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, USA
Tel: (217) 244-8270
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 scheduling, iteration scheduling, barriers, and data dependence handling. We present synchronization algorithms for efficient execution of programs with nested parallel loops. Next, we identify how various hardware synchronization support can be used to satisfy these software synchronization needs. The synchronization primitives studied are *test&set*, *fetch&add* and *exchange-byte* operations. In addition to these, synchronization bus implementation of *lock/unlock* and *fetch&add* operations are also considered. 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 [1][11][21][6][10]. 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 degradation in the execution of parallel programs. Many hardware and software techniques have been proposed to reduce the synchronization cost in multiprocessor systems [12][23][22][2][13][14][15]. 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 for 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 scheduling, iteration 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 iteration scheduling overhead is addressed in the context of hardware synchronization support. We use an analytical model for the effect of iteration scheduling overhead and loop granularity on execution time. The model is then used to explain the differences in the iteration scheduling overhead of different synchronization primitives for a simulated shared-memory multiprocessor.

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 [4] 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.

2 Background and Related Work

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 the implementation efficiency of scheduling and synchronization algorithms.

2.1 Parallel Fortran Programs

The application programs used in this study are selected from the Perfect Club benchmark set [4]. 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 KAP/Cedar source-to-source parallelizer [17][10] which generates a parallel Fortran dialect, Cedar Fortran. This process exploits parallelism at the loop level, which has been shown by Chen, Su, and Yew to capture

```
      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

most of the available parallelism for Perfect Club benchmark set programs [5]. They measured the instruction level parallelism by trace based data flow analysis and concluded that parallel loop structures sufficiently exploit this parallelism. However this assumes that all memory and control dependences can be resolved in the parallelization process. In practice, compile time analysis of dependences may not be successful due to complex array indexing and limited inter-procedural data-flow analysis.

Cedar Fortran has two major constructs to express loop level parallelism: DOALL loops 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 [8], 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, a simple iteration 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 [1]. 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 [10]. 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 needs for synchronization arise in four contexts: task scheduling, iteration scheduling, barrier synchronization, and Advance/Await. In this section, we discuss the nature of these synchronization needs.

Task scheduling is used to start the execution of a parallel loop on multiple processors. All processors to participate in the execution of a parallel loop, or task, must be informed that the loop is ready for execution. In this study, all experiments assume a task scheduling algorithm that uses a centralized task queue to assign tasks to processors. The processor which executes a DOALL or DOACROSS statement places the loop descriptor into the task queue. All idle processors acquire the loop descriptor from the task queue and start executing the loop iterations. The accesses to the task queue by the processors are mutually exclusive. A lock is used to enforce mutual exclusion.

A number of distributed task scheduling algorithms have been proposed in the past, Anderson, Lazowska, and Lewy [3] compared the performance of several algorithms in the context of thread managers. Most distributed task scheduling algorithms rely on a large supply of parallel tasks to maintain load balance. Also, they usually assume that each task needs to be executed by only one processor. These are valid assumptions for thread managers because there are usually a large number of tasks (threads) in their application programs and each task represents a piece of sequential code. These assumptions are, however, not valid for the current generation of automatically parallelized Fortran programs where parallelism is typically exploited at only one or two loop nest levels. Since all parallel iterations of a single loop nest level form a task, there is typically only a very small number of tasks in the task queue. Also, multiple processors need to acquire the same task so that they can work on different iterations of the task loop. This lack of task level parallelism makes it difficult to effectively use distributed task queues. Thus, while distributed task queues may become attractive when production parallelizing compilers can effectively exploit more advanced constructs of parallelism, such as nested parallel loops, the experiments reported in this paper assume a task scheduling algorithm based on a centralized task queue.

Figures 3 and 4 show the task scheduling algorithms for the processor which executes a parallel DO statement and for the idle processors respectively. The removal of the loop descriptor from the task queue is performed by the first processor entering the barrier associated with the loop.

The implementation of the `lock()`, `unlock()`, and `increment()` functions 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 assumed to be atomic and

```
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 task scheduling

```
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 task scheduling

can be implemented with a sequence of lock, read-increment-write, and unlock operations. However, we will show that the frequent use of atomic increment in parallel Fortran programs makes it necessary to implement atomic increment with efficient hardware support.

During the execution of a parallel loop, each processor is assigned with different iterations, which is called iteration scheduling. We use the self-scheduling algorithm [20] to implement iteration scheduling. In this method, the self-scheduling code is embedded in the loop body. Each time a processor is ready to execute the next loop iteration, it executes this code to get a unique iteration number. The self-scheduling algorithm shown in Figure 5 is executed at the beginning

```
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

of each loop iteration and it performs an atomic increment operation on a shared counter. Unless the multiprocessor supports an atomic *fetch&add* operation, a lock is required to enforce mutual exclusion in accessing the shared counter.

