cache miss rate	0.05

The other component of shared memory traffic is the data needs of task-pieces. The shared memory traffic contributed by the application is modelled based on the measured instruction count and frequency of shared data accesses. Table 2 lists the assumptions used to simulate the memory traffic for the task-pieces. We assume that 20% of the instructions executed are memory references. In addition, we measured that 6-8% of all instructions (approximately 35% of all memory references) are to shared data. We assume that references to shared data cause the majority of cache misses (80% shared data cache miss rate and 5% non-shared data cache miss rate)[3].

The bus transactions due to the cache misses of the task pieces are combined with those contributed by the servicing of *events* to simulate the overall system behavior. The simulation is performed on a cycle-by-cycle basis.

4 Analysis of loop scheduling overhead

In the execution of a parallel loop, the effect of loop scheduling overhead on performance depends on the number of processors, total number of iterations, and the size of an iteration.

[3]When we repeated the experiments by lowering the shared cache miss rate to 40%, the speedup figures reported in Section 5 changed by less than 2%.

In this section we will first derive the expressions for speedup in executing parallel loops where the loop iterations are large (coarse granularity) and where the loops iterations are small (fine granularity). These expressions provide an insight to how loop scheduling overhead influences loop execution time, and will be used in analysis of simulation results later in this section.

Consider a DOALL loop with N iterations where each iteration, without any parallel processing overhead, takes t_l time to execute. For a given synchronization primitive and lock algorithm, let t_{sch} be the time it takes for a processor to schedule an iteration. We will look at the impact of scheduling overhead for two cases. For the first case we assume that when a processor is scheduling an iteration, it is the only processor doing so. When the accesses to the shared counter in loop scheduling algorithm are implemented with a lock, t_{sch} can be written as

$$t_{sch} = t_{lock} + t_{update} + t_{unlock}$$

where t_{lock} and t_{unlock} are the time it takes to acquire and release a lock respectively, and t_{update} is the time for reading and incrementing the shared counter. The execution of several iterations of a loop for this case is shown in Figure 10. In the figure, when P_1 completes the execution of the first iteration, it schedules the next iteration without delay.

For any given P and t_{sch}, the necessary condition for this case is

$$t_l > (P-1) \times t_{sch},$$

and the time to execute the loop with P processors can be written as

$$t_P = ((t_{sch} + t_l) \times \lceil N/P \rceil) + t_{oh},$$

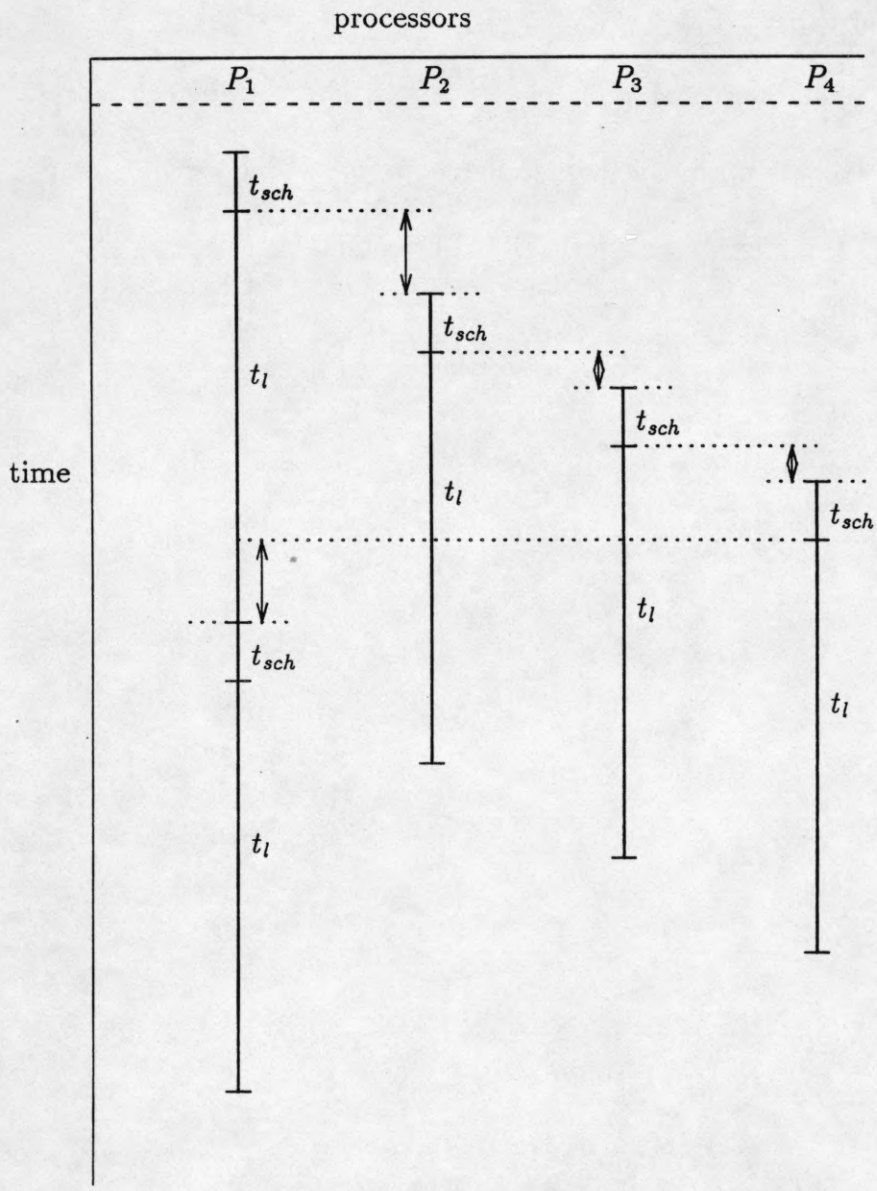

Figure 10: Scheduling of iterations for Case 1

where t_{oh} is the total task scheduling and barrier synchronization overhead per processor. Since the task scheduling and barrier synchronization overhead depends only on the number of processors, t_{oh}, is constant for a given P.

