# ARCHITECTURAL AND SOFTWARE SUPPORT FOR EXECUTING NUMERICAL APPLICATIONS ON HIGH PERFORMANCE COMPUTERS

BY

SADUN ANIK

B.S., Middle East Technical University, 1986
M.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

## ABSTRACT

Numerical applications require large amounts of computing power. Although shared memory multiprocessors provide a cost-effective platform for parallel execution of numerical programs, parallel processing has not delivered the expected performance on these machines. There are two crucial steps in parallel execution of numerical applications: (1) effective parallelization of an application and (2) efficient execution of the parallel program on a multiprocessor. This thesis addresses the second step within the scope of automatically parallelized FORTRAN programs.

In this thesis, the mismatch between the needs of parallelized FORTRAN programs and the support for parallel execution in shared memory multiprocessors is identified as a cause of poor performance. The thesis addresses this problem from two angles, architectural and software support for parallel execution and compiler transformation to enhance program characteristics. Architectural features and synchronization and scheduling algorithms are studied to increase the efficiency of support for parallel execution. It is shown that architectures supporting atomic *fetch&add* primitives and synchronization busses can execute parallel programs more effectively. New algorithms for lock access and parallel task scheduling are proposed.

The thesis also explores compiler transformations which can modify parallel program characteristics to increase the parallel execution efficiency of less sophisticated architectures. It is shown that by using blocking transformations on nested parallel loops, program characteristics can be modified to decrease the need for scheduling and synchronization operations. This results in an increase in the efficiency of parallel execution, especially for multiprocessors with simplistic support for interprocessor synchronization.

## ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Wen-mei Hwu, for his time, guidance and patience. I consider myself very lucky to have worked with him. My respect for him goes beyond the teacher-student relationship. I will always remember him.

I would like to thank my parents for their love and support. Without them, graduate school would have been an unbearable journey. They were always there when I needed them and I will always feel their unconditional love.

I am grateful to all my friends for the good times. Tom Conte, has been a true friend for all these years we spent together. I am thankful to Steve Turner. If it weren't for him, I would not know the joy of skiing in the Rockies. I also thank Bob Janssens for being a true friend, patient and understanding.

Finally I would like to thank Marianne Holma for being my friend through the final years of graduate school. I would like to express my appreciation for everything she did for me and everything she was to me.

TABLE OF CONTENTS

Page

1. INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.1 Shared Memory Multiprocessors and Parallel Processing . . . . . . . 3
   1.2 Performance of Parallel Programs . . . . . . . . . . . . . . . . . . . 5
   1.3 Synchronization Needs of Parallel FORTRAN Programs . . . . . . . . 8
      1.3.1 Synchronization needs . . . . . . . . . . . . . . . . . . . . . . 10
      1.3.2 Locks and hardware synchronization primitives . . . . . . . . 17
   1.4 Dynamic and Static Methods . . . . . . . . . . . . . . . . . . . . . 21

2. INSTRUMENTATION AND SIMULATION METHODS . . . . . . . . . . 23
   2.1 Experimental Method . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   2.2 Architectural Model . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3. SYNCHRONIZATION SUPPORT FOR PARALLEL LOOP EXECUTION 31
   3.1 Analysis of Iteration Scheduling Overhead . . . . . . . . . . . . . . 32
      3.1.1 Granularity effects . . . . . . . . . . . . . . . . . . . . . . . . 35
      3.1.2 Scheduling overhead for fine-grain loops . . . . . . . . . . . . 37
   3.2 Synchronization Characteristics of Applications . . . . . . . . . . . 38
      3.2.1 Parallelism characteristics of application programs . . . . . . . 39
      3.2.2 Locality of lock accesses in synchronization algorithms . . . . 41
   3.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . 43
   3.4 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4. AN EFFICIENT TASK SCHEDULING ALGORITHM . . . . . . . . . . . 51
   4.1 Models for Nested Parallel Loop Execution and Synchronization Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
   4.2 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . 62
   4.3 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

5. COMPILER TRANSFORMATIONS FOR LOOP BLOCKING . . . . . . 67
5.1 Blocking Transformations . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.1.1 Partitioning an iteration space . . . . . . . . . . . . . . . . . . . 72
5.1.2 Legality of a blocking transformation . . . . . . . . . . . . . . . 74
5.2 Loop Bounds Generation . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

6. CONCLUSIONS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

VITA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

## LIST OF TABLES

Table Page

1.1: Granularity and parallelism in parallel benchmarks . . . . . . . . . . 21
2.1: Timing assumptions without contention . . . . . . . . . . . . . . . . . . 29
2.2: Assumptions for memory traffic . . . . . . . . . . . . . . . . . . . . . . . 30
3.1: The use of architectural support and lock algorithms in experiments . 47
4.1: Execution models for nested parallel loops . . . . . . . . . . . . . . . . 56

# LIST OF FIGURES

Figure Page

1.1: A DOALL loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.2: A DOACROSS loop . . . . . . . . . . . . . . . . . . . . . . . . . . 10
1.3: Producer algorithm for task scheduling . . . . . . . . . . . . . . . . 12
1.4: Consumer algorithm for task scheduling . . . . . . . . . . . . . . . . 12
1.5: Self-scheduling algorithm for loop iterations . . . . . . . . . . . . . 14
1.6: Barrier synchronization algorithm . . . . . . . . . . . . . . . . . . . 15
1.7: Algorithm for Advance/Await operations . . . . . . . . . . . . . . . 16
1.8: Queuing lock algorithm for lock accesses . . . . . . . . . . . . . . . 19
1.9: Test&test&set algorithm for lock accesses . . . . . . . . . . . . . . . 20
2.1: Multiprocessor trace example . . . . . . . . . . . . . . . . . . . . . 26
3.1: Execution time vs. granularity for test&set primitive . . . . . . . . . 36
3.2: Execution time vs. number of iterations for test&set primitive . . . . 37
3.3: Iteration scheduling overhead vs. number of processors . . . . . . . . 39
3.4: Speedup and scheduling overhead for BDNA with 16 processors . . . 44
3.5: Speedup and scheduling overhead for FLO52 with 16 processors . . . 45
3.6: Speedup and scheduling overhead for ADM with 16 processors . . . . 45
3.7: Speedup and scheduling overhead for DYFESM with 16 processors . . 46
4.1: Nonblocking queuing lock algorithm with *fetch&add* primitive . . . . 58
4.2: Operation of nonblocking queuing lock . . . . . . . . . . . . . . . . 59
4.3: Snapshot in task distribution (a) . . . . . . . . . . . . . . . . . . . 60
4.4: Snapshot in task distribution (b) . . . . . . . . . . . . . . . . . . . 61
4.5: Speedup of FLO52 program for different execution models . . . . . . 63
4.6: Speedup of DYFESM program for different execution models . . . . . 64
4.7: Speedup of ADM program for different execution models . . . . . . . 64
5.1: Iteration space and dependences of a loop-nest . . . . . . . . . . . . 69
5.2: Same loop-nest after skewing and blocking . . . . . . . . . . . . . . 69
5.3: Same loop-nest with nonrectangular blocking . . . . . . . . . . . . . 70
5.4: The illustration of blocking vectors for the loop-nest in Figure 5.3 . . 72

5.5: A two-level nested DOALL loop . . . . . . . . . . . . . . . . . . . . . . 82
5.6: Nested DOALL loop after blocking transformation . . . . . . . . . . . 82
5.7: Performance of FLO52 for 16 processors and *test&set* primitive . . . . 83
5.8: Performance of FLO52 for 16 processors and *exchange-byte* primitive 84
5.9: Performance of FLO52 for 16 processors and *fetch&add* primitive . . 84
5.10: Performance of FLO52 for 16 processors and synchronization bus . . 85
5.11: Performance of DYFESM for 16 processors and *test&set* primitive . . 87
5.12: Performance of DYFESM for 16 processors and *exchange-byte* primitive 87
5.13: Performance of DYFESM for 16 processors and *fetch&add* primitive . 88
5.14: Performance of DYFESM for 16 processors and synchronization bus . 88
5.15: Performance of ADM for 16 processors and *test&set* primitive . . . . 89
5.16: Performance of ADM for 16 processors and *exchange-byte* primitive . 89
5.17: Performance of ADM for 16 processors and *fetch&add* primitive . . . 90
5.18: Performance of ADM for 16 processors and synchronization bus . . . 90

# 1. INTRODUCTION

Bus-based shared memory multiprocessors have provided a cost-effective platform for parallel processing for more than a decade. However, utilization of these platforms for efficient parallel processing has not been easy. Large scientific applications written in FORTRAN have been the early candidates for parallel execution on shared memory multiprocessors. Scientific programs have relatively simple control structures. The data structures used in these programs have regular structure and predictable access patterns. However, achieving significant speedups with parallel execution of these programs has been an elusive goal. Most of the performance improvements for these programs resulted from tedious hand parallelization and tuning of the program. As this thesis establishes, the inefficiencies in parallel program execution result from a mismatch between the available compiler technology for parallelizing and restructuring programs, and the architectural and run-time support for parallel program execution in shared memory multiprocessors.

This thesis has two purposes. The first is to identify the performance bottlenecks in compiling and running a parallel scientific application. The second is to develop run-time and compile-time methods to increase the efficiency in parallel program execution. The organization of this thesis is follows. This chapter provides the background information on the procedures for compilation and parallel execution of FORTRAN programs. The issues in executing a parallel FORTRAN program on a shared memory multiprocessor are discussed. The need for run-time support for parallel execution is treated in a systematic manner. Existing algorithms for the synchronization and scheduling problems are presented at the level of architectural primitives, synchronization algorithms such as lock/unlock operations, and scheduling algorithms.

Chapter 2 describes the simulation tools developed for studying the performance issues in parallel program execution. A new technique is used to simulate the performance of several architecture/algorithm combinations. The developed tool combines trace-driven simulation of individual processors in a multiprocessor with the emulation of interprocessor communication to obtain accurate timing information.

Chapter 3 presents a discussion of issues in parallel program execution and experimental evaluation of the interaction between architectural support and algorithms used by the run-time system. Two important program characteristics are identified as having direct influence on program performance. The task granularity of a program (the amount of computation within a parallel task) affects the performance of the task scheduling algorithm. For a program in which task granularity is small, the performance of the program can be limited by the rate at which tasks are distributed to available processors. The loop granularity of a parallel loop (the amount of computation within a sequential iteration of a loop) affects the performance of the iteration scheduling algorithm.

Chapter 4 presents a new task scheduling algorithm to increase the efficiency of scheduling a parallel task. It is shown that the proposed algorithm can increase the utilization of processors in a shared memory multiprocessor. A nonblocking queuing lock algorithm is also introduced in this chapter which satisfies the functional requirements of the proposed scheduling algorithm.

Chapter 5 addresses the issue of loop granularity and the use of compiler transformations to increase the size of sequential threads of execution. This chapter provides a formal treatment of loop blocking transformations and describes a prototype implementation within the IMPACT compiler.

Finally, Chapter 6 includes the concluding remarks.

## 1.1 Shared Memory Multiprocessors and Parallel Processing

In executing a parallel program, scheduling computation to processors and synchronizing the processors introduce an overhead. The way this overhead affects the performance of a parallel program depends on both the program characteristics and the run-time system/architecture combination. As the units of computation become larger, the scheduling operations are needed less frequently, hence the effect of scheduling overhead decreases. The application programs used as benchmarks are parallelized scientific

FORTRAN programs taken from the Perfect Club benchmark set [1]. In a previous study of synchronization behavior of parallel programs, Davis and Hennessy used the combination of a shared memory architecture and large-grain parallel applications programmed with message passing paradigm, and concluded that synchronization does not introduce significant overhead [2]. The characteristics of parallel FORTRAN programs are very different from those programs using message passing paradigm. In these programs the basic form of parallelism is the parallel loop structure [3]. The parallel FORTRAN dialect used in this work is Cedar FORTRAN (CF) which has two basic parallel loop structures, DOALL and DOACROSS [4]. There is no dependence between iterations of a DOALL loop; therefore, the iterations can be executed in arbitrary order. A DOACROSS loop can have a dependence relation between iterations. The execution order of iterations of a DOACROSS loop has to preserve the dependencies.

The run-time system performs four basic duties in the execution of parallel loops:

1. Task scheduling. When a processor executes a DOALL/DOACROSS statement in a CF program , the parallel loop corresponding to that statement is ready to be executed. Starting the execution of this loop in parallel requires the notification of all of the available processors in the system through a *task scheduling* algorithm.

2. Iteration scheduling. All of the iterations of a parallel loop are distributed to the processors in a balanced way by an *iteration scheduling* algorithm.

3. Advance/Await operations. The data dependencies across iterations in a DOACROSS loop are enforced by *Advance/Await* operations.

4. Barrier synchronization. When all of the iterations of a parallel loop are executed, the processors participating in the execution synchronize at a *barrier*.

In the existing systems, the run-time system is integrated with the application by the compiler in the form of libraries [5],[6],[7].

## 1.2 Performance of Parallel Programs

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. As parallel architectures became available, parallelizing FORTRAN compilers have been developed for commercial and experimental multiprocessor systems to support these applications [4],[5],[6],[7],[8]. However the sustained performance of parallel systems for real-world applications fall short of the promise of these systems.

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 [9],[10],[11],[12],[13],[14],[15]. Instead of proposing new synchronization techniques, this thesis addresses two simple questions: (1) How much does architecture support for synchronization affect the performance of automatically parallelized numerical programs? (2) What can be done to improve the performance of parallel programs?

To answer these questions, we start with analyzing the needs for synchronization in parallelized FORTRAN programs. 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 of the synchronization needs in automatically parallelized programs. Synchronization issues arise in task scheduling, iteration scheduling, barriers and data dependence handling. A set of algorithms is presented which use lock()/unlock() and increment() operations. We then identify how several hardware synchronization primitives can be used to implement these generic synchronization operations. The synchronization primitives considered 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.

Beckmann and Polychronopoulos studied the effects of the barrier synchronization and iteration scheduling overhead on parallel loops in the presence of a synchronization bus [16]. Polychronopoulos also demonstrated that run-time overhead can limit the performance gain in executing parallel loops [17]. Arenstorf and Jordan modeled the execution overhead for several barrier synchronization algorithms [18]. The developed models provide an insight to the problem but to be able to choose the right combination of algorithms and architectural support, quantification of parameters is necessary.

The analysis of iteration scheduling overhead illustrates the need for quantification. Using the self-scheduling algorithm, iterations of a parallel loop are scheduled to the processors one at a time. Furthermore, self-scheduling is a sequential algorithm in which only one processor can schedule an iteration at a given time. Consider the execution of a loop with $N$ iterations on $P$ processors. For simplicity, it is assumed that all iterations of a loop are identical and each takes $t_{it}$ time to execute. Furthermore let $t_{sch}$ be the time it takes to schedule an iteration. In the case in which

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

the time it takes for a processor to execute a loop iteration is shorter than the the time needed by the remaining $P-1$ processors to schedule their iterations. In this situation, the processor completing execution of an iteration has to wait for the other processors to complete iteration scheduling operations in order to schedule a new iteration. When iteration scheduling becomes the bottleneck, the execution time is

$$t_{total} = N \times t_{sch} + t_{it}.$$

For the case in which

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

sequential iteration scheduling operations do not form the critical path of execution. Then the execution time can be written as

$$t_{total} = ((t_{sch} + t_{it}) \times \lceil N/P \rceil).$$

The important difference between the two cases is that in the first case increasing the number of processors does not decrease the execution time, and in the latter case the execution time is inversely proportional to the number of processors. A more detailed analysis of iteration scheduling overhead is presented in Section 3.1.