Two alternative dynamic iteration scheduling algorithms, chunk scheduling and guided self-scheduling (GSS), have been proposed to avoid the potential bottleneck of scheduling the iterations one at a time [19]. When the number of iterations in a parallel loop is much larger than the number of processors, these algorithms reduce the iteration scheduling overhead by assigning multiple iterations to each processor at a time. This increases the effective granularity of parallel loops. The issue of granularity and scheduling overhead is discussed in Section 4. Both of these algorithms are proposed for DOALL loops. In the presence of dependences across iterations ,i.e., DOACROSS loops, scheduling more than one iteration at a time may sequentialize the execution of a parallel loop. In section 5, we present the program characteristics of our applications to show that the parallelism is mostly in the form of DOACROSS loops or DOALL loops with a small number of iterations. Therefore, our experimental evaluation of the architectural support assume self-scheduling algorithm rather than guided self-scheduling or chunk scheduling.

```
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

After all iterations of a loop have been executed, processors synchronize at a barrier. In this paper, we use a non-blocking linear barrier algorithm which is implemented with 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 iteration 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.

The barrier algorithm shown in Figure 6 specifies that 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 before they start executing another parallel loop whose descriptor is in the task queue. In contrast to the compile time scheduling of "fuzzy barrier" [14], this algorithm allows dynamic scheduling of loops to the

```
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 - dependence_distance] == 0) ;
}
```

Figure 7: Algorithm for Advance/Await operations

processors in a barrier. The linear barrier is a sequential algorithm and for the case where this algorithm proves to be a sequential bottleneck, a parallel algorithm (e.g. Butterfly barrier [15]) can be used. 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].

The combination of task scheduling, iteration self scheduling and non-blocking barrier synchronization algorithms presented in this section allows deadlock free execution of nested parallel loops with the restriction that DOACROSS loops appear only at the deepest nesting level [20].

The last type of synchronization, Advance/Await, is 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,

[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.

uses regular memory read and write operations, thus does not require atomic synchronization primitives. This implementation assumes a sequentially consistent memory system. In the case of weak ordering memory systems, an Await statement can be executed only after the previous memory write operations complete execution. For a multiprocessor with software controlled cache coherency protocol, Cedar Fortran Advance/Await statements include the list of variables whose values should be written to/read from shared memory before/after their execution. The implementation details of these statements under weak ordering memory system models or software controlled cache coherency protocols are beyond the scope of this paper.

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 a lock enforces mutual exclusion in accessing the task queue. Locks are also used to ensure correct modification of shared counters when there is no atomic *fetch&add* primitive in the architecture. Such shared counters are used both by iteration scheduling (See Figure 5) and barrier synchronization (See Figure 6).

There are several algorithms that implement locks in cache coherent multiprocessors using hardware synchronization primitives[2][13]. Virtually all existing multiprocessor architectures provide some type of hardware support for atomic synchronization operations. In theory, any synchronization primitive can be used to satisfy the synchronization needs of a parallel program. In practice, different primitives may result in very different performance levels. For example, a queuing lock algorithm [2][13] can be implemented efficiently with an *exchange-byte* or a *fetch&add* primitive whereas a *test&set* implementation may be less efficient. In this section, we outline the lock algorithms that we choose for each hardware synchronization primitive examined in our experiments.

Exchange-byte. 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 that contend for a lock. The variable `my_id` is set at the start of the program so that its value for the ith processor is $2 \times i$, where processors are numbered from 0 to $P-1$. During the execution, the value of `my_id` alternates between $2 \times i$ and $2 \times i + 1$. This eliminates the race condition between two processors competing for a lock which has just been released by one of them. The variable `queue_tail` holds the I.D. of the last processor which tried to acquire this lock. A processor which tries to acquire the lock receives the I.D. of its preceding processor via `queue_tail`. It then 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 specifically for its predecessor to release the lock. By mapping the elements of synchronization vector `flags[]` to disjoint 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 needs to be invalidated.

Test&set. Because of its limited functionality, *test&set* cannot be used to construct processor queues in a single atomic operation. Therefore, in this study, whenever the architecture offers only *test&set*, a plain *test&test&set* algorithm (see Figure 9) is used to implement all lock operations [2].

Fetch&add. Due to the emphasis on atomic increment operations in iteration scheduling and barrier synchronization, supporting a *fetch&add* primitive in hardware can significantly decrease the need for lock accesses in these algorithms. When the *fetch&add* primitive is supported by a system, a *fetch&add* implementation of *test&test&set* algorithm can be used to support the lock accesses in task scheduling as well as a queuing lock algorithm. The performance implications of

[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.

```
initialization() {
    flags[2P] = FREE ;
    flags[0...2P-1] = BUSY ;
    queue_tail = 2P ;
}

lock() {
    my_id = my_id XOR 1 ;
    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() {
    while(lock == BUSY || test&set(lock) == BUSY) ;
}

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

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

supporting the *fetch&add* primitive will be presented in Section 4 and Section 6.

Synchronization bus. In the Alliant FX/8, a separate synchronization bus and a Concurrency Control Unit is provided [1] which can improve parallel program performance by reducing the latency of both *fetch&add* operations and lock accesses. Such a bus provides the processors with a coherent set of shared counters and lock variables that can be accessed and updated in a single cycle. In this study, we also consider the case where a synchronization bus is used to implement synchronization operations.

The cost performance tradeoffs in synchronization support can only be determined by evaluating the performance implications of different schemes for real parallel applications. The needs for synchronization and scheduling support depend on the application characteristics like granularity of loop iterations, and structure of parallelism in the application. These issues are addressed by experiments reported in Sections 5 and 6.

3 Experimental Method

Trace driven simulation is used in our experiments to evaluate the performance implications of architecture support for synchronization. In our simulation model, a parallel Fortran program consists of a collection of sequential program segments called *task pieces*. To execute task pieces in parallel, several types of *events* arise: execution of DOALL and DOACROSS statements, execution of parallel loop iterations, barriers synchronization, and execution of Advance/Await statements. Each trace used in our simulations is a record of events that takes place during the execution of a parallel program and detailed information about instructions executed between each pair of events.