The execution time of the sequential version of this loop, t_{seq}, is $t_l \times N$ which is *not* equal to t_1 — single processor execution time of the parallel loop. We define *speedup* for P processors as the ratio of t_{seq} to t_P. The speedup for a DOALL loop is

$$\begin{aligned} speedup &= \frac{t_{seq}}{t_P} \\ &= \frac{t_l N}{((t_{sch} + t_l) \times \lceil N/P \rceil) + t_{oh}} \\ &\approx \frac{P}{\frac{t_{sch}+t_l}{t_l} + \frac{P \times t_{oh}}{N \times t_l}} \end{aligned}$$

for $N \gg P$

$$speedup \approx P \times \frac{t_l}{t_{sch} + t_l}$$

using $t_l > (P-1) \times t_{sch}$

$$\begin{aligned} speedup &> P \times \frac{t_l}{\frac{t_l}{P-1} + t_l} \\ &> P \times \frac{P-1}{P} \\ &> P - 1 \end{aligned}$$

Therefore, when $t_l > (P-1) \times t_{sch}$, the speedup is linear with number of processors hence the execution time depends only on P and the total amount of work in the loop, $N \times t_l$.

Now let us consider the case where a processor completing the execution of an iteration *always* has to wait to schedule the next iteration because of another processor scheduling an iteration at that time. We will call the scheduling overhead for this case t'_{sch}. This scenario is illustrated in Figure 11. In this figure after the completion of the first iteration, P_1 waits completion of the scheduling operations by other processors before scheduling an iteration. In general, t'_{sch} is different from t_{sch} because of the contention caused by several processors trying to schedule an iteration. The necessary condition for this case is

$$t_l < (P-1) \times t'_{sch},$$

and the loop scheduling overhead forms the critical path in determining the loop execution time. When loop scheduling becomes the bottleneck, execution time is:

$$t_P = N \times t'_{sch} + t_l,$$

for $N \gg P$

$$t_P \approx N \times t'_{sch}.$$

When the loop scheduling algorithm is implemented with lock operations, scheduling an iteration involves transferring the ownership of the lock from one processor to the next, and reading and incrementing the shared counter. Therefore

$$t'_{sch} = t_{lock-transfer} + t_{update}.$$

In the remainder of this section we will first look at how loop execution time varies with loop granularity. Since t'_{sch} directly influences the execution time of a loop, we will then

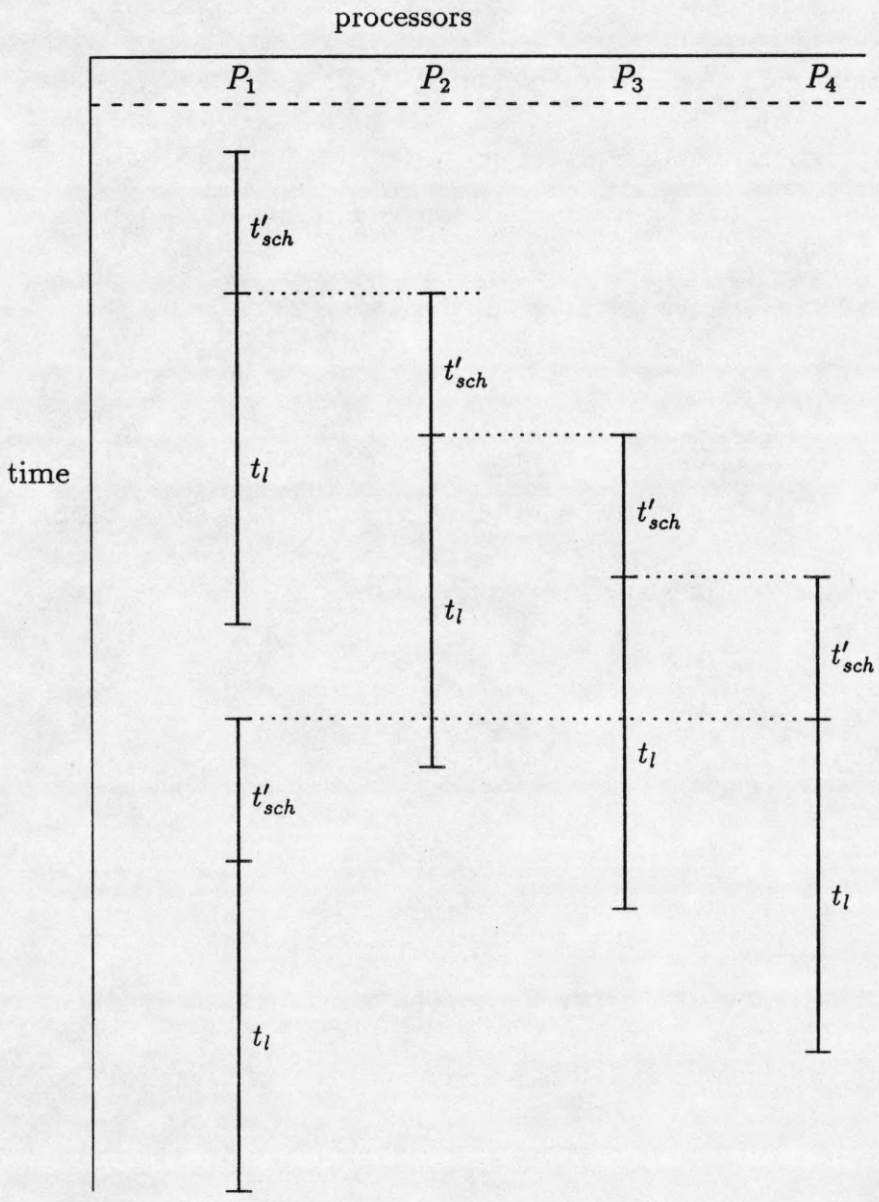

Figure 11: Scheduling of iterations for Case 2

measure this loop scheduling overhead for different hardware synchronization primitives using our simulation tool.

4.1 Granularity effects

The analysis above shows the existence of two different types of behavior of execution time for a parallel loop. Given a multiprocessor system, the parameters P, t_{sch} and t'_{sch} do not change from one loop to another. Keeping these parameters constant, the granularity of a loop, t_l, determines whether scheduling overhead will be significant in overall execution time or not.

The architectural support for synchronization primitives influences the execution time of a parallel loop in two ways. On one hand, different values of t'_{sch} for different primitives result in different execution time when the loop iterations are small (i.e., fine granularity loops). On the other hand t_{sch} determines whether a loop is of fine or coarse granularity. In this section we present the simulation results on how loop execution time varies across different implementations of the loop scheduling algorithm. Since t'_{sch} determines the execution time of a fine granularity loops, we quantify how t'_{sch} changes with synchronization primitives used, and the number of processors in the system. In our simulations, the memory access characteristics of the loops were modelled as presented in Table 2.