Of the two parameters, $t_{it}$ depends on the granularity of loops in the application and $t_{sch}$ is determined by the underlying architecture and synchronization primitives. Two variations of self-scheduling, chunk-scheduling and guided self-scheduling [19], increase the effective granularity by scheduling multiple iterations at a time. Their effectiveness relies on the assumption that $N \gg P$.

## 1.3 Synchronization Needs of Parallel FORTRAN Programs

The application programs used in this study are selected from the Perfect Club benchmark set [1]. The Perfect Club is a collection of numerical programs for benchmarking supercomputers. The programs were written in FORTRAN. For the experimental work, the programs were parallelized by the KAP/Cedar source-to-source parallelizer [20],[4] 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 most of the available parallelism for Perfect Club benchmark set programs [3]. 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 indirect array indexing requiring run-time dependence resolution. Also, limited interprocedural data-flow analysis can result in dependence equations with unknown variables resulting in conservative dependence analysis.

```
        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 1.1: A DOALL loop

Cedar FORTRAN has two major constructs to express loop-level parallelism: DOALL loops and DOACROSS loops. A DOALL loop is a parallel DO loop in which there is no dependence between the iterations. The iterations can be executed in parallel in arbitrary order. Figure 1.1 shows an example of a DOALL loop.

In a DOACROSS loop [21], there is a dependence relation across the iterations. A DOACROSS loop has the restriction that iteration $i$ can depend on iterations $j$ only where $j < i$. Because of this property, a simple iteration scheduling algorithm 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 [5]. An example of a DOACROSS loop is shown in Figure 1.2. The first argument of Advance and Await statements is the identifier for 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 [4]. 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 1.2: A DOACROSS loop

### 1.3.1 Synchronization needs

In executing parallel FORTRAN programs, the need for synchronization arises 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 available for execution. In this study, all experiments assume a task scheduling algorithm that uses a centralized task queue to assign tasks to processors. In Chapter 4, alternatives to this approach are considered. 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.

Use of a centralized task queue serializes the task scheduling. A number of distributed task scheduling algorithms have been proposed in the past, Anderson, Lazowska, and Levy [22] compared the performance of several distributed 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 the applications considered by this study and each task represents a piece of sequential code. These assumptions, however, are not valid for the current generation of automatically parallelized FORTRAN programs in which 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 have to acquire the same task so that they can work on different iterations of the parallel loop. This increases the complexity of the task scheduling algorithm considerably [23]. This lack of task-level parallelism makes it difficult to effectively use distributed task queues.

Figures 1.3 and 1.4 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.

```
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 1.3: 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 1.4: Consumer algorithm for task scheduling

The implementation of the `lock()`, `unlock()`, and `increment()` functions with different primitives is presented in the next section. By definition, the `lock()` and `unlock()` operations are atomic. Whenever underlined in an algorithm, the `increment()` operation is also assumed to be atomic and can be implemented with a sequence of lock, read-increment-write, and unlock operations. However, it is shown in Chapter 3 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 iteration is assigned to a processor, which is called iteration scheduling. The self-scheduling algorithm [24] is used to implement iteration scheduling. In this method, the self-scheduling code is embedded within the loop body. Each time a processor is ready to execute a loop iteration, it executes the embedded scheduling code to acquire a unique iteration number. The self-scheduling algorithm shown in Figure 1.5 is an example of the code executed at the beginning of each loop iteration. The algorithm 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, such algorithms reduce the iteration scheduling overhead by assigning multiple iterations to each processor per iteration scheduling operation. This increases the effective granularity of parallel loops. The issues of granularity and scheduling overhead are discussed in Section 3.1. 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. Therefore, algorithms such as chunk scheduling and GSS are applicable only to DOALL loops.

```
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 1.5: Self-scheduling algorithm for loop iterations

After all iterations of a loop have been executed, processors synchronize at a barrier. A nonblocking linear barrier algorithm which is implemented with a shared counter (see Figure 1.6) can be used for high performance barrier synchronization. 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 1.6 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. Therefore, a processor entering the barrier can immediately 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

```
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 1.6: Barrier synchronization algorithm

to the processors in a barrier. The linear barrier is a sequential algorithm, and for the case in which this algorithm proves to be a sequential bottleneck, a parallel algorithm (e.g., Butterfly barrier [15]) can be used. The experimental data presented in Chapter 3 illustrate that barrier synchronization execution is not a significant source of overhead. 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. 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.

The combination of task scheduling, iteration self-scheduling and nonblocking barrier synchronization algorithms presented in this section allows deadlock-free execution of

```
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 1.7: Algorithm for Advance/Await operations

nested parallel loops with the restriction that DOACROSS loops appear only at the deepest nesting level [24].

The last type of synchronization, Advance/Await, can be implemented by using a bit 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 1.7, 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 thesis.

In the HEP multiprocessor system, an alternative approach for enforcing data dependences across iterations was taken [25]. The use of full/empty bits that are associated with every data item provides a very efficient data synchronization mechanism in the execution of DOACROSS loops.

### 1.3.2 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 1.3 and 1.4), 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 by both iteration scheduling (See Figure 1.5) and barrier synchronization (See Figure 1.6).

There are several algorithms that implement locks in cache coherent multiprocessors using hardware synchronization primitives [12],[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, however, different primitives may result in very different performance levels. For example, a queuing lock algorithm [12],[13] can be implemented efficiently with an *exchange-byte* or a *fetch&add* primitive, whereas a *test&set* implementation may be less efficient. This section outlines the lock algorithms that are chosen for each hardware synchronization primitive considered in the experiments.

**Exchange-byte.** The *exchange-byte* version of the queuing lock algorithm is shown in Figure 1.8. 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 $i$th 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 ID of the last processor which tried to acquire this lock. A processor which tries to acquire the lock receives the ID of its preceding processor via `queue_tail`. It then writes its own ID 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

```
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 1.8: Queuing lock algorithm for lock accesses

caches of processors. When a processor releases the lock, only the cache line read by its successor has 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 1.9) is used to implement all lock operations.[1]

**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*

[1]However, it should be pointed 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.

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

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

Figure 1.9: Test&test&set algorithm for lock accesses

primitive is supported by a system, a *fetch&add* implementation of the *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 supporting the *fetch&add* primitive will be presented in Sections 3.1 and 3.3.

**Synchronization bus.** In the Alliant FX/8 multiprocessor, a separate synchronization bus and a Concurrency Control Unit are provided [5] 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, the case in which a synchronization bus is used to implement synchronization operations is also considered.

The cost performance trade-offs in synchronization support can be determined only by evaluating the performance implications of different schemes for real parallel applications. The needs for synchronization and scheduling support depend on the application characteristics such as granularity of loop iterations and structure of parallelism in the

Table 1.1: Granularity and parallelism in parallel 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 |

application. These issues are addressed by the experiments reported in Sections 3.2 and 3.3.

## 1.4 Dynamic and Static Methods

In general the performance improvements from parallel program execution change with both the application program characteristics and with the system support for parallel execution. In automatically parallelized FORTRAN programs, the amount of parallelism and its granularity are limited by the parallelization technology of the compiler. The existing technology limits the scope of parallelization to leaf functions and a few levels of innermost loops. The most significant effect of this is the granularity of parallelism. Table 1.1 shows the available parallelism and granularity for the innermost parallel loops in several automatically parallelized Perfect Club programs. In the presence of nested parallel loops, it is observed that the number of iterations of outer parallel loops did not differ from that of the innermost parallel loops. Both the limited parallelism and the rather small number of instructions in parallel loops suggest that a combination of both architectural support and good synchronization and scheduling algorithms are needed for efficient execution of these programs.

# 2. INSTRUMENTATION AND SIMULATION METHODS

Simulation-based performance evaluation of parallel systems presents unique challenges. In this thesis the execution of a parallel program on a shared memory multiprocessor is simulated using a hybrid method. The execution of a program has two parts. The first is the program execution local to a processor. For this part, the well-studied techniques for single-processor systems such as trace-driven simulation can be used. The second part comprises the sections of a program in which two or more processors interact through resources such as shared memory, memory bus, and synchronization bus. Such interaction can be both explicit, through the use of synchronization operations, and implicit, cache misses, etc.

In this thesis an integrated simulation/emulation model is used to model the execution of a parallel program on a shared memory multiprocessor. The behavior of the parallel program that is local to a processor is not dependent on the number of processors in the system. Therefore, this behavior can be captured into a trace collected from the execution of the program on a single processor. On the other hand, the interaction among multiple processors on a system is dependent on the number of processors in the system and the particulars of the underlying architecture. It is not possible to reliably capture the processor interaction through software instrumentation. Instrumentation of a parallel program slows down the execution significantly; hence, the generated timing information is distorted. In general, processor interaction relies on asynchronous events, and the execution time of algorithms is a nonlinear function of the number of processors executing them and the existing system load. Therefore, it is not possible to compensate for the distortion in the timing information in a collected trace by using postprocessing techniques.

On the other hand, the timing information concerning interprocessor communication can be captured by emulating the execution of the program segments which require interprocessor interaction. Emulating interprocessor communication both increases the accuracy of collected information and also makes it possible to evaluate new architectural features to enhance interprocessor communication efficiency.

## 2.1 Experimental Method

Trace-driven simulation is used in the experiments to evaluate the performance implications of architecture support for synchronization. In the 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 the simulations is a record of events that take place during the execution of a parallel program and detailed information about instructions executed between pairs of events.

Figure 2.1 shows the trace for a DOALL loop with three iterations. The lines in the trace starting with `I` denote the task pieces. Each line corresponds to the execution of a sequential portion of the application and contain the number of lines of code executed, and the number of read and write accesses to shared data. The line `F` indicates execution of a DOALL statement and the line `J` the end of the DOALL loop. The `B`-`E` pairs identify the beginning and end of iterations of the parallel loop respectively. The plain trace in the figure shows that after the execution of 10 sequential lines of the application, a DOALL statement is executed. The loop has three iterations, each executing 24 lines. After the execution of the parallel loop, 4 more lines are executed corresponding to the last `I` line of the trace.

It can be seen from this example that whenever iterations of a parallel loop execute the same number of instructions, there is redundancy in the trace concerning this loop. The execution of the loop can be summarized by providing the execution characteristics of a single iteration and providing the total number of iterations. The summarized trace is shown in Figure 2.1 for which the `F` line is replaced by `M` and the number of iterations. A parallel loop where the loop body does not contain any control flow instructions can be identified at the time of instrumentation as a candidate for summarized tracing. The

| *Plain* | *Summarized* | *Descriptions* |
|---|---|---|
| I 10 2 3 | | F : Start of parallel loop (Fork) |
| F | | J : End of parallel loop (Join) |
| B | | B : Beginning of an iteration |
| I 24 14 6 | | E : End of an iteration |
| E | I 10 2 3 | M $p_1$ : Similar to F (summarized) |
| B | M 3 | $p_1$ : Number of iterations |
| I 24 14 6 | I 24 14 6 | I $p_1 p_2 p_3$ : Block of instructions |
| E | J | $p_1$ : # of source code lines |
| B | I 4 0 4 | $p_2$ : # of shared data reads |
| I 24 14 6 | | $p_2$ : # of shared data reads |
| E | | |
| J | | |
| I 4 0 4 | | |

Figure 2.1: Multiprocessor trace example

compression of the trace from such loops results in the overall compression of the trace in excess of 1:100, hence making off-line storing of the execution trace possible.

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 [26]. Using a RISC processor model similar to the MIPS R2000, in which instruction execution times are defined by the architecture, the time to execute instructions in the 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 of the task pieces and an architectural level emulator is used to dynamically calculate the execution time of events. To assess the performance implications of synchronization primitives, a library of scheduling and synchronization routines as described in Section 1.3 is included in the simulator.

## 2.2 Architectural Model

In the simulation model, the processor memory interconnect is a split transaction or decoupled access bus, in which a memory access requested by a processor occupies the bus only when its request and response are transmitted between the processor and the memory modules. The bus is made available to other transactions while the memory modules process an access. When the memory modules have long access latency, the split transaction bus in addition to memory interleaving allows the multiple accesses to be overlapped. In the 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 in which slow processors are used in low-cost multiprocessor systems. The 20-cycle latency represents the case in which fast processors are used in high performance multiprocessor systems.

In the 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 the Encore Multimax 300 series multiprocessors [6]. Section 3.2 presents the characteristics of the 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 the two memory module latencies, respectively. A memory access that misses from cache takes one cycle to detect the miss, one cycle to transmit a cache refill request via the bus, one memory module cycle time to access the first word in the missing block, and four clock cycles to transmit the four words back to cache via the memory bus. This amounts to 9 and 26 cycles for the assumed memory module latencies. Note that the latency for executing synchronization primitives and refilling caches increases considerably in the presence of

Table 2.1: Timing assumptions without contention

| **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 |

bus and memory contention. This effect is accounted for in the 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 the timing assumptions for synchronization primitives is shown in Table 2.1.

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 2.2 lists the assumptions used to simulate the memory traffic for the task pieces. It is assumed that 20% of the instructions executed are memory references. In addition, it is measured that 6-8% of all instructions (approximately 35% of all memory references) are to shared data. It is assumed that references to shared data cause the majority of cache misses

Table 2.2: Assumptions for memory traffic

| **parameter** | **value** |
|---|---|
| memory/instruction ratio | 0.20 |
| shared data cache miss rate | 0.80 |
| nonshared data cache miss rate | 0.05 |

(80% shared data cache miss rate and 5% nonshared data cache miss rate). When the experiments are repeated by lowering the shared cache miss rate to 40%, the speedup figures reported in Chapter 3 change by less than 2%.

# 3. SYNCHRONIZATION SUPPORT FOR PARALLEL LOOP EXECUTION

This chapter presents an analysis and experimental evaluation of synchronization overhead on shared memory multiprocessors. Section 3.1 provides the analysis of iteration scheduling overhead and its effect on program execution time. This section also provides experimental data on architecture specific parameters in the formulations. The presented data provide a comparison of basic synchronization and scheduling operations across architectures supporting different synchronization mechanisms.

Section 3.2 describes the important characteristics of the benchmark programs used in the experiments and the behavior of lock algorithms used for mutual exclusion in scheduling algorithms.

Section 3.3 provides an experimental evaluation of the relation between architectural support for synchronization and parallel program performance. The results presented identify the contribution of iteration and task scheduling algorithms as well the available parallelism to the overall program performance. The presented data explore two systems with different memory access latencies.

Finally Section 3.4 includes the concluding remarks for this section.

## 3.1 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, the total number of iterations, and the size of each iteration. This section first presents the expressions for speedup in executing parallel loops in which 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 [27].

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. This section will look at the impact of scheduling overhead for two cases. For the first case it is assumed 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 depend 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 the 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 consider the case in which 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. Hence,

$$t_{sch} = t_{lock-transfer} + t_{update}.$$

The remainder of this section first look at how loop execution time varies with loop granularity. Then the iteration scheduling overhead ($t_{sch}$) is quanitified for different hardware synchronization primitives by simulating execution of a parallel loop with very fine granularity.