In this study, traces are collected by instrumenting the source code of parallelized applications. In a trace, each event is identified by its type and arguments, e.g., the synchronization point and the iteration number for an Await event. Each task piece is annotated with the number of dynamic instructions executed in the task piece and the dynamic count of shared memory accesses. These numbers are collected with the help of *pixie*, an instruction level instrumentation tool for the MIPS architecture [18]. Using a RISC processor model similar to MIPS R2000, where instruction execution times are defined by the architecture, the time to execute instructions in CPU and local cache can be calculated directly from the dynamic instruction count. On the other hand, the time to service the cache misses and the atomic accesses to the shared memory depends on the activities of other processors in the system. Therefore, a multiprocessor simulator is used to calculate the program execution time from a trace.

In order to assess the performance implications of synchronization primitives, a library of scheduling and synchronization routines as described in Section 2 is included in the simulator.

In the simulation model, the processor memory interconnect is a split transaction or decoupled access bus, where a memory access requested by a processor only occupies the bus when its request and response are transmitted between the processor and the memory modules. The bus is made available to other memory accesses while the memory modules process the current accesses. When the memory modules have long access latency, the split transaction bus plus memory interleaving allows the multiple accesses to be overlapped. In our experiments, we assume that shared memory is 8-way interleaved. Two memory module cycle times are used: 3 and 20 processor cycles. The 3-cycle memory module cycle time is chosen to represent the situation where slow processors are used in low cost multiprocessor systems. The 20-cycle latency represents the case where fast processors are used in high performance multiprocessor systems.

In our experiments, the atomic operations *test&set*, *exchange-byte* and *fetch&add* are performed in the memory modules rather than through the cache coherence protocol. Whenever a memory location is accessed by one of these synchronization primitives, the location is invalidated from the caches. The read-modify-write operation specified by the primitive is then carried out by the controller of the memory module that contains the accessed location. Note that this memory location may be brought into cache later by normal memory accesses made to that location due to spin waiting. This combination of atomic operation implementation in memory modules, the cache coherence protocol, and the split transaction bus is similar to that of Encore Multimax 300 series multiprocessors [11]. In Section 5, we present the characteristics of our application programs that lead to the choice of performing the read-modify-write in memory modules rather than through the cache coherence protocol.

Without any memory or bus contention, a synchronization primitive takes one cycle to invalidate local cache, one cycle to transmit request via the memory bus, two memory module cycles to perform the read-modify-write operation, and one cycle to transmit response via the memory bus. This translates into 9 and 43 cycles for our two memory module latencies respectively. A memory access that misses from cache takes one cycle to detect the miss, one cycle to transmit cache refill request via the bus, one memory module cycle time to access the first word in the missing block, four clock cycles to transmit the four words back to cache via the memory bus. This amounts to 9 and 26 cycles for our assumed memory module latencies. Note that the latency for executing synchronization primitives and refilling caches increases considerably in the presence of bus and memory contention. This effect is accounted for in our simulations on a cycle-by-cycle basis.

To evaluate the effectiveness of a synchronization bus, a single cycle access synchronization bus model is used. The synchronization bus provides single cycle lock/unlock operations on shared lock variables and single cycle fetch&add operations on shared counters. In the presence of conflicts, i.e., multiple requests in the same cycle, requests are served in round robin fashion. A summary of

Table 1: Timing assumptions without contention. M is the memory module cycle time.

primitive	**latency**
test&set	$3 + 2 * M$
exchange-byte	$3 + 2 * M$
fetch&add	$3 + 2 * M$
cache miss	$6 + M$
lock/unlock (synchronization bus)	1
fetch&add (synchronization bus)	1

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 timing assumptions for synchronization primitives is shown in Table 1.

In all the simulations, an invalidation based write-back cache coherence scheme is used. The shared memory traffic contributed by the application is modeled 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].

[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%.

4 Analysis of iteration scheduling overhead

In the execution of a parallel loop, the effect of iteration scheduling overhead on performance depends on the number of processors, total number of iterations, and the size of each iteration. In this section we first present 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 insight into how iteration scheduling overhead influences loop execution time, and will be used to analyze the simulation results later in this section. A more general treatment of program granularity and run-time overhead can be found in [16].

Consider a DOALL loop with N iterations where each iteration takes t_l time to execute without parallel processing overhead. 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.

For a 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

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

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 $t_l \times N$. 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 increases linearly 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 at least one other processor is scheduling an iteration at that time. The necessary condition for this case is

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

and the iteration scheduling overhead forms the critical path in determining the loop execution time. When iteration 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 iteration 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 first look at how loop execution time varies with loop granularity. Then we quantify the iteration scheduling overhead (t_{sch}) for different hardware synchronization primitives by simulating execution of a parallel loop with very fine granularity.

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 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 is significant in overall execution time or not.

The architectural support for synchronization primitives influences the execution time of 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 iteration scheduling algorithm. Since t_{sch} determines the execution time of fine granularity loops, we quantify how t_{sch} changes with synchronization primitives used, and the number of processors in the system.