Figures 12-15 show the simulation results for execution time vs. the size of an iteration in a DOALL loop. The loop sizes are in terms of the number of instructions, and the execution time in terms of CPU cycles. In these simulations, the total amount of instructions in the

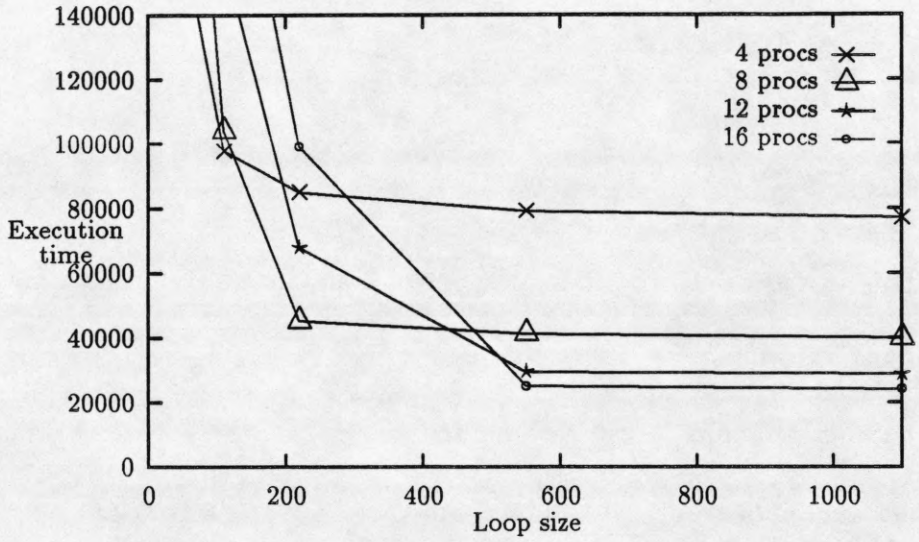

Figure 12: Execution time vs. granularity for test&set primitive

loop is kept constant while changing the number of instructions in an iteration. It can be seen in these figures that when loop iterations are very large, the execution time of a loop on a given number of processors is that same for different synchronization primitives. There is also a monotonic increase in execution time as loop size gets smaller in all cases.

Figure 12 shows that for 16 processors and using test&set primitive, there is a sharp increase in execution time when iteration size is less than 550 instructions. This number is around 300 for exchange-byte, and 200 for a synchronization bus, see Figures 13 and 14. As shown in Figure 15, using the fetch&add primitive, the iteration size where execution time starts increasing is around 100 instructions. We observed that in the FORTRAN programs we used in our experiments the iteration sizes of the parallel loops vary from 20 to 1000 instructions. This shows that the choice of a synchronization primitive will influence the perrformance of some loops. The distribution of instructions in the dynamic execution