### 3.1.1 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 or not scheduling overhead is significant in overall execution time.

The architectural support for synchronization primitives influences the execution time of parallel loop in two ways. On the one hand, different values of $t_{sch}$ for different primitives result in different execution times 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 the simulation results on the variation of loop execution times across different implementations of the iteration scheduling algorithm are presented. Since $t_{sch}$ determines the execution time of fine-granularity loops, it is important to quantify how $t_{sch}$ changes with synchronization primitives and the number of processors in the system.

Figure 3.1 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

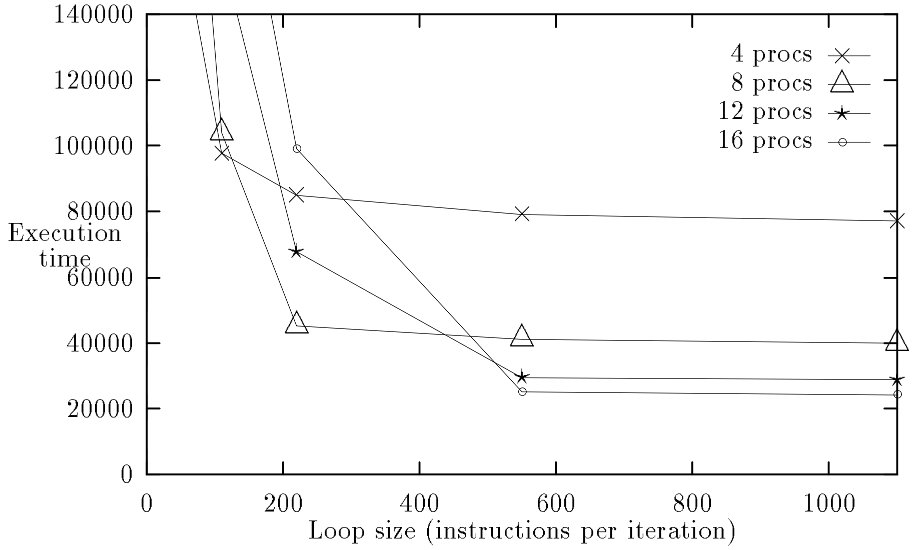

Figure 3.1: Execution time vs. granularity for test&set primitive

simulations, the total number of executed instructions in the loop is kept constant while the number of instructions in an iteration is changed.

Figure 3.1 shows that for 16 processors and using *test&set* primitive, there is a sharp increase in execution time when the iteration size is less than 550 instructions. The memory module cycle time is assumed to be 3 processor cycles. Similar plots for other primitives 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 3.2, in the application programs used in these experiments, the iteration sizes of the parallel loops vary from 10 to 1000 instructions. This shows that the choice of synchronization primitives will influence the performance of some loops.

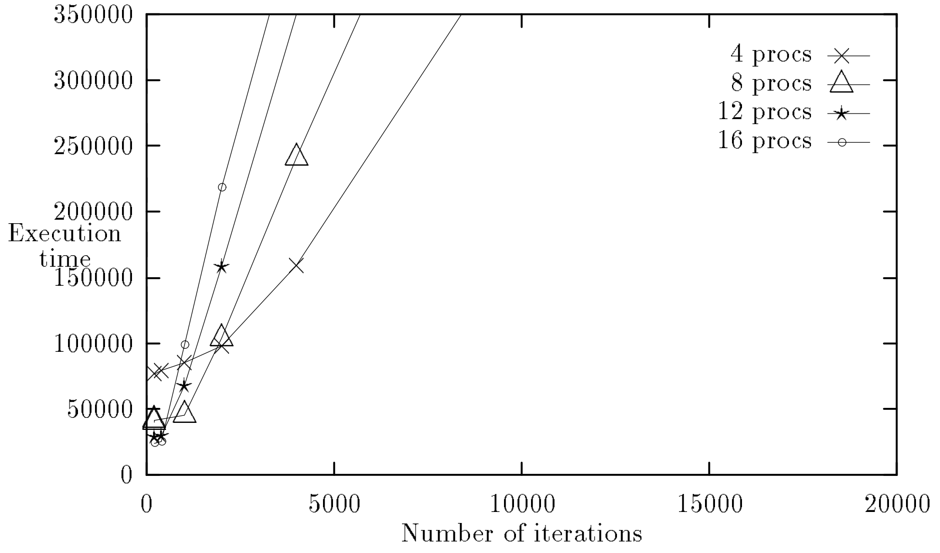

Figure 3.2: Execution time vs. number of iterations for test&set primitive

### 3.1.2 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 3.2 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 in which 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 3.3 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 the 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 is 98, 31, 17 and 7 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 show the highest performance achievable by hardware support for lock accesses alone. In Section 3.3, the performance figures are given for a synchronization bus which also supports single-cycle *fetch&add* operation. 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 that of all of the primitives analyzed in this section.

## 3.2 Synchronization Characteristics of Applications

In this section some synchronization characteristics of the application programs used in these experiments are reported. These characteristics help to focus the experiments and to analyze the experimental results. Section 3.2.1 presents the granularity of the

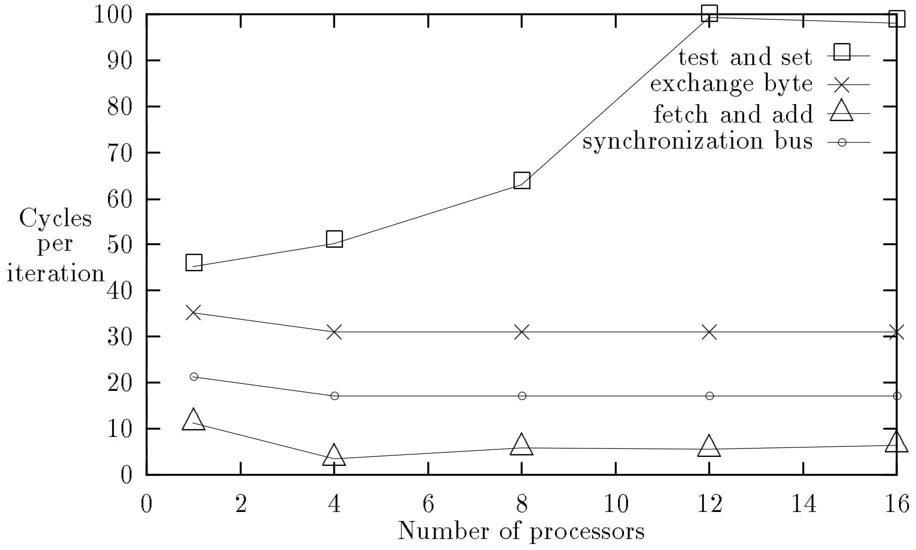

Figure 3.3: Iteration scheduling overhead vs. number of processors

parallel loops in these application programs. Section 3.2.2 discusses the lock access locality in the applications.

### 3.2.1 Parallelism characteristics of application programs

Experimental investigation of parallel processing requires realistic parallel programs. To support the experiments, a set of programs from the Perfect Club benchmark set are parallelized. The program KAP [20] was used as the primary parallelization tool. Using statement-level profiling (tcov), the frequently executed parts of the program were identified. If the parallelization of these parts was not satisfactory, the causes 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 of the programs thus parallelized, four 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. The program ADM is a three-dimensional code which simulates pollutant concentration and deposition patterns in a lakeshore environment by solving a 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.

To perform experiments with these four programs, instrumentation code is inserted in the programs and their traces are collected. An in-depth treatment of automatic parallelization and the available parallelism in the Perfect Club programs can be found in [28],[29].

Table 1.1 on page 22 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.

### 3.2.2 Locality of lock accesses in synchronization algorithms

In these simulations, all four programs exhibited very low locality for lock accesses. When a processor acquires a lock, it is considered a *lock hit* if the processor is also the one that last released the lock. Otherwise, the lock acquisition 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 of 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 such as 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 uses cache to eliminate excessive memory traffic due to spinning while it efficiently executes atomic synchronization primitive in memory modules. For example, while waiting for a busy lock, the test loop of a test&test&set algorithm brings the data corresponding to the lock value into the cache. As long as the processor owning the lock does not release it, the test loops of other processors are confined to their local caches. When the lock is released by its owner with a write operation to the lock data structure, the copies of the lock data structure in the other processors caches are invalidated. In the next test cycle of the test&test&set algorithm, the memory accesses to the lock location result in a cache miss for each waiting processor. After fetching the new value of the freed lock location from memory, the test cycle terminates and the test&set operation is executed. The caching of lock data structure reduces the memory traffic during the test cycle.

## 3.3 Experimental Results

In this section the performance implications of synchronization primitives on four application programs are presented. The performance results are obtained by simulating a 16-processor system assuming centralized task scheduling, iteration self-scheduling, and linear nonblocking barrier synchronization. The system timing assumptions are the same as those summarized in Section 2.1. 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 3.4–3.7 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 3.1. 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.

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

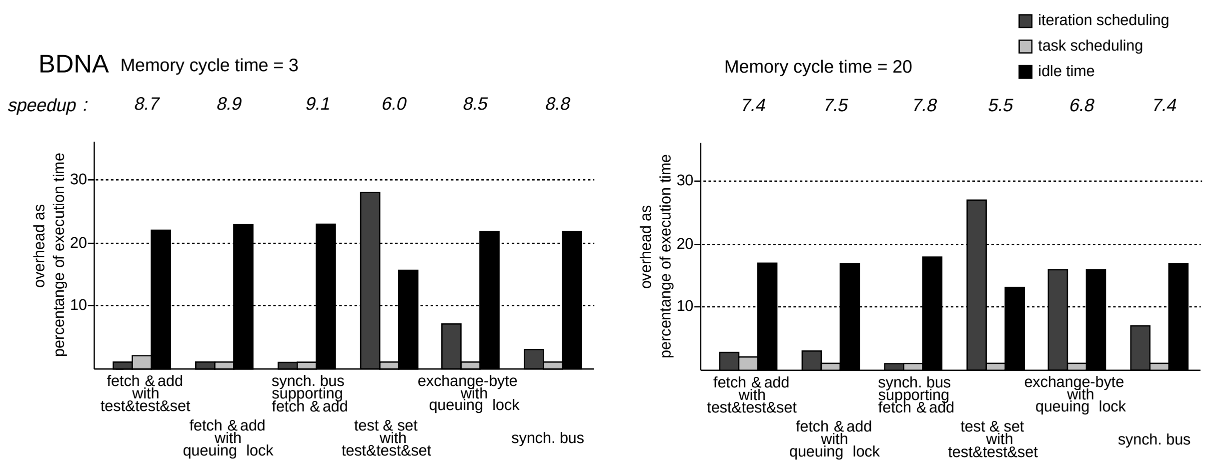

Figure 3.4: Speedup and scheduling overhead for BDNA with 16 processors

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 3.4–3.7 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 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.

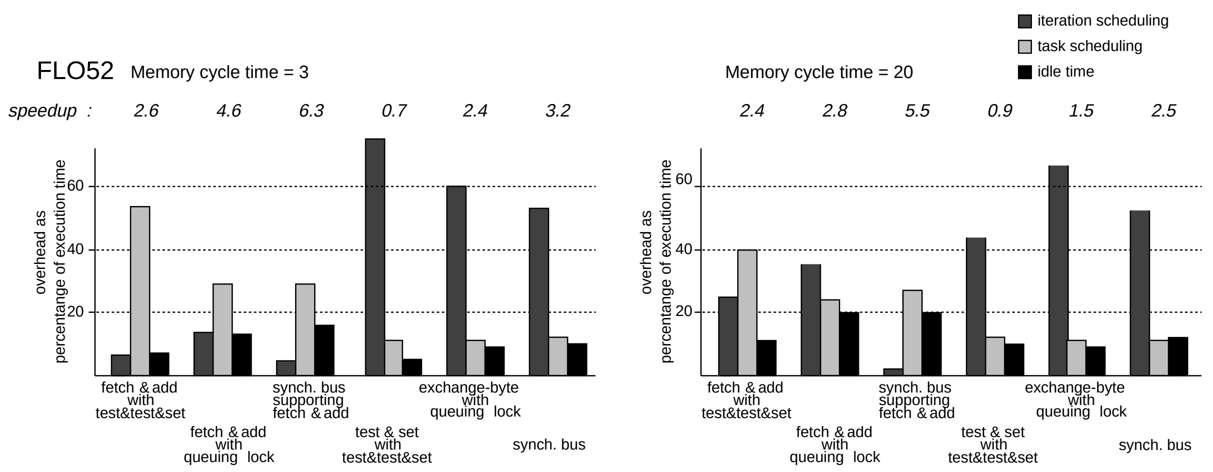

Figure 3.5: Speedup and scheduling overhead for FLO52 with 16 processors

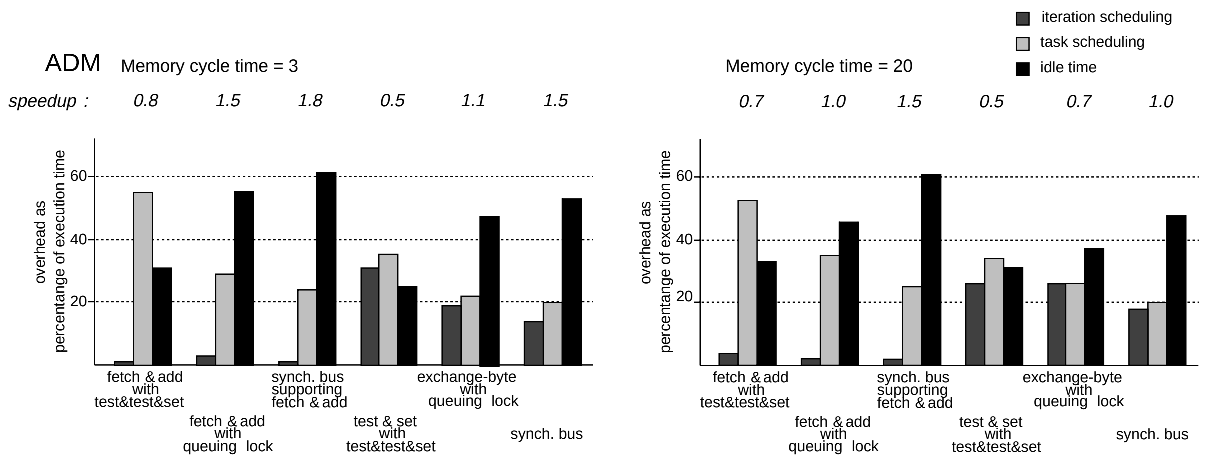

Figure 3.6: Speedup and scheduling overhead for ADM with 16 processors

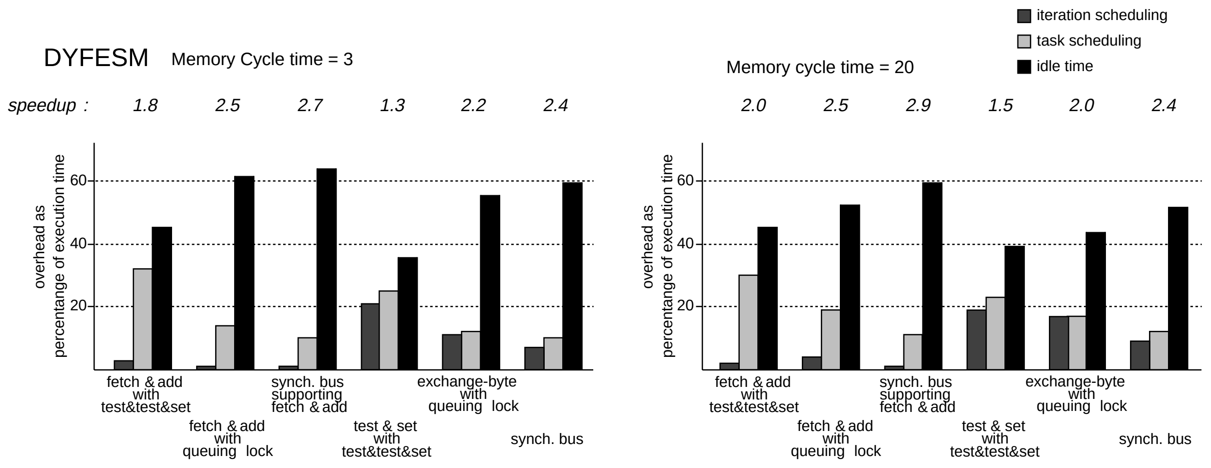

Figure 3.7: Speedup and scheduling overhead for DYFESM with 16 processors

In Figures 3.4-3.7, the three experiments on the left side of each graph correspond to the cases in which some form of *fetch&add* primitive is supported in hardware. For all four applications, when the *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 the *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 as effective as providing a synchronization bus that supports one cycle lock/unlock operations.