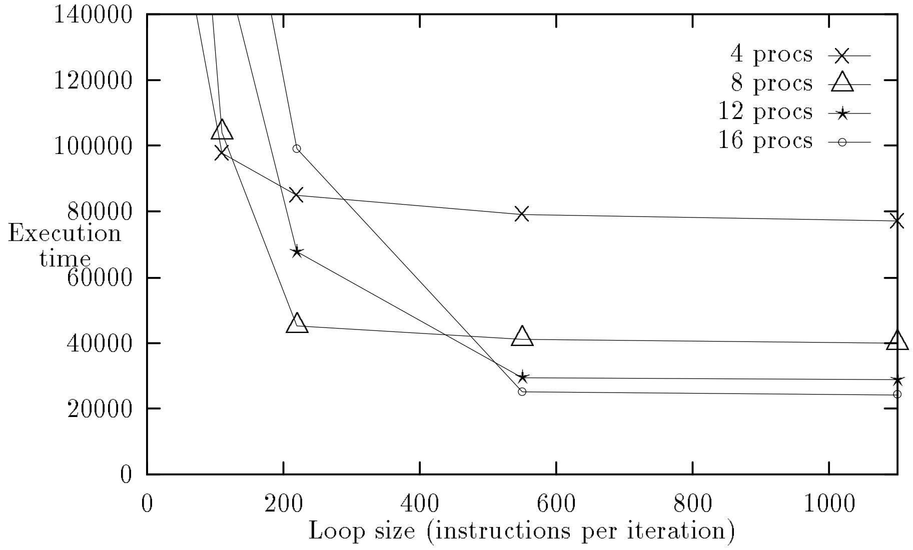

Figure 10: Execution time vs. granularity for test&set primitive

Figure 10 shows the simulation results for execution time vs. the size of an iteration in a DOALL loop with the *test&set* primitive implementing *test&test&set* algorithm for lock accesses. Similar curves were obtained for other synchronization primitives and for a synchronization bus supporting atomic lock operations. 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 number of executed instructions in the loop is kept constant while changing the number of instructions in an iteration.

Figure 10 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. The memory modules cycle time is assumed to be 3 processor cycles. Similar plots for other primitives (not shown due to space constraints) indicate that the critical iteration sizes are around 300 for exchange-byte, and 200 for a synchronization bus. Using the *fetch&add* primitive, the critical iteration size is around 100 instructions. As will be shown in Section 5, in the application programs we used in our experiments, the iteration sizes of the parallel loops vary from 10 to 1000 instructions. This shows that the choice

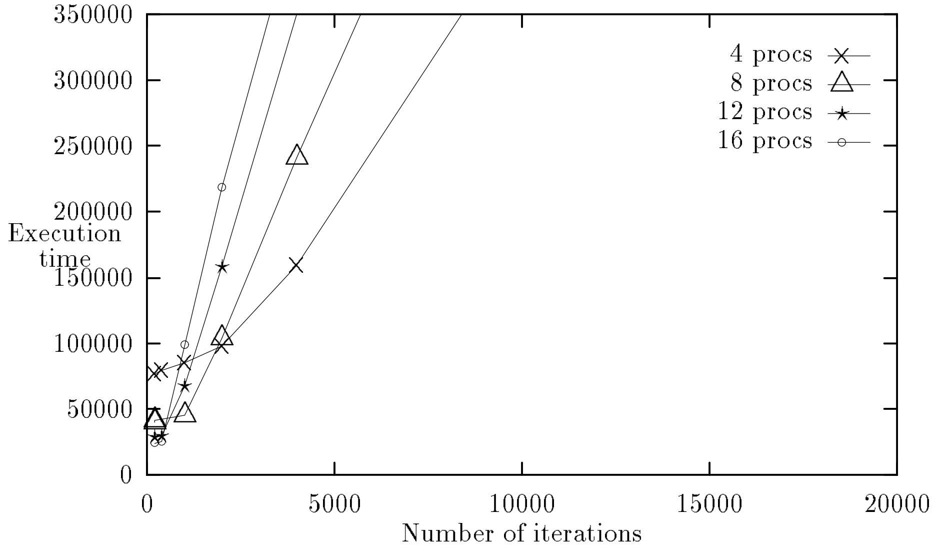

Figure 11: Execution time vs. number of iterations for test&set primitive

of synchronization primitives will influence the performance of some loops.

4.2 Scheduling overhead for fine grain loops

For fine grain loops, the loop execution time T_P is approximately $N \times t_{sch}$. The change of execution time with respect to the granularity of a set of synthetic loops is shown in Figure 11 for the *test&set* primitive implementing the *test&test&set* algorithm. Each of the synthetic loops has a total of 220000 executed 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.

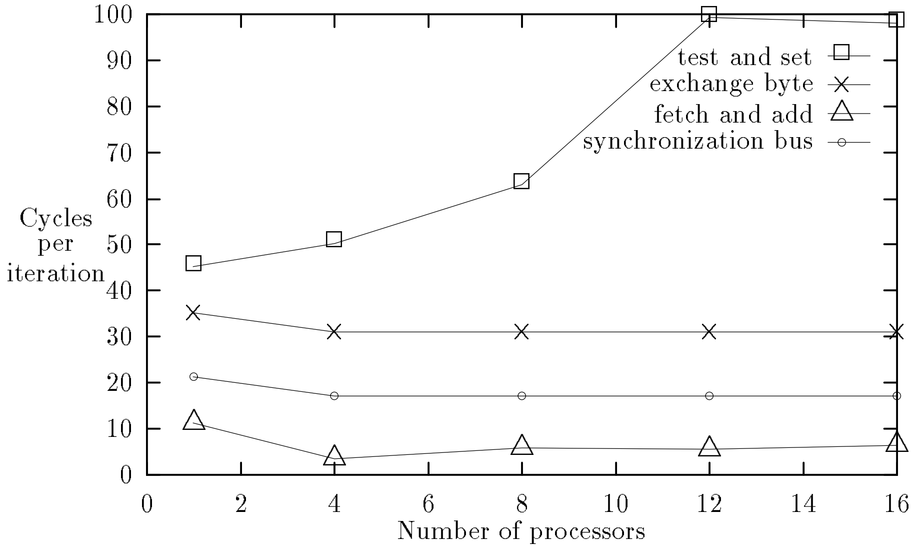

Figure 12: Iteration scheduling overhead vs. number of processors

Figure 12 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 wide variation 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 analyzed in this section.