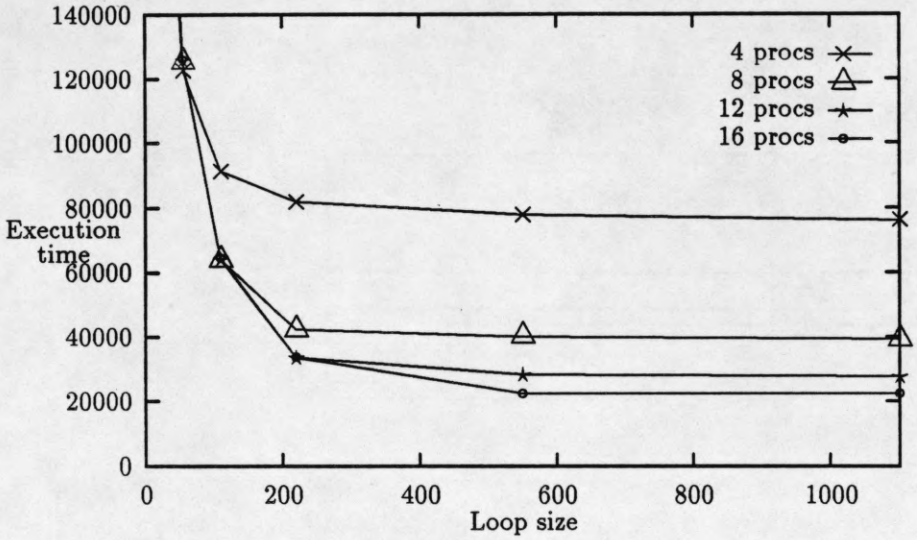

Figure 13: Execution time vs. granularity for exchange-byte primitive

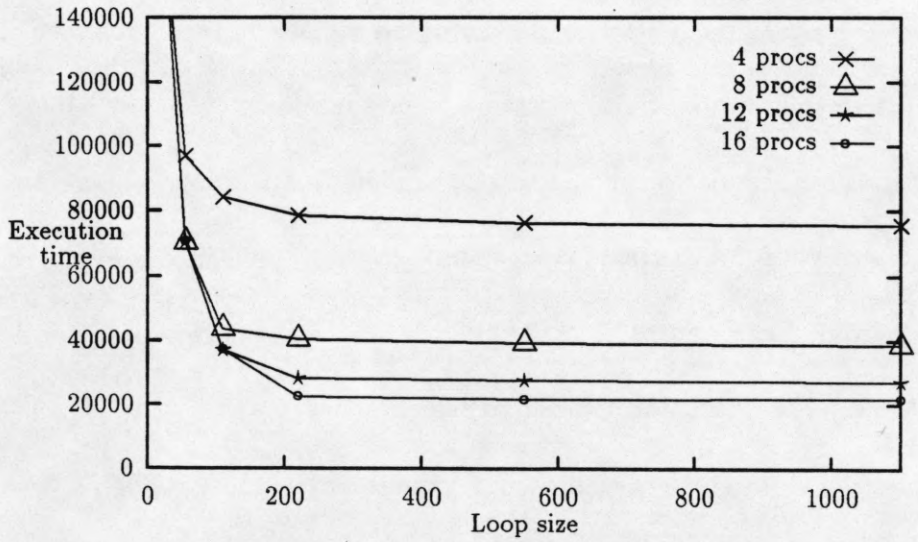

Figure 14: Execution time vs. granularity for synchronization bus

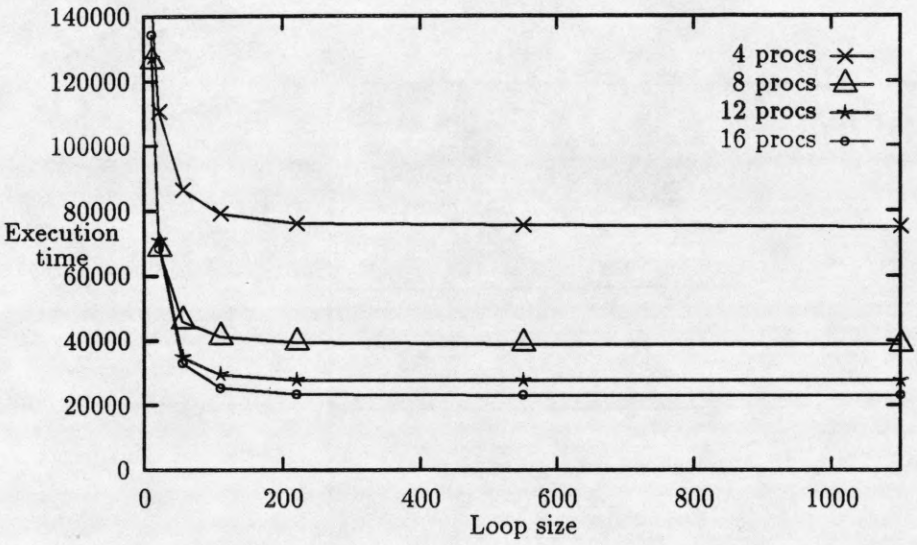

Figure 15: Execution time vs. granularity for fetch&add primitive

traces with respect to loop granularity for the application programs is presented in Section 5.

4.2 Scheduling overhead for fine grain loops

For fine grain loops, the loop execution time T_P is approximated by $N \times t'_{sch}$. The change of execution time with respect to the number of iterations of a loop is shown in Figures 16-19. The synthetic loops used in these simulations has a total of 220000 instructions. Therefore, the region where iteration size < 50 instructions corresponds to $N > 4400$ in these figures. The common observation from these figures is that when loop iterations are sufficiently small (N is sufficiently large), the execution time increases linearly with N. Also, when extrapolated, T_P vs. N lines go through the origin which validates the linear model

$$T_P = N \times t'_{sch}$$

for execution time.