For the BDNA program, task scheduling overhead is not significant for all experiments. As shown in Table 1.1, loops in BDNA have a large number of iterations and relatively large granularity, which results in infrequent invocation of the task scheduling

Table 3.1: 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. |

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 3.5–3.7). 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 a reduction in time spent in the iteration scheduling rather than an increase in time spent in task scheduling.

There are three different implementations of lock accesses. They are the test&test&set algorithm (columns 1 and 4), the 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 the 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 in columns 4 and 5 for the four programs show that reducing the bus contention is important for the performance of all 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 the *fetch&add* results shown in columns 1 and 2, even though lock operations are not used for iteration scheduling here. A 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, the 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 3.6 and 3.7. Finally, the results presented here demonstrate that the architectural support for synchronization and the choice of lock algorithms significantly influence the performance of all four parallel application programs.

## 3.4 Concluding Remarks

This chapter analyzed 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, it is shown 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, lock algorithms which reduce bus contention do enhance performance. For the applications studied, 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.

Simulations with real programs show that for applications with fine granularity loops and limited parallelism, the execution times 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 moderate memory latency and a split transaction bus are assumed in the simulations. For architectures with very long memory access latency or those in which 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.

Chapter 4 addresses the task scheduling overhead in nested parallel loop execution. Using a distributed parallel task scheduling algorithm, it is shown that task scheduling overhead can be reduced. In Chapter 5, compiler-assisted loop restructuring techniques are discussed to increase loop granularity of parallel loops. By grouping iterations of a nested-loop structure together, it is shown that loop granularity can be increased without sacrificing parallelism.

# 4. AN EFFICIENT TASK SCHEDULING ALGORITHM

In scientific parallel applications, most of the computation involves the processing of large, multidimensional data structures which results in a high degree of data parallelism. This parallelism can be exploited in the form of nested parallel loops. In Chapter 3, task scheduling, scheduling parallel loops to multiple processors, is identified as a source of overhead in parallel program execution. This chapter proposes an efficient algorithm for executing nested parallel loops and presents a simulation-based performance comparison of different techniques using real application traces.

For this study, several compiler-parallelized applications are chosen from the Perfect Club Benchmark Suite [1] which show a high degree of parallelism in the form of nested parallel loops. Experience with these applications shows that most programs use moderately sized data structures, in the range of 400-10000 elements. When a two-dimensional array of 900 elements ($30*30$) is processed in a doubly nested parallel loop, the parallelism at each level is limited to 30. This demonstrates the need to exploit the parallelism of the inner loops. There are two performance-related issues in executing fine-to-medium-grain parallel loops [17]. The first is the relatively high overhead in scheduling individual iterations. This overhead can be reduced either by using dedicated architectural support for synchronization, e.g., a synchronization bus or powerful synchronization primitives, or by using blocking compiler transformations to increase loop granularity as will be discussed in Chapter 5

Existing systems with dedicated synchronization buses or other similar hardware have limited resources which are designed mostly for executing one parallel loop at a time. These limitations arise either from the number of available synchronization registers or the inability to repartition and manage the resources at run time in an efficient way. On the other hand, iteration scheduling algorithms which are more complicated than the self-scheduling algorithm are efficient only when the number of iterations in a parallel loop is much larger than the number of processors.

The second source of overhead is due to starting the execution of a parallel loop on multiple processors and synchronizing processors at the end of its execution — usually with a barrier. This overhead becomes significant when the total amount of computation in a parallel loop is small. This is true in general for the innermost parallel loops due to both the finer granularity and the relatively small number of iterations.

The results presented in Chapter 3 show that most of the overhead in executing a parallel loop is due to time spent by processors waiting to acquire a lock or spinning (mostly in cache) on a memory location which is used for signalling across processors. These situations arise in all synchronization algorithms, task distribution, iteration scheduling, and barrier synchronization.

Most existing high performance lock algorithms, e.g., tournament lock and queuing lock, are blocking algorithms. That is, a processor which is trying to acquire a lock is committed until the lock is acquired. This makes it impossible for a processor to utilize the idle time spent waiting for a lock. In Section 4.1, a nonblocking version of the queuing lock algorithm is presented and a task distribution algorithm which uses the this nonblocking property is described.

When all the iterations of a parallel loop are scheduled, the processors completing execution synchronize at a barrier. Traditionally, barrier synchronization has two phases. In the first phase, processors enter the barrier and start waiting for other processors to enter. In the second phase, when all processors have entered the barrier, the barrier exit is signalled and the processors can proceed in executing the program. The time between the barrier entry and exit is idle time for processors. In the *fuzzy barrier* [14], the compiler schedules instructions to be executed in this otherwise idle period. In the next section, a more relaxed barrier model in which only one processor is required to wait for the barrier exit is described. In this model, all processors but one are allowed to leave the barrier upon barrier entry. In executing nested parallel loops, this type of barrier synchronization allows the dynamic overlapping of execution of one loop with the barrier synchronization of another loop.

Nonblocking lock algorithms combined with nonblocking barrier synchronization allows us to explore more flexible dynamic execution schemes for nested parallel loops and compare their performances with the traditional style (one loop at a time) execution of parallel loops. The modifications on the algorithms and the related changes in execution model are discussed in the next section.

Section 4.2 presents the simulation results for different techniques of nested parallel loop execution, and concluding remarks for this chapter are presented in Section 4.3.

## 4.1 Models for Nested Parallel Loop Execution and Synchronization Algorithms

An application with nested parallel loops can be executed in several ways depending on the available compiler, hardware and run-time system support. The simplest model involves executing the outermost loop in parallel and the inner parallel loops sequentially. This model is employed in the run-time system of existing shared memory multiprocessors [5]. It has the advantage of being simple but exploits only part of the available parallelism.

Another model for execution of nested parallel loops involves exploiting the parallelism at a single level but using compiler transformations to *collapse* the nested loops into a one-level loop. When successful, these transformations result in the exploitation of all of the available parallelism within the single-level parallel loop execution paradigm. Such compiler transformations are a current field of research, and the study of different varieties of transformations and their applicability to particular loop structures are beyond the scope of this thesis. For the purpose of comparison, it will be assumed that a compiler can successfully perform loop restructuring in which all of the parallelism of the inner loops can be exploited in a single DOALL. Although such a transformation may not be realistic, it does provide a good reference point in comparing the performance of different models. It is also assumed that all of the loop iterations are scheduled one at a time. It should be pointed out that exploitation of high degree of parallelism in a single-level DOALL loop may make it possible to efficiently use different types of loop iteration scheduling algorithms other than self-scheduling [19].

The third model of execution is executing nested parallel loops by executing inner loops one at a time. The parallelism of each inner loop is then exploited with multiple processors. This model stays true to the parallelism structure in the application and can be implemented easily with existing task scheduling, loop iteration scheduling and barrier synchronization algorithms. One disadvantage of this model is the increased overhead in distributing loops to the processors to begin their execution and synchronizing at barriers at the end of execution for each inner loop.

A refinement of this model involves loosening the definition of barrier synchronization by allowing processors entering a barrier to leave it immediately and start execution of other parallel loops — whenever there are some. The parallel FORTRAN semantics require only the processor which is going to execute the code after the parallel loop to wait for the barrier exit. This model has the advantage of reducing the barrier execution overhead for most of the processors by allowing the execution of application code in the time which would otherwise be spent waiting at a barrier. Such a barrier algorithm can

Table 4.1: Execution models for nested parallel loops

| **execution model** | **description** |
|---|---|
| 1 | Inner loops are executed sequentially |
| 2 | Nested parallel loops collapsed into single parallel loops (ideal) |
| 3 | Nested parallel execution, blocking barriers |
| 4 | Nested parallel execution, nonblocking barriers |
| 5 | Nested parallel execution, multiple simultaneous access task queues |

be easily implemented as a linear barrier in which all processors but one do not wait for the barrier exit signal. An implementation of this model involves a task queue from which the parallel loops, whose control dependencies are satisfied, are distributed to the processors. In parallel FORTRAN, execution of a DOALL or DOACROSS statement implies that the control dependencies for corresponding parallel loops are satisfied.

The final model is a further refinement of the fourth model. In developing this model the goal was to decrease the task scheduling overhead, specifically the waiting time to acquire exclusive access to the task queue, which was observed to be significant for the innermost loops in preliminary simulations. Here, a number of task queues are used for distributing tasks in which all processors have access to all task queues; therefore, the throughput for task scheduling is increased. A summary of these five models is given in Table 4.1

Multiple task queues to increase task scheduling throughput have been proposed before [22]. Unlike the previously proposed algorithms, the method proposed in this thesis does not require any assignment of processors to task queues and does not introduce a load-balancing problem. Furthermore, the proposed method allows scheduling of a parallel loop to multiple processors. Previous work in task scheduling has focused exclusively on sequential tasks.

In the proposed scheme, accesses to task queues are controlled by locks. Each processor tries to gain exclusive access to all of the queues. Using a nonblocking lock algorithm, a processor can wait for multiple locks simultaneously. When a processor acquires the lock of a task queue (gains the exclusive access), it stops trying to gain access to the remaining task queue locks. This type of queue access is possible only if the underlying locking algorithm allows a processor to abort the process of acquiring a lock. The low-contention, high performance lock algorithms such as queuing lock or tournament lock require the processors to commit themselves upon a lock() operation [12],[13]. In this chapter a nonblocking queuing lock algorithm is presented which allows a processor to abort a lock() operation with the option of retrying in the future.

The nonblocking queuing lock algorithm shown in Figure 4.1 uses the *fetch&add* primitive for setting up a lock queue. The atomic *fetch&add* operation is used to obtain a unique number to set up a queue. The array `a[]` is used for mapping these unique numbers to the processors. The algorithm assumes that on a `P`-processor system, the processors are numbered `0` to `P-1`, and the variable `my_id` contains this number. A processor entering the lock queue reads the number of the processor ahead of itself from array `a`.

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

Figure 4.1: Nonblocking queuing lock algorithm with *fetch&add* primitive

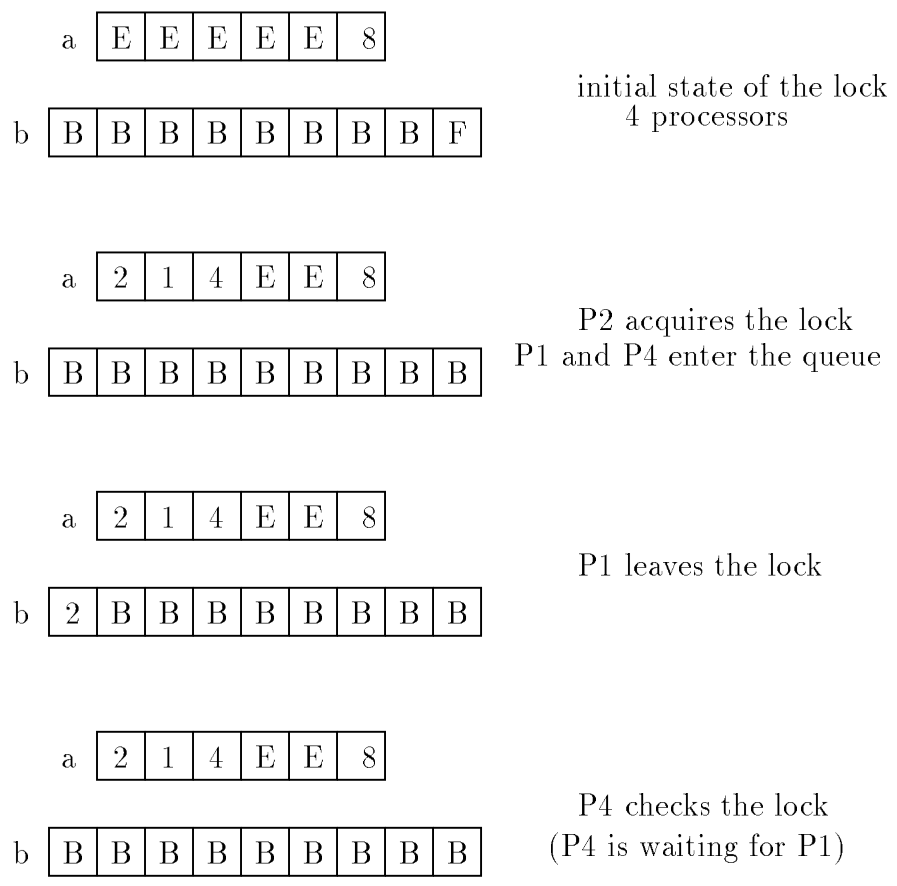

Figure 4.2: Operation of nonblocking queuing lock

A processor set up in the lock queue waits for the processor in front of it to release the lock. The array `b` is used for signalling between the processors. A processor leaving the lock queue also uses array `b` to pass the ID of the processor ahead to the processor behind for proper operation of the queue. While the lock is held by a processor, other processors can enter and leave the lock queue many times. The algorithm includes several checks to prevent race conditions and to ensure the proper use of arrays `a` and `b`. Use of the algorithm is shown in Figure 4.2

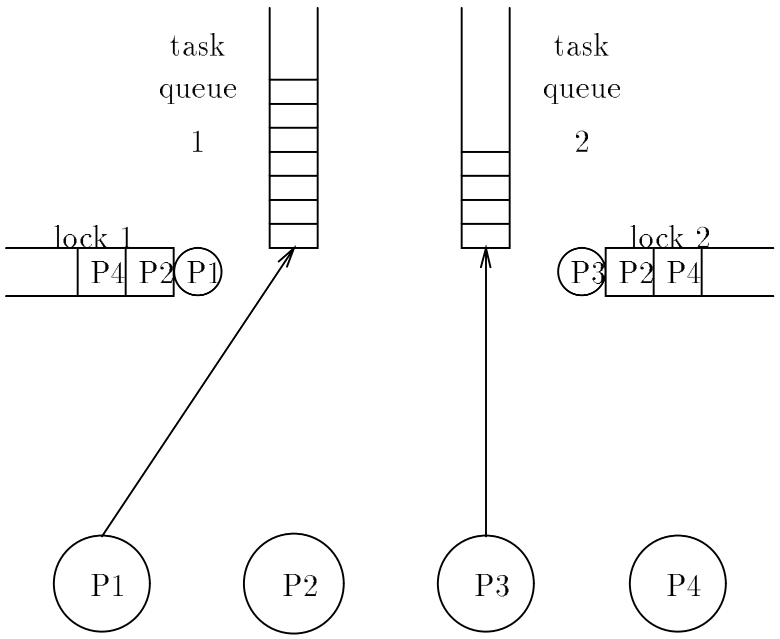

Figure 4.3: Snapshot in task distribution (a)

The task scheduling algorithm used for model 5 to increase task distribution throughput is a straightforward extension of centralized task scheduling. A fixed number of task queues are used for distributing tasks. Each processor starts the task scheduling algorithm by checking whether the queues are empty or not. A processor tries to gain exclusive access to the queues which are not empty. Upon gaining access to a task queue, it aborts the lock operations for other queues and schedules a parallel loop for execution.