5 Synchronization Characteristics of Applications

In this section we report some synchronization characteristics of the application programs used in our experiments. These characteristics help to focus our experiments and to analyze the experimental results. Section 5.1 presents the granularity of the parallel loops in these application programs. Section 5.2 deals with their lock access locality.

5.1 Parallelism characteristics of application programs

Experimental investigation of parallel processing requires realistic parallel programs. To support our experiments, we parallelized a set of programs from the Perfect Club benchmark set. KAP [17] was used as the primary parallelization tool. Using basic-block profiling (tcov), the frequently executed parts of the program were identified. If the parallelization of these parts were not satisfactory, the reasons for were investigated. In some cases, the unsatisfactory parallelization results were simply due to KAP's limitations in manipulating loop structures, e.g., too many instructions in loop body or too many levels of nesting. In these cases, the important loops were parallelized manually.

Among all the programs thus parallelized, four of them show a relatively high degree of parallelism, i.e., at least 60% of the computation was done in the parallel loops. These four programs are ADM, BDNA, DYFESM, and FLO52. ADM is a three-dimensional code which simulates pollutant concentration and deposition patterns in a lakeshore environment by solving complete system of hydrodynamic equations. The BDNA code performs molecular dynamic simulations of biomolecules in water. The DYFESM code is a two-dimensional, dynamic, finite element code for the analysis of symmetric anisotropic structures. The FLO52 code analyses the transonic inviscid flow past an airfoil by solving unsteady Euler equations.

Table 3: Granularity and parallelism in innermost parallel loops of benchmarks

program name	**average number of iterations**	**average number of instructions per iteration**
BDNA	450	515
FLO52	58	39
ADM	11	48
DYFESM	14	112

To perform experiments with these four programs, we insert instrumentation code in the programs and collected their traces. An in-depth treatment of automatic parallelization and the available parallelism in the Perfect Club programs can be found in [7][9].

Table 3 shows the available parallelism and granularity for the innermost parallel loops in the four automatically parallelized programs. In three of the four programs, FLO52, ADM, and DYFESM, the parallelism was exploited in the form of nested DOALL loops. For the BDNA program, the parallel loops were not nested and two thirds of the dynamic parallel loops were DOACROSS loops with dependence distances of one iteration.

For nested parallel loops, the number of iterations of outer loops does not differ from that of innermost parallel loops. Therefore, the number of iterations of parallel loops cannot be increased with techniques such as parallelizing outer loops or loop interchange. The small number of loop iterations suggests that chunk scheduling and guided self scheduling cannot be used to improve performance significantly beyond self-scheduling. The small number of instructions in each iteration suggests that architectural support is needed to execute these programs efficiently.

5.2 Locality of lock accesses in synchronization algorithms

In our simulations, all four programs exhibited very low locality for lock accesses. When a processor acquires a lock, we consider it a *lock hit* if the processor is also the one that last released the lock.

Otherwise, the lock acquisition is results in a *lock miss*. The measured lock hit rate for the four programs with four or more processors was less than 0.2%. Such a low lock access 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 of accesses to these locks. For each 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 processor which most recently acquired an iteration finishes the execution of that iteration before the completion of all the previously scheduled iterations. Due to low variation in the size of iterations of a parallel loop, 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 via the cache coherence protocol would result in excessive invalidation traffic, and would also increase the latency of atomic operations. On the other hand, algorithms like test&test&set require spinning on memory locations which are modified by atomic operations. Therefore all memory locations are cached with an invalidation based write-back cache coherence scheme. This simple scheme effectively use cache to eliminate excessive memory traffic due to spinning while efficiently executes atomic synchronization primitive in memory modules.

6 Experimental Results

In this section we present the performance implications of synchronization primitives on four application programs. The performance results are obtained by simulating a 16-processor system assuming centralized task scheduling, iteration self-scheduling, and linear non-blocking barrier synchronization. The system timing assumptions are the same as those summerized in Section 3.

To calculate the speedup, the execution time for the sequential version of a program *without any parallel processing overhead* is used as the basis.

Figures 13–16 present the speedup obtained in the execution of these program together with three categories of parallel processing overhead: iteration scheduling, task scheduling, and idle time. Each figure shows the results for one benchmark in two graphs, one for 3-cycle memory modules and the other for 20-cycle memory modules. The horizontal axis lists the combinations of architectural support and lock algorithms used in the experiments; these combinations are described in Table 4[4].

The task scheduling overhead corresponds to the time the processors spent to acquire tasks from the task queue. The iteration scheduling overhead refers to the time the processors spent in the self-scheduling code to acquire iterations. The processor idle time is defined as the time spent by processors waiting for a task to be put into the empty task queue. According to this definition, a processor is idle only if the task queue is empty when the processor completes its previously assigned task. This provides a measure of available parallelism in the parallelized programs.