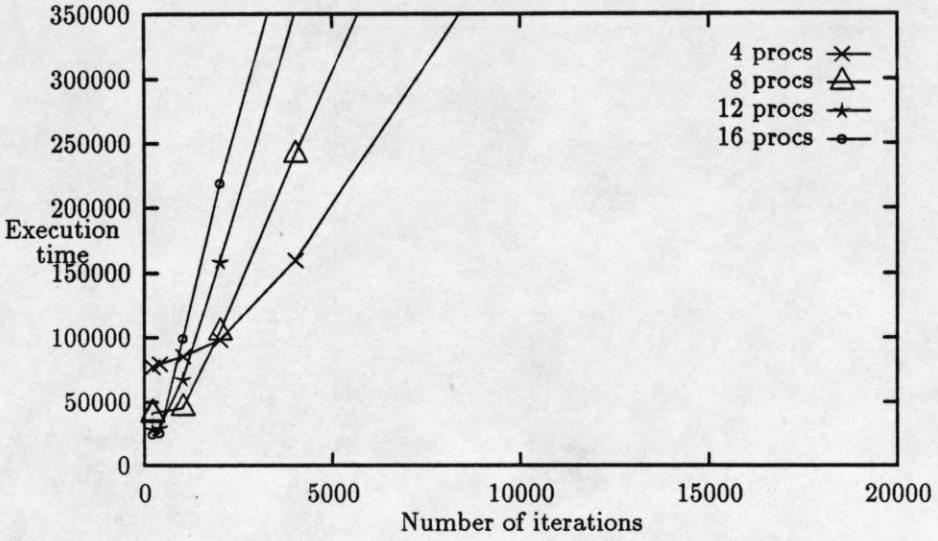

Figure 16: Execution time vs. number of iterations for test&set primitive

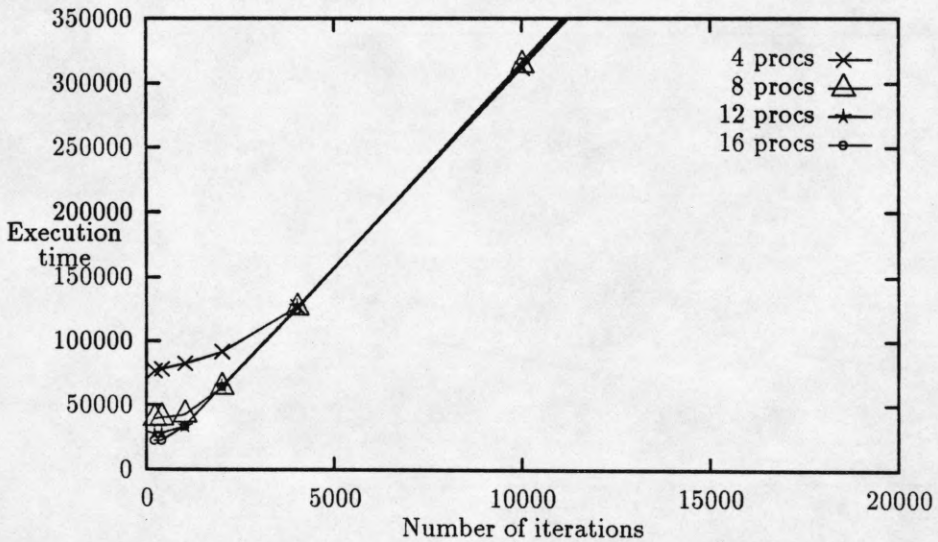

Figure 17: Execution time vs. number of iterations for exchange-byte primitive

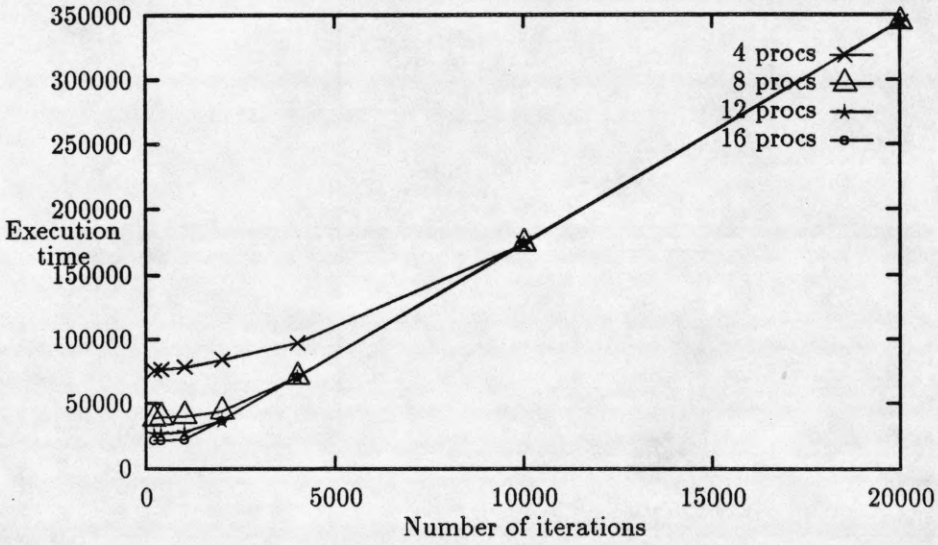

Figure 18: Execution time vs. number of iterations for synchronization bus

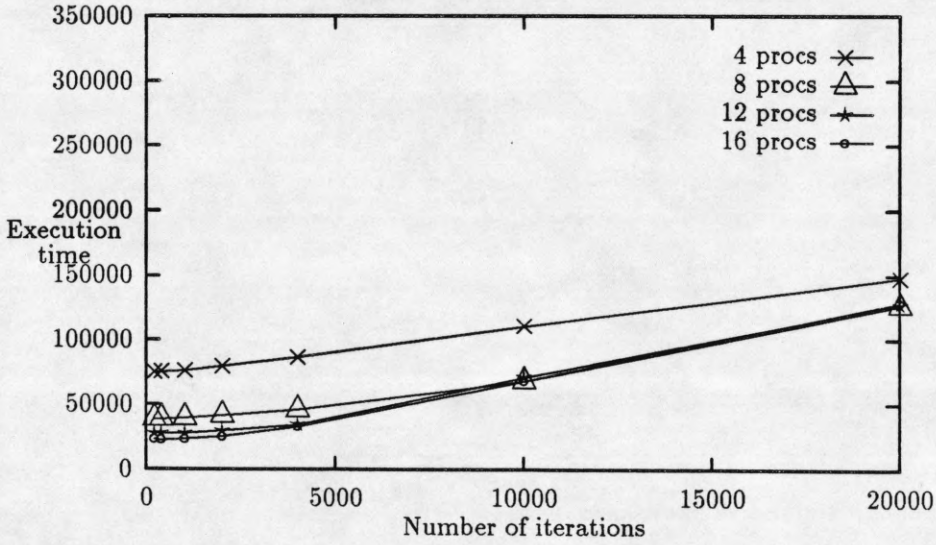

Figure 19: Execution time vs. number of iterations for fetch&add primitive

Figure 20 shows how scheduling overhead per iteration, t'_{sch}, changes for the different synchronization primitives as the number of processors increases.

Using the test&set primitive, the scheduling overhead increases with number of processors. For the exchange-byte and fetch&add primitives and the synchronization bus, the scheduling overhead scales well. Furthermore t'_{sch} shows great variance across primitives. For the 16 processor case the average number of cycles to schedule a loop iteration are 98, 31, 17 and 7 cycles for test&set, exchange-byte, synchronization bus, and fetch&add primitives respectively.

The synchronization bus model used in these simulations has single cycle access time for free locks and single cycle lock transfer time. Therefore the synchronization bus data shows the highest performance achievable by hardware support for lock accesses alone. In Section 6, the performance figures for a synchronization bus which also supports single cycle fetch&add operation are given. Such a synchronization bus is capable of scheduling a loop iteration every clock cycle. Therefore its overall performance can be expected to be better than all the primitives analysed in this section.

5 Synchronization Characteristics of Applications

5.1 Loop granularity of application programs

The two applications we used in this study are BDNA and FLO52. BDNA is a molecular dynamics simulator for biomolecules in water and it uses ordinary differential equation solvers.

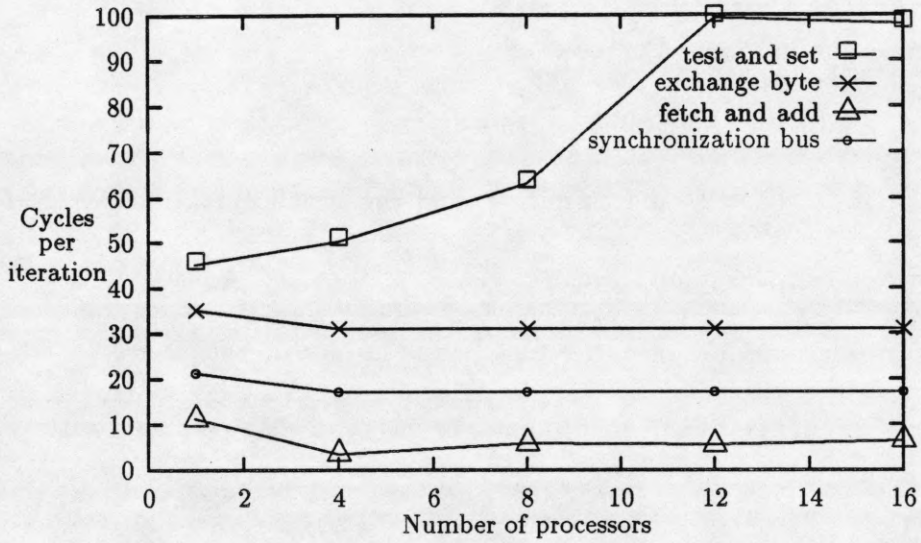

Figure 20: Loop scheduling overhead vs. number of processors

FLO52 is a fluid dynamics program which uses multigrid schemes and ordinary differential equation solvers. Both programs are vectorizable.

These programs have different parallelism structures and loop granularity. In the BDNA program, most of the parallel loops are not nested and the iterations are 200-1000 instructions long. Two thirds of all parallel loops in the trace are DOACROSS loops. In the FLO52 program, most of the parallelism exists in the form of nested DOALL loops. The size of the innermost loop iterations varies between 20-250 instructions.