Figures 4.3 and 4.4 demonstrate the simultaneous use of multiple task queues. In Figure 4.3, processors P1 and P3 own the locks for accessing the two task queues. Processors P2 and P4 are waiting for both of the locks in the respective lock queues. In Figure 4.4, P1 releases the lock of task queue 1 and the next processor in the queue of lock 1, P2, acquires lock 1. After acquiring lock 1, P2 leaves the queue of lock 2. Therefore, P4

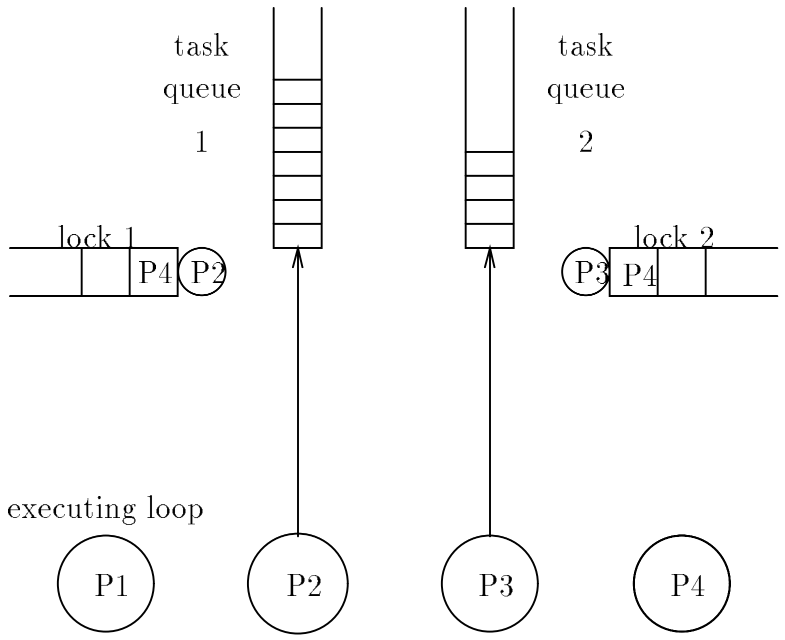

Figure 4.4: Snapshot in task distribution (b)

advances in both lock queues. If P2 were behind P4 in the queue of lock 2, it would simply disappear from that queue.

Balancing the amount of computation scheduled from each queue is important only for the throughput of task scheduling. Even in the case in which all loops are distributed from one of the task queues, all processors will execute the loop iterations in parallel as in model 4. To obtain a balance at least in the number of loops scheduled from task queues, processors inserting tasks to the queues start from a random queue and visit queues in round-robin fashion for each new parallel loop.

To evaluate the performance of several parallel processor architectures, the high-level trace driven simulator described in Chapter 2 is used. The simulator implements the task management and synchronization algorithms for different models of nested parallel loop execution described in Section 4.1. In the experiments, the atomic operation *fetch&add* is implemented in shared memory. The processor memory interconnection is a decoupled access bus whose cycle time is equal to the processor cycle time. It is assumed that shared memory is 8-way interleaved where an access to a module takes 3 bus cycles. An atomic *fetch&add* operation takes 6 bus cycles to execute in memory.

## 4.2 Experimental Results

In the comparison of the performance of different execution models for nested parallel loops, the traces used are from three parallelized Perfect Club applications. These programs are FLO52, DYFESM and ADM. Of the thirteen programs in the Perfect Club set, four of them were parallelized to the extent that at least half of the computation is done in parallel loops. Out of these four applications, the parallel loops in the BDNA program are not nested. Therefore, the three applications listed above are used for this study. In the parallelization process, a state-of-the-art source-level parallelizer with minimal user interference in the process was used.

Among the three programs, FLO52 had the highest level of parallelization. Only 1% of the instructions in the program trace were in sequential sections. The percentage of sequential instructions in the trace were considerably higher for the other two programs — around 25% for DYFESM and 32% for ADM. The programs have varying levels of granularity and parallelism for the innermost parallel loops. Program FLO52 has an average parallelism of 58 iterations per innermost parallel loop, and these loops have a

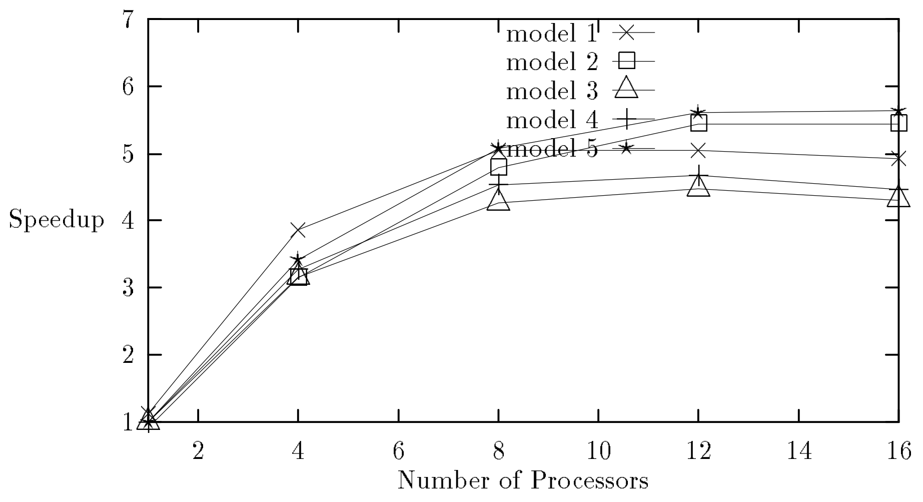

Figure 4.5: Speedup of FLO52 program for different execution models

granularity of 39 instructions per iteration. Program DYFESM has an average parallelism of 14 and granularity of 112 at innermost parallel loops, and these numbers for ADM are 11 and 48, respectively.

The speedup figures for the three programs, FLO52, DYFESM, and ADM are shown in Figures 4.5, 4.6, and 4.7, respectively. A quick comparison of these figures shows that multilevel parallel loop execution with nonblocking barriers and multiple task queues (model 5) and perfect loop collapsing (model 2) perform consistently better than other execution models when the number of processors is large. Furthermore, the two models always resulted in very similar performances. This demonstrates that for the architecture model used, performance gains of perfect loop collapsing by a compiler can be achieved by executing nested parallel loops with proper run-time scheduling and synchronization algorithms.

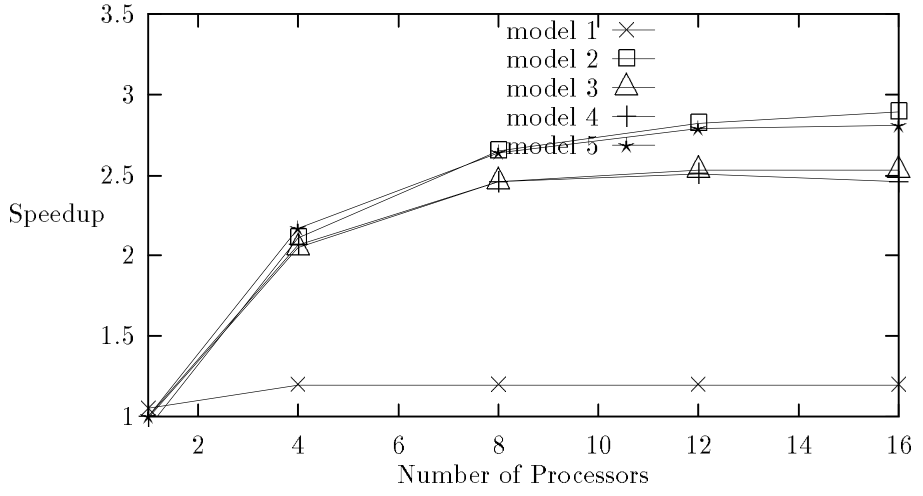

Figure 4.6: Speedup of DYFESM program for different execution models

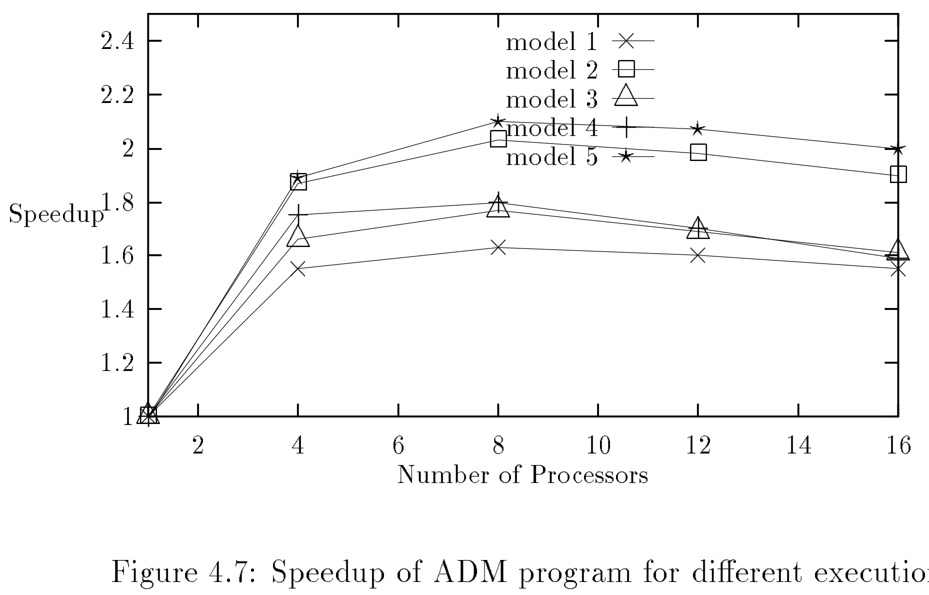

Figure 4.7: Speedup of ADM program for different execution models

Executing the outermost loop in parallel and inner loops sequentially (model 1) resulted in different behavior among the three programs. In FLO52, where parallelization is most successful, the outer loop parallelism proved to be sufficient for achieving speedup. It actually performed slightly better than models 3 and 4, which were simpler models for executing the innermost loop in parallel. The poor performance of these two models for this program can be attributed to the low granularity of innermost loops — hence high overhead for iteration scheduling. In DYFESM, model 1 resulted in the loss of almost all available parallelism. This model was also the worst performer for program ADM though the speedup was close to that obtained from models 3 and 4.

Finally, it is observed that the use of nonblocking barriers alone didn't contribute significantly to performance. The task distribution algorithm proved to be much more significant than the barrier synchronization algorithm.

## 4.3 Conclusions

This chapter addresses several practical aspects of nested parallel loop execution. Five different models are used for executing nested parallel loops. In the first model, only the outermost loop of a nested loop structure is executed sequentially. It is concluded that the performance of this model depends heavily on the application and can cause serious performance degradation due to loss of parallelism. The second model used single-level parallel loop execution with perfect loop collapsing (by a compiler). This combination resulted in good performance, a combination of a simple execution model and fully exploited parallelism.

Three variations of true nested parallel loop execution are explored. In the first one, model 3, the innermost loops are executed one at a time and synchronization is done with a conventional barrier. Then a nonblocking barrier version of this model was used, model 4, in which processor idle time in a barrier is minimized. However, this does not result in any significant performance difference. As a refinement to this model, in model 5, a nonblocking queuing lock algorithm and a multiple-task queue-based task distribution algorithm are used to decrease task scheduling overhead. The simulations show that this model can give the performance of a perfect loop-collapsing transformation.

# 5. COMPILER TRANSFORMATIONS FOR LOOP BLOCKING

A blocking transformation partitions the iteration space of a loop-nest. Blocking transformations have been demonstrated to be useful in improving the memory system performance of programs [30],[31], and also used to restructure parallel programs to enhance data locality and reduce interprocessor communication [32],[30]. In compiling parallel programs for multiprocessor systems, blocking transformations can also decrease the run-time synchronization overhead by increasing the granularity of the parallel threads [3].

A blocking transformation that enhances the performance of a loop-nest is applied in three steps. First, dependence analysis determines conditions for preserving the semantics of the loop-nest. Second, a legal blocking transformation is selected to enhance program performance. Finally, the transformation is applied to the program [33]. The area of dependence analysis has been studied extensively in literature, and powerful tools such as the Omega Test [34] can be used to perform the dependence analysis of most programs. This chapter deals mainly with the first and third steps, that is, validity of blocking transformations and restructuring of the application programs. Finally, the impact of blocking transformations on nested parallel loop execution on shared memory multiprocessors is evaluated for three Perfect Club benchmarks.

The class of blocking transformations investigated are multidimensional parallelograms. In the case in which blocking transformations are limited to rectangular shapes, it has been shown that the transformations can be applied when a loop-nest is *fully permutable* [31]. In Section 5.1.2 it is shown that a weaker constraint exists for blocking transformations in which blocks are multidimensional parallelograms. Therefore, unimodular transformations are not necessary to exploit *fully permutable loop-nests* before a blocking transformation [35],[36]. Consider the loop-nest shown in Figure 5.1. When blocking transformations are limited to rectangular shapes, it is necessary to first skew the loop-nest before blocking the iterations in order to preserve the data dependences, (see Figure 5.2). However, by using a parallelogram-shaped block, the loop can be blocked without restructuring as shown in Figure 5.3.