Note that the three overhead numbers in Figures 13– 16 for each combination do not add up to 100%. The major part of the difference is the time that is actually spent in the execution of the application code. In addition, there are three more categories of overhead that are measured but not shown because they are usually too small to report. They are due to task queue insertion, barrier synchronization, and Advance/Await synchronization. The time it takes for processors to insert tasks into the task queue is less than 2% of the execution time for all experiments. For all four benchmarks, the barrier synchronization overhead is also measured to be less than 2% of the execution time. Of the four benchmarks, we encounter a significant number of DOACROSS loops

[4]The combination of *exchange-byte* primitive with *test&test&set* algorithm is not included because this case has the same performance as the *test&set* with *test&test&set* combination.

Table 4: The use of architectural support and lock algorithms in experiments.

architectural support and lock algorithm	**use in scheduling and synchronization**
fetch&add with test&test&set	Iteration scheduling and barrier synchronization algorithms use fetch&add for shared counter increments. Test&test&set algorithm based on fetch&add is used to access the task queue lock.
fetch&add with queuing lock	Iteration scheduling and barrier synchronization algorithms use fetch&add for shared counter increments. Queuing lock based on fetch&add is used to access the task queue lock.
synch. bus supporting fetch&add	Iteration scheduling and barrier synchronization algorithms use single cycle fetch&add on synchronization bus for shared counter increments. Synchronization bus provides single cycle lock operations to access the task queue lock.
exchange-byte with queuing lock	Queuing lock algorithm is used to access the locks associated with shared counters in iteration scheduling and barrier synchronization. It is also used to access the task queue lock.
test&set with test&test&set	Test&test&set algorithm is used to access the locks associated with the shared counters in iteration scheduling and barrier synchronization algorithms. It is also used to access the task queue lock.
synch. bus	Synchronization bus provides the single cycle lock/unlock operations to access the locks associated with the shared counters in iteration scheduling and barrier synchronization algorithms. They are also used to access the task queue lock.

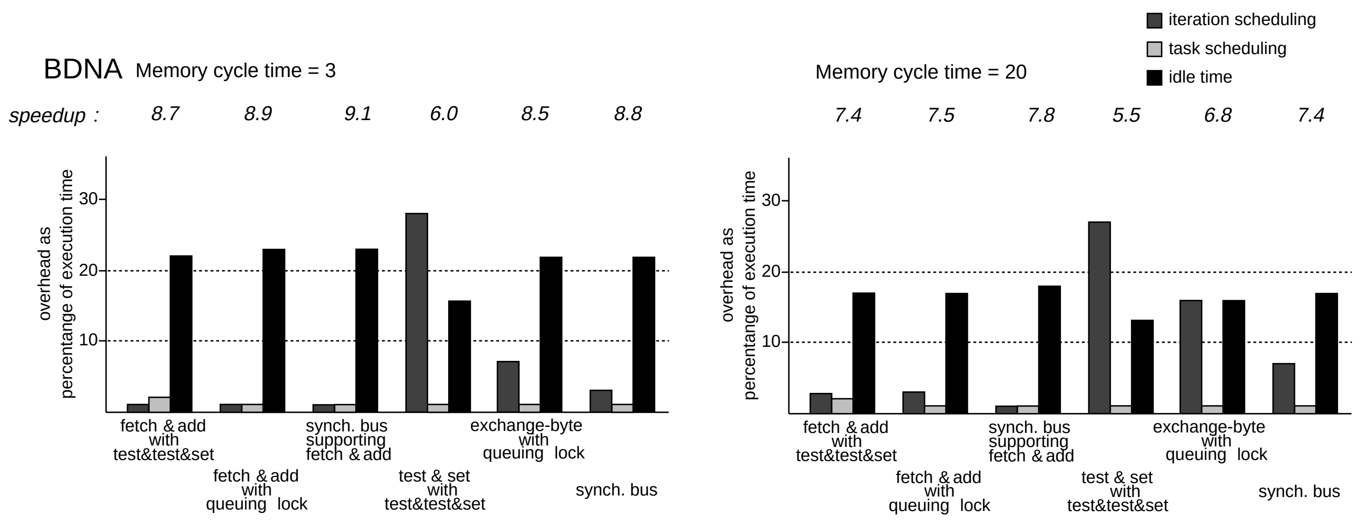

Figure 13: Speedup and scheduling overhead for BDNA with 16 processors

only in the BDNA program. The overhead for Advance and Await synchronization is about 11% of the execution time for 3-cycle memory modules and 18% for 20-cycle memory modules.

In Figures 13-16, the three experiments on the left side of each graph correspond to the cases where some form of *fetch&add* primitive is supported in hardware. For all four applications, when *fetch&add* operation is not supported, the iteration scheduling overhead increased significantly. This increase in overhead has a direct impact on the performance of the applications. Furthermore, the performance of *fetch&add* primitive with queuing lock algorithm (column 2) was at least as good as the performance of a synchronization bus supporting single cycle atomic lock accesses(column 6). This is true even when the memory module cycle time is 20 processor cycles, which implies a minimal latency of 43 cycles to execute *fetch&add*. Therefore, implementing the *fetch&add* primitive in memory modules is a effective as providing a synchronization bus that supports one cycle lock/unlock primitives.

For the BDNA program, task scheduling overhead is not significant for all experiments. As shown in Table 3, loops in BDNA have a large number of iterations and relatively large granularity.