The cumulative distribution of instructions executed with respect to the iteration size of parallel loops is shown in Figure 21 for both programs. Because the analysis of loop scheduling showed that execution of loops with iterations larger than 500 instructions do not suffer from scheduling overhead, only the loops with iterations less than 500 instructions are of concern. The BDNA curve in Figure 21 shows that for this program only 17% of all

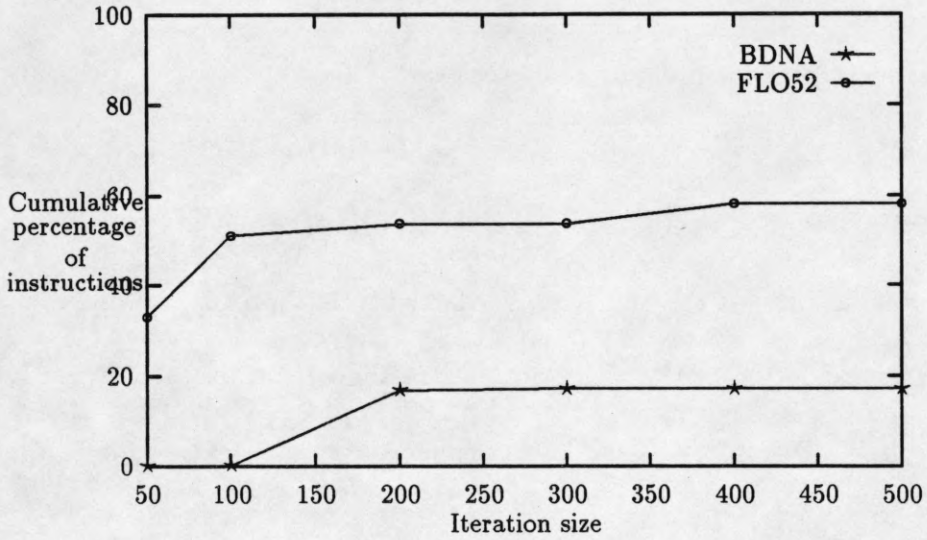

Figure 21: Cumulative distribution of dynamic instructions for loop iteration size

instructions were executed in loops with iteration size less than 500 instructions. On the other hand, more than half of the total computation in FLO52 program is done in loops where iterations have less than 100 instructions. From the analysis and simulation results in the previous section, we can expect the performance of FLO52 program be limited by loop scheduling overhead.

5.2 Locality of lock accesses in synchronization algorithms

In our simulations, we observed that both programs exhibit very low locality for lock accesses. When a processor acquires a lock, we consider it a *lock hit* if the processor which released the lock last is the same processor. Otherwise, acquiring a lock is said to result in a *lock miss*. The measured lock hit rate for the two programs with 4 or more processors was less than 0.2%. Such a low value of lock locality can be explained by the dynamic behavior of

scheduling and synchronization algorithms.

For each parallel loop, every processor acquires the task queue lock and barrier lock only once. This results in a round-robin style accesses to these locks. For the same parallel loop, the loop counter lock used in the loop self-scheduling algorithm is accessed multiple times by each processor. However, a lock hit can occur only when, the last processor which got an iteration number finishes execution of that iteration before the executions of previously scheduled iterations complete. Due to the very low variance of iteration size among the iterations of a parallel loop in these programs, this scenario is unlikely.

In the experiments, because of the low lock hit rate, the atomic memory operations are implemented in shared memory. An implementation of atomic operations in caches or processors would result in excessive invalidation traffic, and would also increase the latency of atomic operations.

6 Experimental Results

In this section we present the experimental results for two programs from the Perfect Club benchmark set: BDNA and FLO52. Speedup of parallel programs with respect to the execution time of the sequential version of the programs is used as the performance metric. There is no parallel processing overhead in the sequential version. In Section 6.1 the issues of lock contention and lock access latency are discussed. For this analysis we use the test&set atomic operation to implement the test&test&set algorithm and the exchange-byte atomic operation to implement the queuing lock algorithm. In Section 6.2, we present the

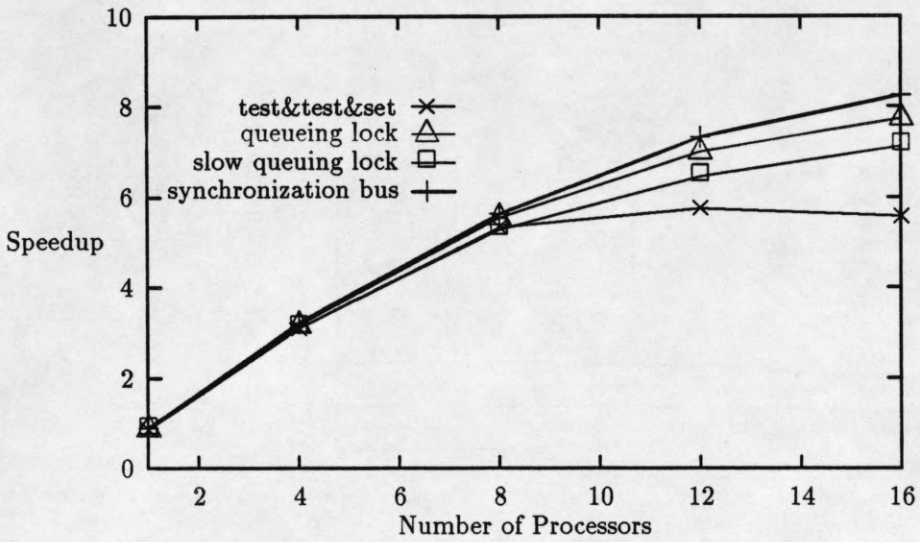

Figure 22: Lock latency and contention effects on BDNA program

performance implications of more sophisticated synchronization support.

6.1 Lock contention and latency

The first issue we focus on is lock contention. In the first set of simulations, we used the test&test&set algorithm to implement lock accesses. The speedup obtained from the two programs are shown in Figure 22 and Figure 23. In the BDNA program (Figure 22), using the test&test&set algorithm, the peak speedup of 5.75 is reached with 12 processors. In FLO52 (Figure 23), a speedup of 1.14 is obtained with 4 processors. Increasing the number of processors beyond 4 makes the performance worse.

The speedup for the single processor case is 0.86 for BDNA and 0.54 for FLO52. This data shows that, running on a single processor, FLO52 spends almost half of its execution time in scheduling and synchronization algorithms.