The organization of this chapter is as follows. Section 5.1 provides the basic definitions. In this section, the issues concerning legality of a blocking transformation are also discussed. Section 5.2 presents the methods used in generating index expressions for a blocked loop. In Section 5.3, the effect of blocking transformations on parallel program execution is evaluated. Section 5.4 includes the concluding remarks.

```
for (j= 0 ; j<4 ; j++) {
   for (i=0 ; i<8 ; i++) {
      A(i+1) = 1 / (3 * (A[i] + A[i+1] + A[i+2])) ;
}}
```

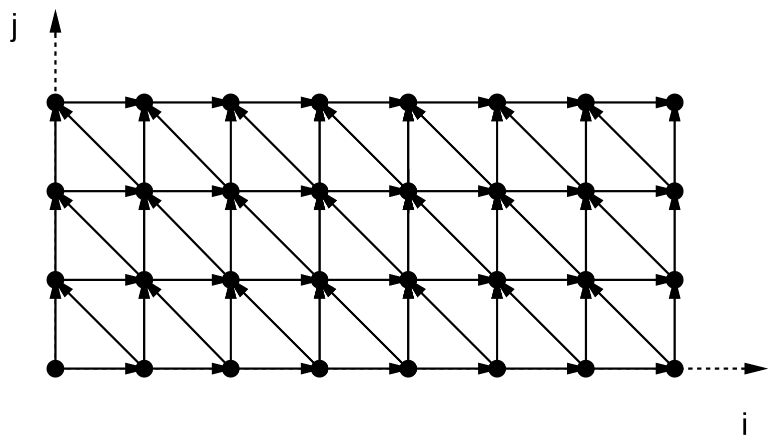

Figure 5.1: Iteration space and dependences of a loop-nest

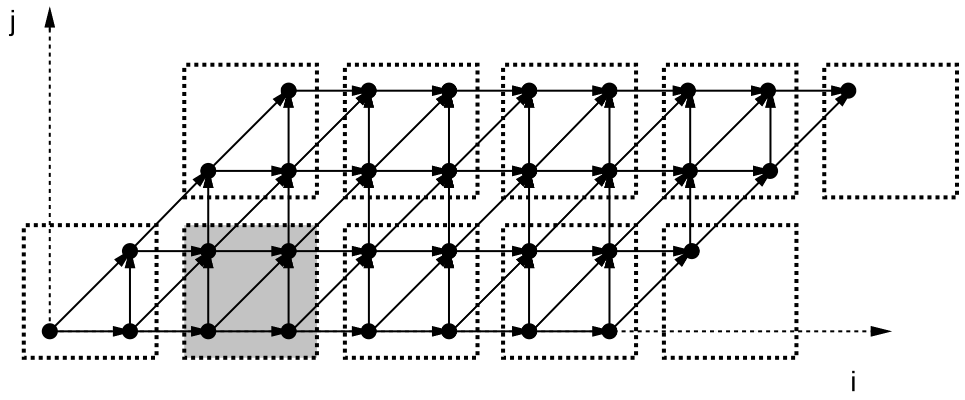

Figure 5.2: Same loop-nest after skewing and blocking

```
for (jj=0 ; j<4 ; jj = jj + 2) {
   for (ii= 0 ; ii<10 ; ii = ii + 2) {
      for (j=max(0,jj) ; j<min(4,jj+2) ; j = j + 1) {
         for (i=max(0,ii+jj-j) ; i<min(8,ii+jj+2) ; i = i + 1) {
            A(i+1) = 1 / (3 * (A[i] + A[i+1] + A[i+2])) ;
}}}}
```

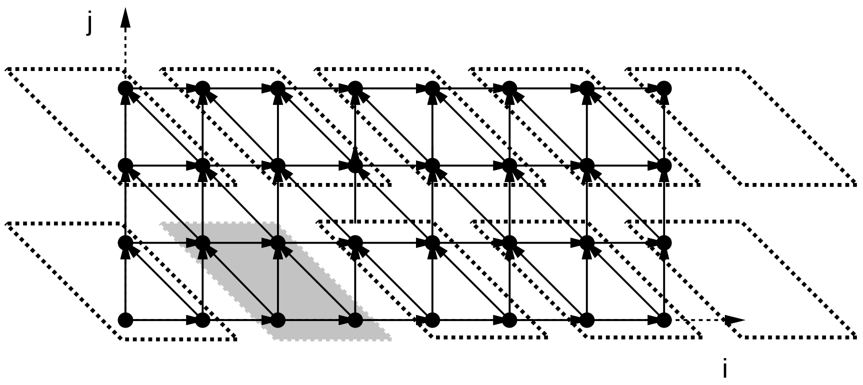

Figure 5.3: Same loop-nest with nonrectangular blocking

## 5.1 Blocking Transformations

This section presents the definitions and the properties of general blocking transformations. The term *loop-nest* is used to designate a loop structure in which there is one and only one loop at every level of nesting. The loop structures for which this condition does not hold can be transformed to a set of loop-nests by applying loop distribution [37].

In this section, a blocking transformation is represented as a set of vectors in the iteration space. This restricts the geometry of blocks to multidimensional parallelograms, which provides a regular tiling of an iteration space. A vector notation has many advantages over the more general and more complex representation using inequalities [33].

First, vector representation allows the use of linear algebra techniques both in the formal treatment of the subject and in the implementation of transformations. Second, as discussed in Section 5.2, the loop transformation techniques developed for partitioning parallel loops can be readily adapted to calculate loop bounds for blocked loops [38]. Last, vector notation allows one to treat dependence vectors and blocking transformations in a uniform manner. The remainder of this section defines blocking transformations using vector representation and discusses the conditions under which a blocking transformation is legal.

**Definition 1** *A vector $\vec{x}$ is lexically positive, denoted as $\vec{x} \succ \vec{0}$, if* $\exists i \ni x_i > 0$ and $\forall j < i,\ x_j = 0$.

**Definition 2** *The vector representation of a blocking transformation on an n-level loop-nest is $B = [\vec{b}_1, \ldots, \vec{b}_n]^T$, where $\vec{b}_i$, $i = 1, \ldots, n$, are n-dimensional, linearly independent and lexically positive vectors.*

Figure 5.4 illustrates the vector representation of a blocking transformation. Each block is defined by two vectors $\vec{b}_1 = [0, 2]^T$ and $\vec{b}_2 = [2, -2]^T$, where the first dimension is j and the second dimension is i. In general, the outermost loop appears as the first dimension of the vector representation. Note also that both vectors are lexically positive. For this loop-nest, the blocking defined by the two vectors is

$$B = \begin{bmatrix} 0 & 2 \\ 2 & -2 \end{bmatrix}$$

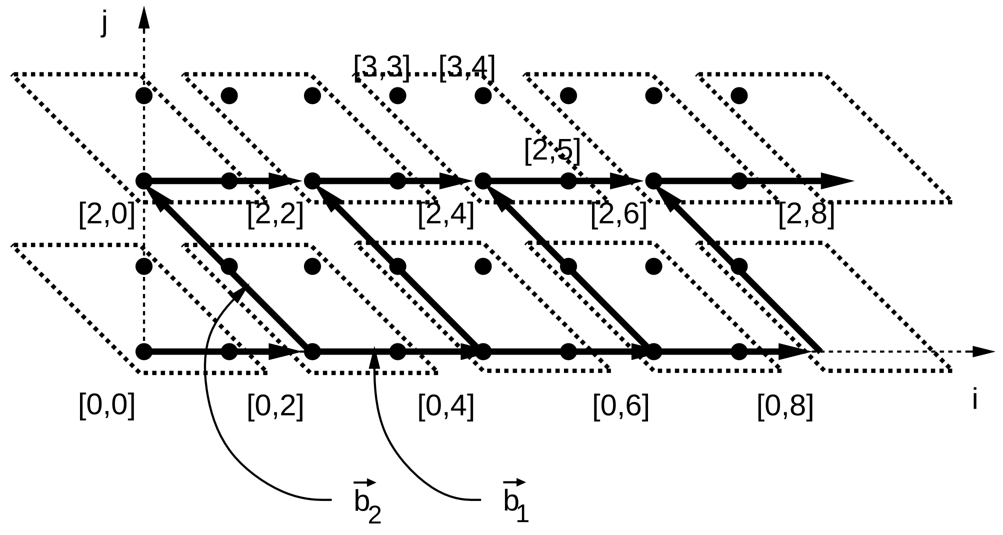

Figure 5.4: The illustration of blocking vectors for the loop-nest in Figure 5.3

The vectors that represent a blocking transformation are referred to as *blocking vectors*. Since blocking vectors are defined to be lexically positive, they have the same properties as those of dependence vectors. As shown in Section 5.2, this allows the use of dependence-vector-based loop partitioning techniques in applying a blocking transformations.

### 5.1.1 Partitioning an iteration space

A blocking transformation partitions the iteration space into a set of blocks in which each block contains a disjoint subset of the iterations. For example, in Figure 5.4, iterations [2,4], [2,5], [3,3], and [3,4] belong to the same block.

Assume that the vector $\vec{b_0}$ is the starting point of the iteration space of a loop-nest. The starting point of any block can be written as $\vec{b_0} + B \times \vec{\alpha}$ where $\vec{\alpha} \in \mathcal{N}^n$. Any point within the block for a given $\vec{\alpha}$ can be expressed as $\vec{b_0} + B \times \vec{\alpha} + B \times \vec{\delta}$, where $\vec{\delta} \in [0,1)^n$.

Therefore, given a blocking $B$, a point $\vec{x} = (x_1, \ldots, x_n)$ in the $n$-dimensional iteration space can be represented as

$$\begin{aligned} \vec{x} &= \vec{b_0} + B \times \vec{\alpha} + B \times \vec{\delta} \\ &= \vec{b_0} + B \times (\vec{\alpha} + \vec{\delta}) \\ &= \vec{b_0} + B \times \vec{a} \end{aligned}$$

where $\vec{a} \in \mathcal{R}^n$. Because B is an invertible matrix, given a $B$, any iteration $\vec{x}$ can be uniquely represented by a vector $\vec{a}$. Furthermore, the vector $\vec{a}$ can be uniquely decomposed into its integral and fractional parts, $\vec{\alpha}$ and $\vec{\delta}$. This decomposition is best illustrated with an example. In Figure 5.4, iteration [3,4] can be expressed as follows:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} = B \times \begin{bmatrix} 3.5 \\ 1.5 \end{bmatrix} \quad \text{hence} \quad \begin{bmatrix} 3 \\ 4 \end{bmatrix} = B \times \begin{bmatrix} 3 \\ 1 \end{bmatrix} + B \times \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

where $\vec{\alpha} = [3, 1]^T$, $\vec{\delta} = [0.5, 0.5]^T$, and $\vec{a} = [3.5, 1.5]^T$.

When applied to an $n$-level loop-nest, a blocking transformation generates a new $2n$-level loop-nest. The outer n levels sequence the execution through blocks, whereas the inner n levels cover the iterations within a block. Therefore, after a blocking transformation, the original $n$-dimensional iteration space becomes a $2n$-dimensional space. The following definition specifies how each point is mapped from one space to the other.

**Definition 3** *Let $\vec{x}$ be an iteration in an $n$-dimensional iteration space and $B$ be a blocking with the starting point $\vec{b_0}$. Furthermore, let $\vec{x} = \vec{b_0} + B \times \vec{\alpha} + B \times \vec{\delta}$ where*

*$\vec{\alpha} \in \mathcal{N}^n$ and $\vec{\delta} \in [0,1)^n$. The blocking operator $\mathcal{B} : \mathcal{N}^n \to \mathcal{N}^{2n}$ maps iteration $\vec{x}$ to an iteration $\vec{y} \in \mathcal{N}^{2n}$ and is defined as follows:*

$$\mathcal{B}(\vec{x}) = \{\vec{y} \mid \vec{y} = [y_1, \ldots, y_{2n}] \text{ where } [y_1, \ldots, y_n] = B \times \vec{\alpha} \text{ and } [y_{n+1}, \ldots, y_{2n}] = B \times \vec{\delta}\}.$$

Consider iteration [3,4] in Figure 5.4; its coordinate can be decomposed as follows:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where [2,4] is the starting coordinate of the block that contains [3,4] and [1,0] is the offset of [3,4] within the block. According to Definition 3, the coordinate of this iteration after blocking is [2,4,1,0].

### 5.1.2 Legality of a blocking transformation

By executing all iterations of a block before staring the next block, a blocking transformation modifies the order in which iterations are executed in the original loop-nest. Within each block, the algorithm presented in Section 5.2 guarantees that the iterations in a block are executed according to their original lexical ordering [33]. Furthermore, the starting points of blocks are traversed in their lexical order [38]. However, the lexical ordering between two points across blocks is not preserved in general. In the presence of data dependences across iterations, such reordering must not violate the order required by data dependences for correct execution.

**Definition 4** *A blocking transformation is legal if the transformed dependence vectors are lexically positive.*

Definition 4 is based on the fact that if all of the dependence vectors remain lexically positive after the transformation, the source point of each dependence vector always executes before its destination. The following theorem provides a practical test of whether a blocking transformation is legal.

**Theorem 1** *Let $D$ be the set of dependence vectors of a loop nest. A blocking transformation $B$ is legal if $\forall \vec{d} \in D\ \exists \vec{d}_a \ \ni \vec{d} = B \times \vec{d}_a$ where $\vec{d}_a \in \mathcal{R}^n$ and $d_{a_i} \geq 0$ for $i = 1, \ldots, n$.*

*Sketch of proof:* Let $\vec{p}$ and $\vec{q}$ be two points in the iteration space where $\vec{q} - \vec{p} = \vec{d}$, i.e., $\vec{q}$ depends on $\vec{p}$. Let $\vec{p'} = \mathcal{B}(\vec{p})$ and $\vec{q'} = \mathcal{B}(\vec{q})$. It has to be proven that $\vec{q'} - \vec{p'} \succ \vec{0}$, i.e., after the blocking transformation, iteration $\vec{p'}$ is executed before iteration $\vec{q'}$.

In the case in which $\vec{p'}$ and $\vec{q'}$ are in the same block, $\vec{q'}$ is executed after $\vec{p'}$. In the case that they are not in the same block let $\vec{p} = B \times \vec{p}_a$ and $\vec{q} = B \times \vec{q}_a$ where $\vec{p}_a \in \mathcal{R}^n$ and $\vec{q}_a \in \mathcal{R}^n$.

$$\begin{aligned} \vec{d} &= \vec{q} - \vec{p} \\ B \times \vec{d}_a &= B \times \vec{q}_a - B \times \vec{p}_a \\ B \times \vec{d}_a &= B \times (\vec{q}_a - \vec{p}_a) \\ \vec{d}_a &= (\vec{q}_a - \vec{p}_a); \end{aligned}$$

$d_{a_i} \geq 0$ for $i = 1, \ldots, n$ therefore $q_{a_i} \geq p_{a_i}$. From the definition of the $\mathcal{B}$ operator, $p'_i = integer(p_{a_i})$ and $q'_i = integer(q_{a_i})\ \forall\ i = 1, \ldots, n$. Therefore, $q'_i \geq p'_i\ \forall\ i = 1, \ldots, n$. Since $\vec{p'}$ and $\vec{q'}$ are not within the same block there is at least one $i$, $i = 1, \ldots, n$, for which $q'_i > p'_i$. Therefore $\vec{q'} - \vec{p'} \succ \vec{0}$, i.e., iteration $\vec{q'}$ is executed after $\vec{p'}$ in the blocked loop-nest.

Theorem 1 states a sufficient condition for a given blocking transformation to be legal: all dependence vectors of the loop-nest should be a positive linear combination of all blocking vectors. Geometrical interpretation of the theorem is that the dependence vectors should lie in the solid angle defined by the blocking vectors.

Previous works on blocking transformations have focused mainly on rectangular blocks. In the presence of data dependences, it has been necessary to perform unimodular transformations to make all dependence vectors nonnegative in all dimensions, i.e., making the loop-nests fully permutable. In this special case in which only rectangular blocks are used, Theorem 1 can be simplified into the following corollary:

**Corollary 1** *Let $D$ be the set of dependence vectors of a loop-nest. A rectangular blocking transformation $B$ is valid if the loop-nest is fully permutable.*

When a blocking transformation is not required to be rectangular, it is no longer necessary to generate a fully permutable loop-nest in order to perform blocking.