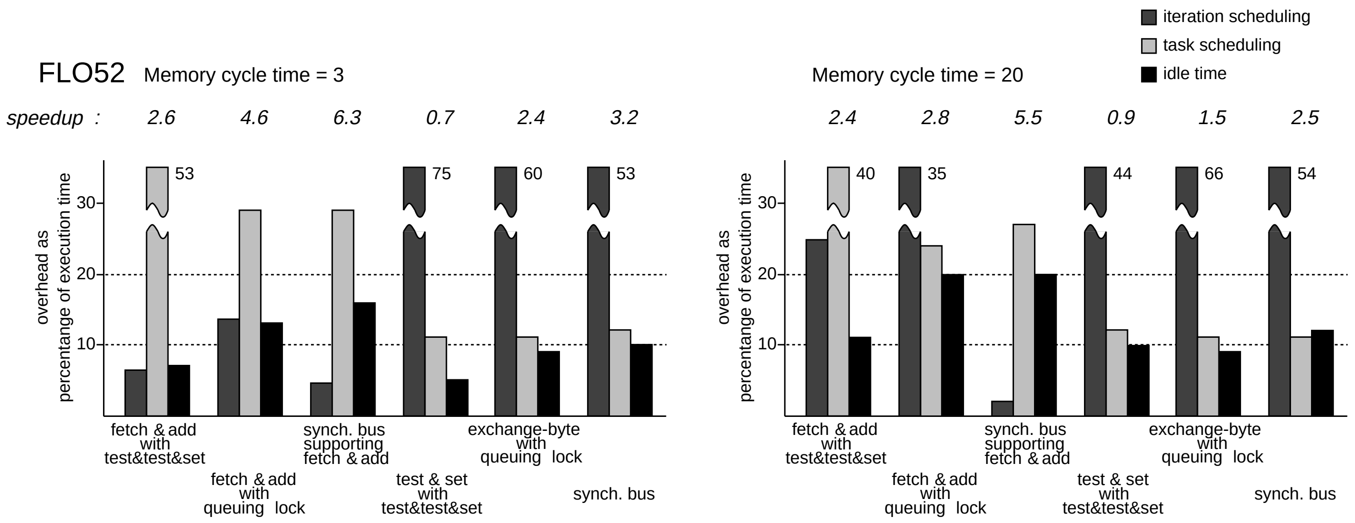

Figure 14: Speedup and scheduling overhead for FLO52 with 16 processors

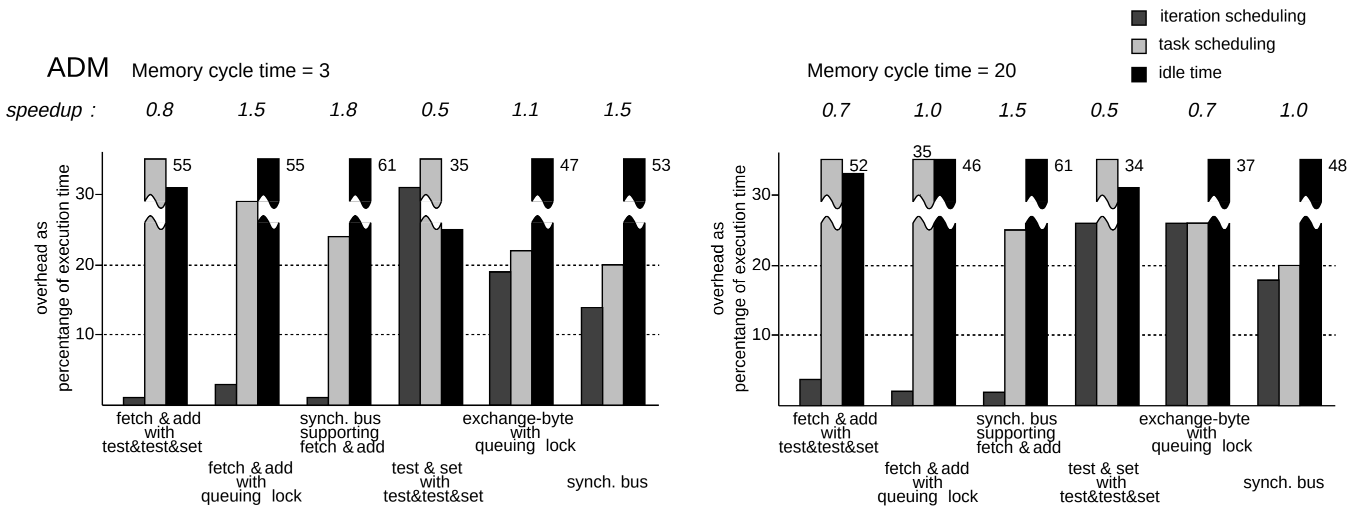

Figure 15: Speedup and scheduling overhead for ADM with 16 processors

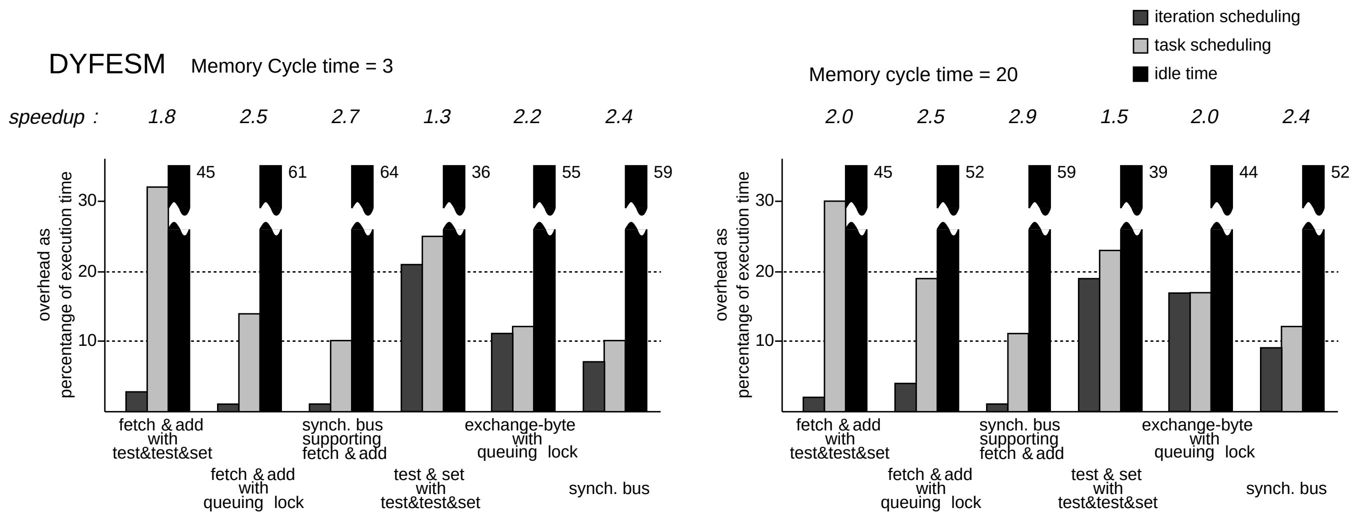

Figure 16: Speedup and scheduling overhead for DYFESM with 16 processors

Which results in infrequent invocation of the task scheduling algorithm. On the other hand, the remaining three programs have much less computation per parallel loop and this is reflected in the significant task scheduling overhead in their performance. Even with a synchronization bus that implements single cycle lock/unlock and single cycle *fetch&add* (column 3), the task scheduling overhead is still significant (See Figures 14–16). Note also that in FLO52, the relative percentage of time spent in task scheduling is higher with *fetch&add* support. This increased importance of task scheduling overhead is due to reduction of time spent in the iteration scheduling rather than increase of time spent in task scheduling.

We would like to make two more points about the lock algorithms. We have three different implementations of lock accesses. They are test&test&set algorithm (columns 1 and 4), queuing lock algorithm (columns 2 and 5), and a synchronization bus implementation of lock operations (columns 3 and 6). The test&test&set algorithm differs from queuing lock algorithm in the amount of bus contention it causes. On the other hand, the queuing lock algorithm is similar to a synchronization bus implementation of lock operations, except for a much higher lock access latency.

A comparison of speedup figures for columns 4 and 5 for the four programs show that reducing the bus contention is important for the performance of all the four application programs. The bus contention introduced by the test&test&set algorithm can seriously limit the speedup achieved by parallel processing. The same conclusion holds for *fetch&add* results shown in columns 1 and 2, even though lock operations are not used for iteration scheduling here. Comparison of the speedup figures for columns 5 and 6 shows that decreasing lock access latency can substantially increase the application program performance.

As for ADM and DYFESM, lack of parallelism is also an important factor for the low speedup figures. This can be observed from the idle time of processors in Figures 15 and 16. Finally, the results presented here demonstrate that the architectural support for synchronization and the choice of lock algorithms significantly influence the performance of all the four parallel application programs.

7 Concluding Remarks

In this paper, we analyze the performance implications of synchronization support for Fortran programs parallelized by a state-of-the-art compiler. In these programs, parallelism is exploited at the loop level that requires task scheduling, iteration scheduling, barrier synchronization, and advance/await.