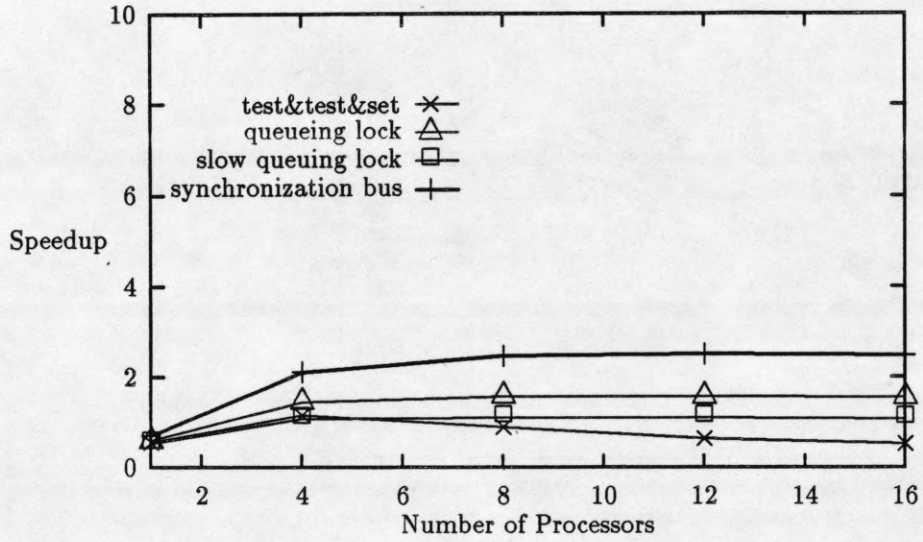

Figure 23: Lock latency and contention effects on FLO52 program

In the second set of experiments, we used a queuing lock algorithm for lock accesses to observe the effect of decreasing lock contention. As shown in Figures 22 and 23, the queuing lock significantly increases the performance of both programs when the number of processors is large. With 16 processors, the speedup improves by 50% in BDNA and 200% in FLO52. These results show that controlling lock contention with algorithms such as queuing lock does increase program performance.

The next issue we looked at was the importance of lock access latency. The effect of doubling the lock access latency in a queuing lock is shown in Figures 22 and 23 (slow queuing lock). For the 16 processor case, doubling the lock access latency decreases the speed up by 10% for BDNA and by 50% for FLO52.

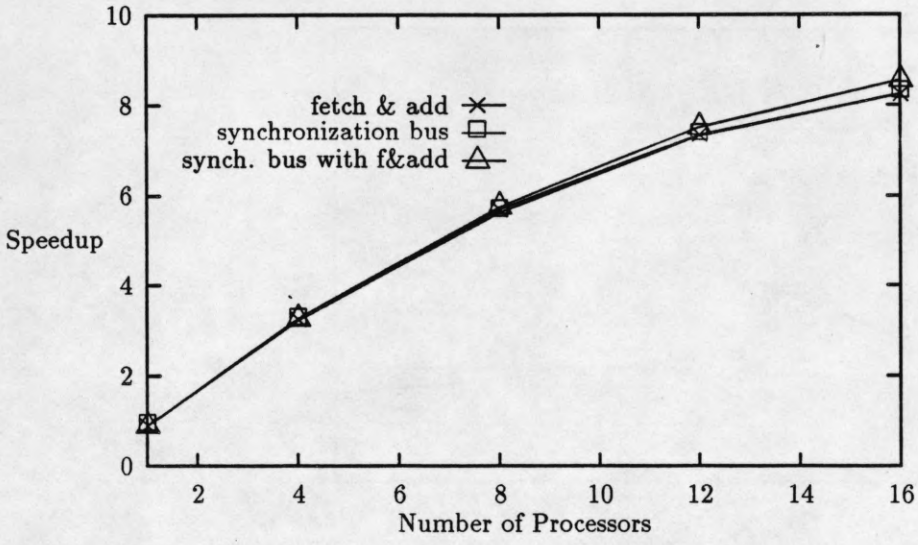

Figure 24: Performance of BDNA program with the use of synchronization bus and fetch&add primitive

6.2 Efficient architectural support for synchronization

In the previous section, we pointed out the importance of efficient lock operations. The next issue is the effect of using a synchronization bus for lock operations on program performance. A synchronization bus allows execution of synchronization operations with minimal latency and isolates synchronization traffic from memory traffic.

As shown in Figure 22, the speedup obtained from BDNA using a synchronization bus is 8.25 for the 16 processor case. The improvement in speedup over the queuing lock case (see Figure 22) ranges from 2% to 7% when the number of processors changes from 8 to 16. For the FLO52 program, from Figure 23 it can be seen that using a synchronization bus has a dramatic effect on program execution time. The speedup obtained for 8 or more processors is 2.5 which is 60% higher than the speedup in the case of a queuing lock.

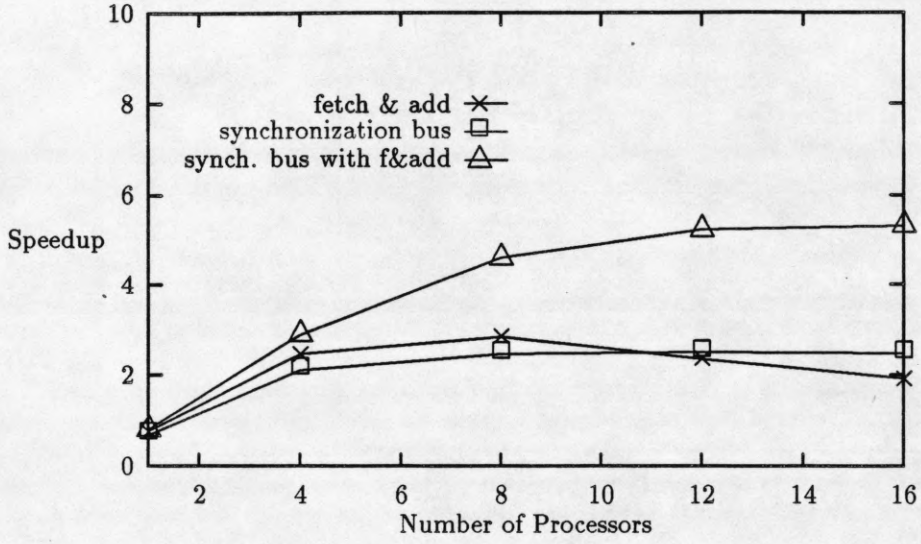

Figure 25: Performance of FLO52 program with the use of synchronization bus and fetch&add primitive

As discussed in Section 2.3, a significant number of the lock accesses required to increment the shared counters can be eliminated with the support of a fetch&add primitive in hardware. In the next experiment, the fetch&add operation (implemented in shared memory) was used for incrementing shared counters in loop self-scheduling and barrier synchronization algorithms. The results of these simulations are shown in Figures 24 and 25 for programs BDNA and FLO52 respectively.