## 5.2 Loop Bounds Generation

A compiler-based blocking transformation has to transform the original $n$-level loop-nest to a $2n$-level loop-nest with correct loop bounds. Determining the loop bounds for the outermost $n$ levels involves generating the starting indices for the blocks in the original loop-nest order. Loop bounds for the innermost $n$ levels of loops are selected such that the indices for the iterations within a block are generated in the original loop-nest order.

The index generation problem for the outermost $n$ levels of loops is very similar to the loop bound generation problem for parallel loop partitioning discussed in [38]. The *Minimum Distance* method [38] for generating loop indices for partitioned loops is directly applicable to the index generation problem for the starting points of loop blocks with a minor modification. The *Minimum Distance* method generates indices for the blocks whose indices are in the index space of the original loop-nest. However, a blocking transformation can generate blocks which contain iterations of the original loop-nest where the first iteration of the block lie outside the original loop-nest. Expanding the iteration space in all dimensions to include all such blocks for the outermost $n$ levels of the transformed loop-nest corrects this problem. By extending the bounds of the original loop-nest in direction $j$ by $\Delta_j$, where $\Delta_j = max(|b_{i,j}|)$ for $i = 1, \ldots, n$, where $b_{i,j}$ is the $j$th element of the $i$th blocking vector, the *Minimum Distance* method can be used to generate indices for all blocks. For a complete discussion of the *Minimum Distance* and computation of the actual loop bounds, the reader is directed to [38] and [39].

To generate the loop bounds for the innermost $n$ levels of the transformed loop, the hyperplane method is used. In [33], a discussion of this method for general polyhedra can be found. When the geometry of the blocks is limited to $n$-dimensional parallelograms, the hyperplane method gives satisfactory results for a small number (2-3) of dimensions.

The loop bound generation can be illustrated for the blocked loop shown in Figure 5.3. The blocking vectors, for this loop are

$$\vec{b}_1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \vec{b}_2 = \begin{bmatrix} 2 \\ -2 \end{bmatrix}, \text{ therefore } \quad B = \begin{bmatrix} 0 & 2 \\ 2 & -2 \end{bmatrix}$$

and with the *Minimum Distance* method, the *shifting distances* for the outermost two levels of loops are given by

$$\begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix}$$

where the diagonal entries define the increments of the two outermost loops [38]. The *shifting distance* matrix is an upper diagonal normal form for the blocking description $B$. It provides the set of parameters in the index expressions of the outermost $n$ levels of loops to generate the indices for blocks in the lexical order determined by the original loop-nest [38]. Using this representation, the loop

```
for (j= 0 ; j<4 ; j++) {
    for (i=0 ; i<8 ; i++) {
        A(i+1) = 1 / (3 * (A[i] + A[i+1] + A[i+2])) ;
}}
```

is transformed into

```
for (jj=0 - Δ_j ; j<4 + Δ_j ; jj = jj + 2) {
    for (ii= 0 - Δ_i ; ii<8 + Δ_i ; ii = ii + 2) {
        for (j= LBj ; j<UBj ; j = j + 1) {
```

```
        for (i=LBi ; i<UBi ; i = i + 1) {
            A(i+1) = 1 / (3 * (A[i] + A[i+1] + A[i+2])) ;
}}}}
```

From the blocking vectors, it can be determined that $\Delta_j = 2$ and $\Delta_i = 2$. This concludes the bound calculation for the 2 outermost loops.

Using the hyperplane method, for a given block with coordinates $jj$ and $ii$, the inequalities for the iteration points within this block can be written as

j $\geq$ max(0,min(jj, jj + $b_{1,1}$, jj + $b_{2,1}$))

j $<$ min(4,max(jj, jj + $b_{1,1}$, jj + $b_{2,1}$))

and by substituting the $b_{1,1}$ and $b_{2,1}$ values from the blocking vectors $\vec{b}_1$ and $\vec{b}_2$ in the example loop, these inequalities can be reduced to

j $\geq$ max(0,jj)

j $<$ min(4,jj+2).

Similarly, the bounds for the innermost loop can be calculated as

i $\geq$ ii + jj - j

i $<$ ii + jj + $b_{1,2}$.

After the generation of the loop bounds expressions for the innermost 2 loops, the final form of the blocked loop can be generated as

```
for (jj=-2 ; j<6 ; jj = jj + 2) {
    for (ii= -2 ; ii<10 ; ii = ii + 2) {
        for (j=max(0,jj) ; j<min(4,jj+2) ; j = j + 1) {
            for (i=max(0,ii+jj-j) ; i<min(8,ii+jj+2) ; i = i + 1) {
                A(i+1) = 1 / (3 * (A[i] + A[i+1] + A[i+2])) ;
}}}}
```

One disadvantage of the hyperplane method is that, when one or more blocking vectors are aligned with the axis of the iteration space, e.g., a rectangular blocking, there can be redundancy in the loop bound expressions. In the example above, the general expression in two dimensions for the lower bound of the `j` loop

$j \geq \max(0, \min(jj, jj + \vec{b}_1[1], jj + \vec{b}_2[1]))$

can be simply reduced to

$j \geq \max(0, jj)$

because vectors $\vec{b}_1$ and $\vec{b}_2$ are lexically positive, i.e., $b_{1,1} \geq 0$ and $b_{2,1} \geq 0$. In the implementation of the bounds calculation for the innermost $n$ levels of the loop, a template-based approach is used. For the cases in which $n < 4$, depending on the direction of the blocking vectors, an optimized template for index expressions is selected.

## 5.3 Experimental Results

This section presents simulation results on the effectiveness of blocking transformations in reducing iteration scheduling overhead for nested parallel loops. The simulations use the shared memory multiprocessor model described in Chapter 2. The three parallelized Perfect Club benchmarks which contain nested parallel loops, FLO52, ADM, and DYFESM, are used for measurements. Blocking parallel loops of a program changes three characteristics of a parallel application, namely, loop granularity, task granularity, and parallelism. After blocking parallel loops, both loop granularity and task granularity are increased at the expense of available parallelism.

Grouping several parallel iterations for sequential execution increases the loop granularity of the application. Increasing granularity results in a less frequent need for iteration scheduling, thus lowering iteration scheduling overhead. However, this performance increase comes at the cost of parallelism. Since after the blocking transformation the number of parallel iterations within a loop-nest decreases, this can result in lower utilization of processors in the system. Figures 5.5 and 5.6 illustrate how loop granularity and also task granularity increases as a result of the blocking transformation. The execution of the two-level nested loop in Figure 5.5 results in 17 parallel loop executions, one outer loop and 16 inner loops. This loop is transformed into the parallel loop in Figure 5.6. The execution of the blocked loop results in executing five parallel loops, one outer loop and four inner loops. After blocking, each inner parallel loop contains four times the

```
      DOALL 10 I=1,16
      DOALL 20 J=1,16
      .
      .
      .
 10   CONTINUE
 20   CONTINUE
```

Figure 5.5: A two-level nested DOALL loop

```
      DOALL 10 I=1,16,4
      DOALL 20 J=1,16,4
      DO 30 II=I,I+3
      DO 30 JJ=J,J+3
      .
      .
      .
 30   CONTINUE
 10   CONTINUE
 20   CONTINUE
```

Figure 5.6: Nested DOALL loop after blocking transformation

computation of the original inner parallel loops. As the result of the transformation, the iteration granularity is increased 16 times.

Figures 5.7-5.10 illustrate the effect of blocking transformations on the FLO52 program. The x-axis is the blocking factor. A blocking factor of $n$ results in blocking single-level parallel loops by $n$, two-level parallel loops by $n \times n$, etc. Formally, a blocking factor of $n$ corresponds to a blocking $B$, as in Definition 2, where $B$ is diagonal and all diagonal entries are $n$. For example, the blocked loop in Figure 5.6 has a blocking factor of 4.

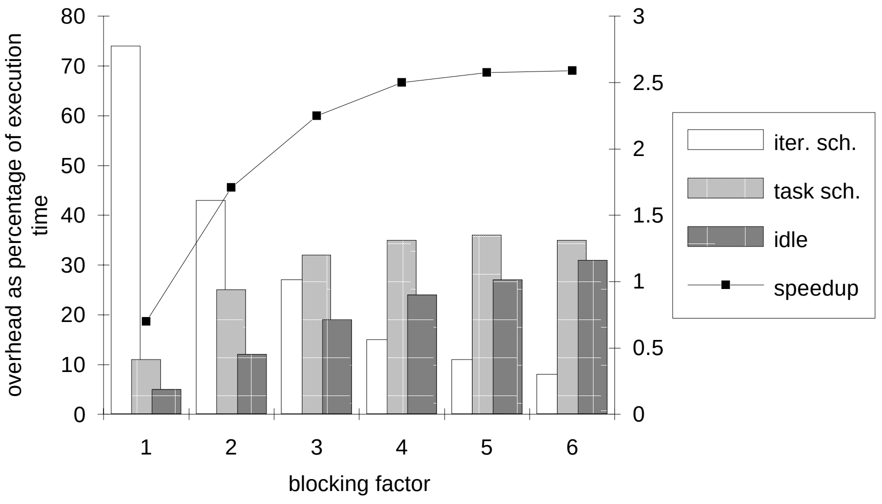

Figure 5.7: Performance of FLO52 for 16 processors and *test&set* primitive

For each blocking factor, the three bars show the percentage of program execution time spent in scheduling iterations, scheduling parallel tasks, and the idle time for processors. The numbers presented are obtained by averaging the respective times over all of the processors within the multiprocessor. The axis for the bars are on the left-hand side of the charts. The lines in the figures show the speedup obtained from parallel execution of the program. The base number for calculating the speedup is the execution time of the sequential version of the program on a single-processor system. The axis for the speedup figures is shown on the right-hand side of the figures.

Figures 5.7 and 5.8 show how the performance of FLO52 improves due to the decrease in iteration scheduling overhead. The primitives used in these experiments, *test&set* and exchange-byte, can not support iteration scheduling operations as fast as the *fetch&add* primitive. The architectures supporting these primitives have the most to benefit from

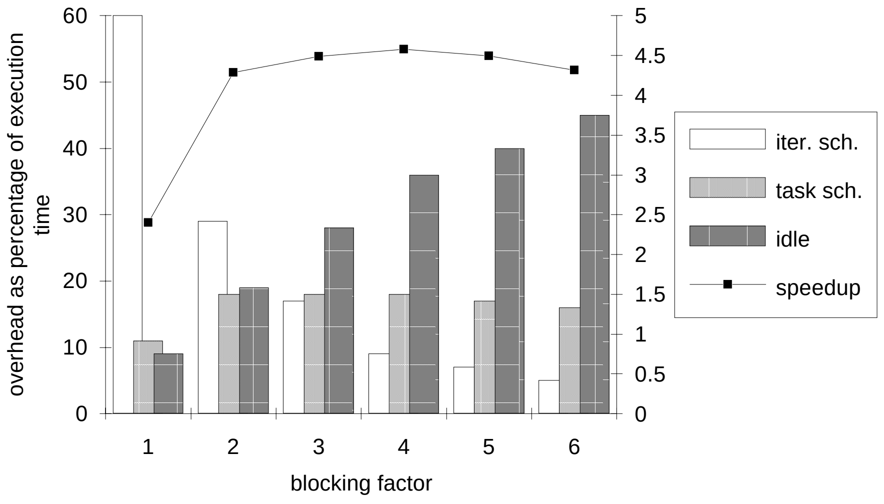

Figure 5.8: Performance of FLO52 for 16 processors and *exchange-byte* primitive

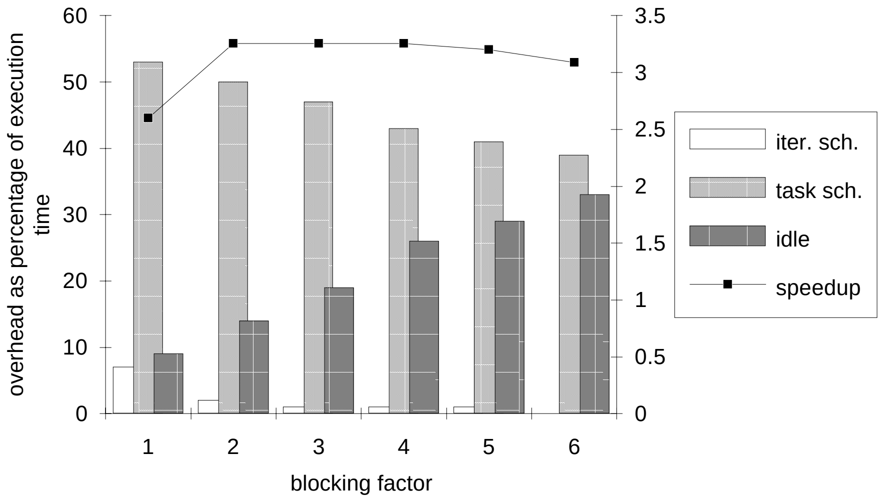

Figure 5.9: Performance of FLO52 for 16 processors and *fetch&add* primitive

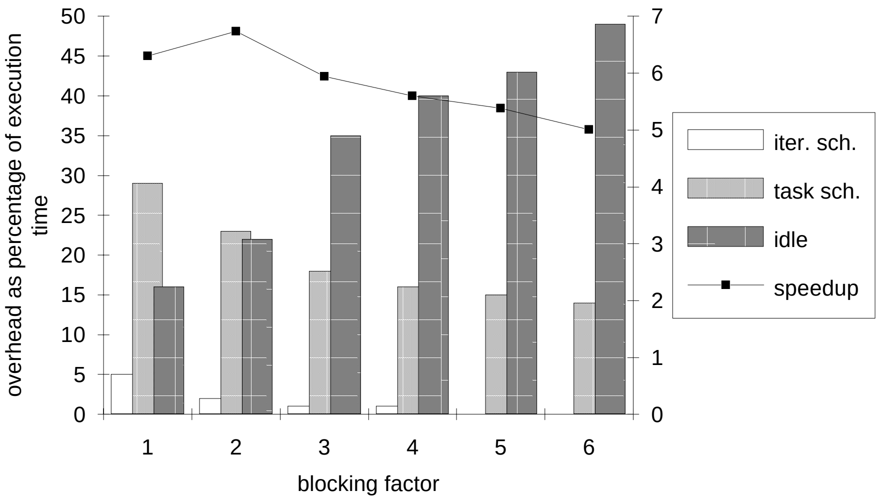

Figure 5.10: Performance of FLO52 for 16 processors and synchronization bus

loop blocking. With the aid of the blocking transformation, the speedup for FLO52 increases 3.7 times from 0.7 to 2.6 for the *test&set* case. Similarly, a factor of 2 increase in speedup is obtained for the exchange-byte case. However, for the exchange-byte primitive, increasing the blocking factor beyond 3 results in a slight decrease in the obtained speedup. This results from the loss of parallelism in the program for large blocking factors. This also explains the continuing increase in processor idle time for large blocking factors as shown in Figure 5.8.

Figures 5.9 and 5.10 show the effect of the blocking transformation for architectures which support the *fetch&add* primitive. In these cases, for the blocking factor of two, there is a slight increase in performance which can be attributed to the reductions in both iteration scheduling overhead and task scheduling overhead. However, for larger blocking factors, the loss of parallelism has a negative effect on performance. Although the task scheduling overhead continues to drop as the blocking factor increases, the idle time of processors increases, negating the effect of this decrease and worsening the overall performance.