Using simulation, we show that the time to schedule an iteration varies significantly with the architectural synchronization support. The synchronization algorithms used in executing these programs depend heavily on shared counters. In accessing shared counters, we conclude that lock algorithms which reduce bus contention do enhance performance. For the applications we examined, due to the importance of shared counters, a *fetch&add* primitive implemented in memory modules can be as effective as a special synchronization bus which supports single-cycle lock access.

Simulation with real programs show that for applications with fine granularity loops and limited parallelism, the execution time vary widely across synchronization primitives and lock algorithms. This is caused by the differences in the efficiency of iteration and task scheduling algorithms. Note that we assumed in our simulation moderate memory latency and split transaction bus. For architectures with very long memory access latency or those where atomic operations consume more memory bus bandwidth by requiring exclusive bus access during synchronization, the performance implications of synchronization support are expected to be even stronger.

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 Office of Naval Research under Contract N00014-91-J-1283, 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] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance of thread management alternatives for shared memory multiprocessors. *Proceedings of SIGMETRICS*, pages 49–60, 1989.

[4] 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.

[5] 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.

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

[7] G. Cybenko, J. Bruner, S. Ho, and S. Sharma. Parallel computing and the perfect benchmarks. Technical Report CSRD Rpt. No. 1191, Center for Supercomputing Research and Development, University of Illinois, 1991.

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

[9] R. Eigenmann, J Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect benchmark programs. Technical Report CSRD Rpt. No. 827, Center for Supercomputing Research and Development, University of Illinois, 1991.

[10] 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.

[11] Encore Computer Corp. *Multimax Technical Summary*, January 1989.

[12] 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.

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

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

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

[16] H. F. Jordan. Interpreting parallel processor performance measurements. *SIAM J. Sci. Stat. Comput.*, 8, No. 2:s220–s226, March 1987.

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

[18] MIPS Computer Systems, Inc. *RISCompiler and C Programmer's*, 1989.

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

[20] 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.

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

[22] 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.

[23] 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.