In the BDNA program, the speedup obtained by using a fetch&add primitive implemented in hardware is the same as the speedup obtained by using a synchronization bus for lock accesses. In FLO52, a similar behavior is observed. Therefore, the performance benefits of a dedicated synchronization bus for lock accesses can be achieved at a lower cost by implementing an atomic fetch&operation in shared memory.

Finally, we considered the case where a synchronization bus is used to implement both

lock accesses and fetch&add operation. As shown in Figure 24, this resulted in a marginal increase in performance for the BDNA program. However, for the synchronization bound program FLO52, Figure 25 shows that the performance increase is in excess of 100%, reaching 5.3 for 16 processors.

7 Related Work

There has been considerable attention paid to the synchronization problem for multiprocessors. Brooks proposed the Butterfly Barrier [16] which does not have the *hot spots* observed in linear barriers. Gupta's "Fuzzy Barrier" improves processor utilization by allowing processors to do useful work in a barrier as a result of compile time analysis. The barrier algorithm we presented overlaps barrier execution with useful work by exploiting parallelism in nested parallel loops. An analytical analysis of different barrier synchronization algorithms were made by Arenstrof and Jordan [4]. Beckmann and Polychronopoulos studied the effect of barrier synchronization and scheduling overhead and presented a similar analytical formulation for execution time characterization for loop scheduling bound execution [5]. They used synthetic parallel loops as workload in their experiments. Polychronopoulos also developed guided self-scheduling scheme for loop scheduling [17]. In the future, we plan to evaluate performance implications of alternative loop scheduling and barrier synchronization algorithms for parallel scientific applications.

Finally, another experimental study on synchronization in real parallel applications was done previously by Davis and Hennesey [10]. Their work concentrated on how program

characteristics change synchronization behavior. For their class of applications, they concluded that implementation of synchronization operations have little effect on program performance.

8 Concluding Remarks

In this paper we demonstrated the feasibility of concentrating on one aspect of parallel program execution within the perspective of the overall program performance. We analyzed the performance implications of synchronization support for FORTRAN programs parallelized by a state-of-the-art compiler. In these programs, parallelism was exploited at loop level where the granularity of loops showed large variance across applications. We addressed the task management and synchronization issues that arise in executing these programs at different levels of abstraction. We presented dynamic task management and barrier synchronization algorithms for efficient execution of programs with nested parallel loops. The issues in implementation of the atomic lock access and counter increment operations that are used in loop self-scheduling, task management and barrier synchronization were addressed at the level of hardware synchronization primitives.

In the execution of parallel FORTRAN programs, we focused on loop scheduling overhead as the potential cause of performance degradation. Loop scheduling overhead was shown to determine execution time for fine granularity loops and to vary significantly with the architectural synchronization support. The synchronization algorithms used in executing these programs depend heavily on shared counters. In accessing shared counters, we concluded

that lock algorithms which reduce bus contention do enhance performance. For the class of applications we looked at, due to the importance placed on shared counters, a hardware implementation of a fetch&add primitive in shared memory can be as effective as a special synchronization bus which handles lock accesses.

We observed that in the execution of parallel FORTRAN programs, there is a very low locality of lock accesses. This implies that implementing atomic operations in private caches or in processors rather than in shared memory will result in loss of performance and additional memory traffic.

The simulation results with real program traces showed that while for an application with fine granularity loops the execution time showed large variance across synchronization primitives, performance of an application with coarse granularity is less sensitive to the particulars of hardware synchronization support. The simulation results with real parallel application traces showed that the choice of the hardware synchronization primitive in a shared-memory multiprocessor does have a significant effect on overall program performance.

Our assumptions on system architecture were targeted to increase the throughput of synchronization operations by implementing them in shared memory, and to decrease the interaction between regular memory operations and synchronization primitives by using split phase transactions. On architectures with longer memory access latency or where atomic operations consume more shared memory bus bandwith, we expect to see a more severe performance degredation due to synchronization overhead.

Acknowledgements

This research has been supported by the, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

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

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

[3] N. S. Arenstorf and H. F. Jordan. Comparing barrier algorithms. *Parallel Computing*, No. 12:157–170, 1989.

[4] *Balance(tm) 8000 Guide to Parallel Programming*, 1003-40425 rev. a edition, July 1985.

[5] C. J. Beckmann and C. D. Polychronopoulos. The effect of barrier synchronization and scheduling overhead on parallel loops. *Proceedings of the 1989 International Conference on Parallel Processing*, 2:200–204.

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

[7] D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity on parallel systems. Technical Report CSRD Rpt. No. 942, Center for Supercomputing Research and Development, University of Illinois, 1989.

[8] Cray Research Inc. *CRAY XM-P Multitasking Programmer's Reference Manual*, publication sr-0222 edition, 1987.

[9] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–845, 1986.

[10] H. Davis and J. Hennessy. Characterizing the synchronization behavior of parallel programs. *Proceedings of PPEALS*, pages 198–211, 1988.

[11] P. A. Emrath, D. A. Padua, and P. Yew. Cedar architecture and its software. *Proceedings of Twentysecond Hawaii International Conference on System Sciences*, 1:306–315, 1989.

[12] Encore. *Multimax Technical Summary*, January 1989.

[13] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *Proceedings of ASPLOS*, pages 64–75, 1989.

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

[15] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. *Proceedings of ASPLOS*, pages 54–63, 1989.

[16] E. D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15, No. 4:295–307, 1986.

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

[18] P.Tang and P. Yew. Processor self-scheduling for multiple-nested parallel loops. In *Proceedings of International Conference on Parallel Processing*, pages 528–534, 1986.

[19] G. S. Sohi, J. E. Smith, and J. R. Goodman. Restricted fetch&ϕ operations for parallel processing. *Proceedings of the 16th International Symposium on Computer Architecture*, pages 410–416, 1989.

[20] C. Zhu and P. Yew. A scheme to enforce data dependence on large multiprocessor systems. *Transactions on Software Engineering*, SE-13, No. 6:726–739, June 1987.