The effect of the blocking transformation on the DYFESM program is shown in Figures 5.11- 5.14. Blocking transformation improves the performance of the program significantly for the *test&set* primitive. The performance increase for this case is the result of a decrease in iteration scheduling overhead. For the exchange-byte primitve, small blocking factors result in a slight increase in performance (see Figure 5.12). However, as the blocking factor increases beyond 3, the loss of parallelism in the program results in performance degradation. As shown in Figures 5.13 and 5.14, there is very little performance improvement in the presence of efficient architectural support for synchronization. For these two cases, as the blocking factor increases, the speedup figures go down.

Figures 5.15-5.18 present the simulation results for the ADM program. Of the three benchmarks, ADM has the lowest level of parallelism and the lowest task granularity. As shown in Figure 5.15, although blocking parallel loops improves the performance of ADM when the *test&set* primitive is supported, the overall speedup remains below one. A similar performance increase due to the increased loop granularity can be observed for the exchange-byte case in Figure 5.16. For the two architectures in which *fetch&add* operation is supported, there is either no performance improvement or a performance degradation. As illustrated by the increasing idle time percentages in Figures 5.17 and 5.18, this is caused by the decrease in parallelism of the application due to blocking.

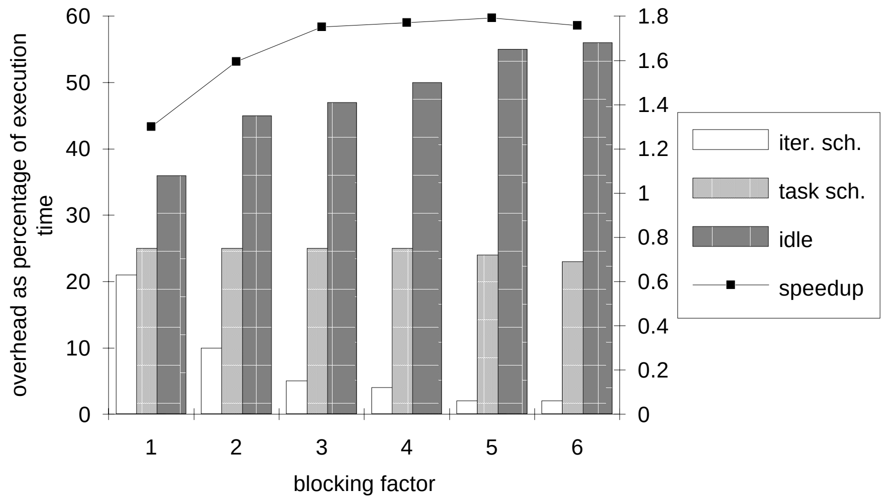

Figure 5.11: Performance of DYFESM for 16 processors and *test&set* primitive

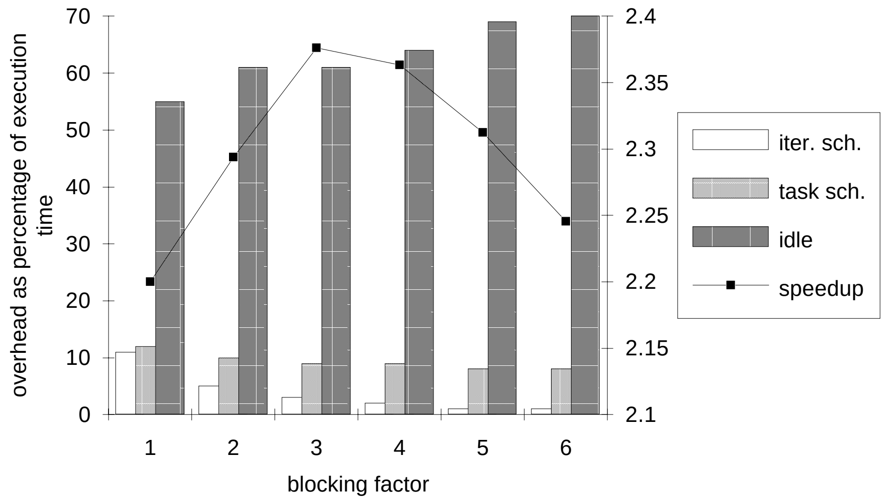

Figure 5.12: Performance of DYFESM for 16 processors and *exchange-byte* primitive

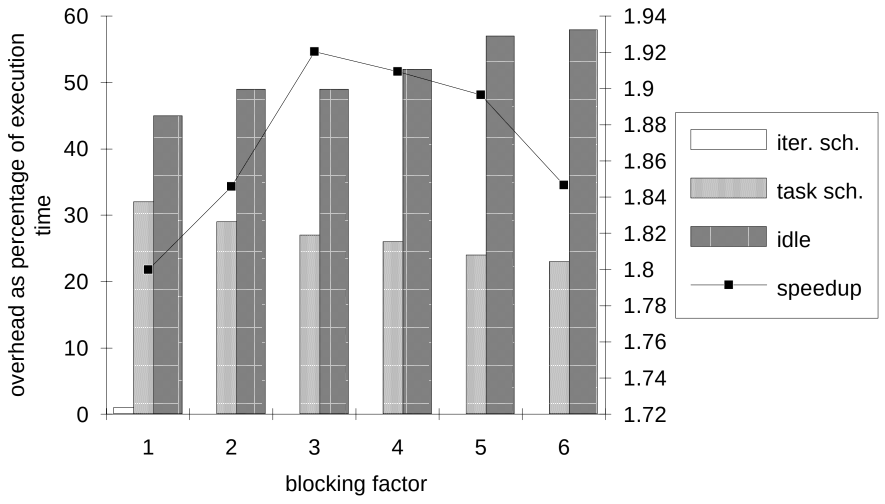

Figure 5.13: Performance of DYFESM for 16 processors and *fetch&add* primitive

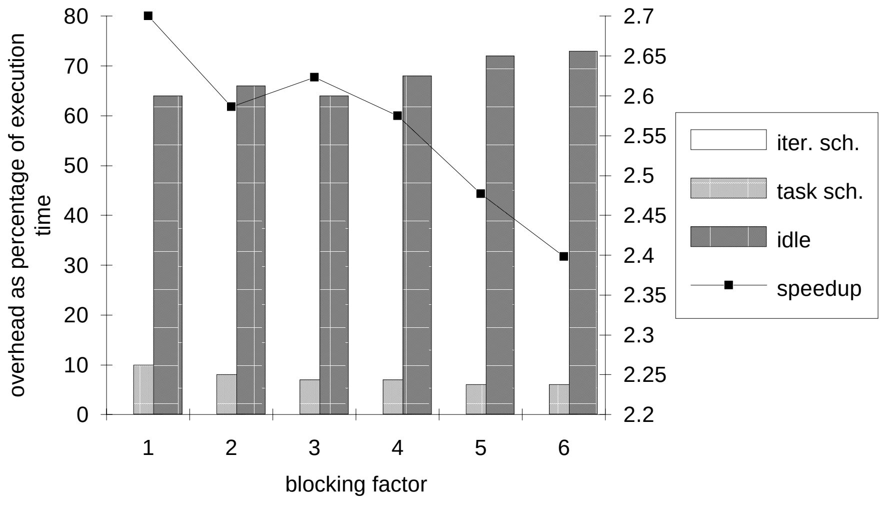

Figure 5.14: Performance of DYFESM for 16 processors and synchronization bus

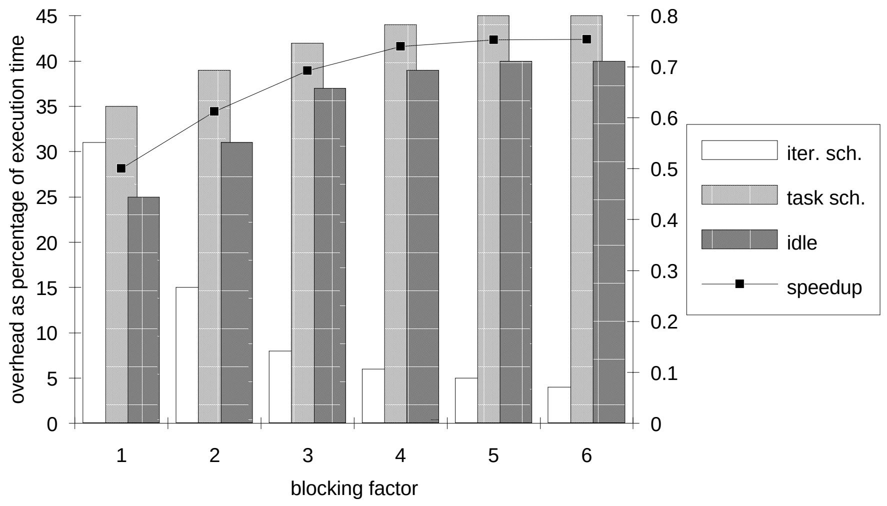

Figure 5.15: Performance of ADM for 16 processors and *test&set* primitive

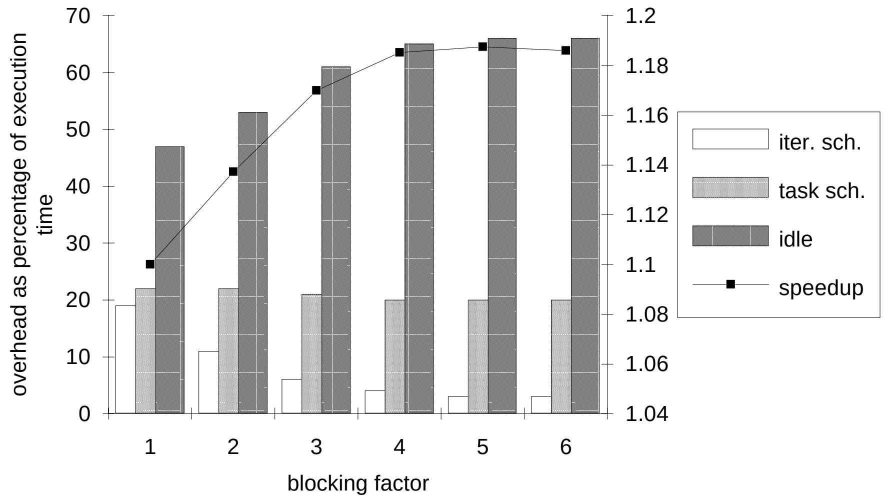

Figure 5.16: Performance of ADM for 16 processors and *exchange-byte* primitive

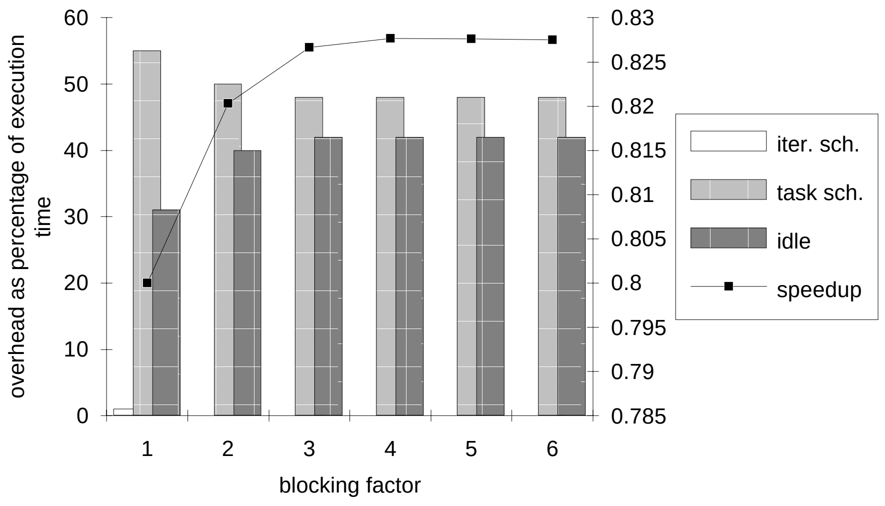

Figure 5.17: Performance of ADM for 16 processors and *fetch&add* primitive

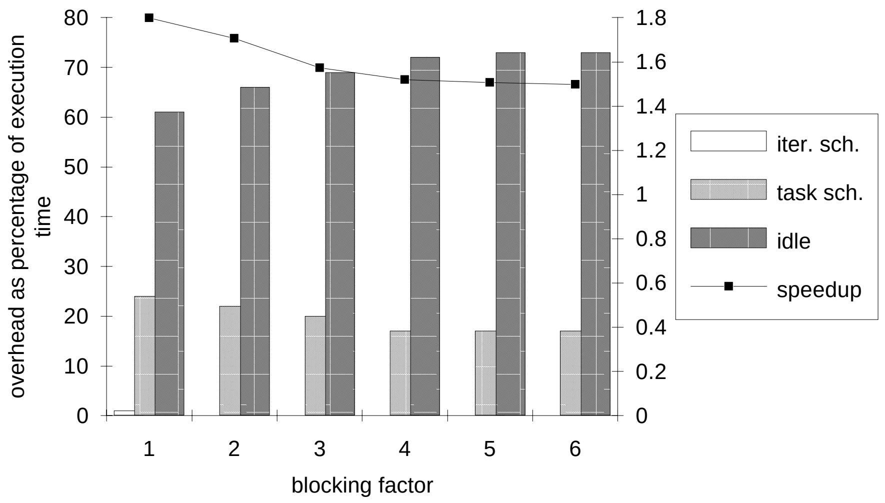

Figure 5.18: Performance of ADM for 16 processors and synchronization bus

## 5.4 Concluding Remarks

A blocking transformation can be used to improve the performance of both parallel and sequential programs. This chapter presents a formal treatment of blocking transformations for nested loop structures. In the presence of loop-carried dependences within a loop-nest, blocking transformations must preserve the dependences. In this chapter a sufficient condition for the validity of the blocking transformations was presented. Furthermore, issues concerning code transformation and loop bound calculation for applying blocking transformations were discussed in detail.

It is shown that blocking transformations can be used to decrease the iteration scheduling overhead in nested parallel loop execution. Blocking a parallel loop-nest has mixed effects on program performance. Blocking a loop-nest reduces the available parallelism within the loop-nest. In the cases in which a multiprocessor can efficiently exploit the parallelism, e.g., a multiprocessor with a synchronization bus, the overall program performance can degrade after applying a blocking transformation. However, for multiprocessors in which there is no support for an atomic *fetch&add* primitive, blocking greatly reduces the iteration scheduling overhead resulting in significant performance increase. For these cases, since the parallelism in the application was not being exploited efficiently, the decrease in parallelism is not visible for small blocking factors.

Blocking transformations change three characteristics of a program: loop granularity, task granularity and parallelism. Since most parallel FORTRAN programs exploit parallelism in the data structures, the task granularity and the parallelism within these applications increase with increasing input data sets. However, loop granularity is determined by the loop structures and does not depend on data set size. Therefore, for applications processing large input data sets, the blocking transformation can increase overall program performance with minimal negative effects due to the associated decrease in parallelism.

# 6. CONCLUSIONS

Numerical applications require large amounts of computing power. Although shared memory multiprocessors provide a cost-effective platform for parallel execution of numerical programs, parallel processing has not delivered the expected performance on these machines. There are several reasons for this. Compiler technology for automatic parallelization of programs is still developing. A parallelizing compiler can fail to identify the parallelism in a program. Furthermore, technologies required for parallelization, e.g., dependence analysis, focus on nested loop structures and individual functions resulting in relatively small units of parallel computation. During parallel program execution, the overhead introduced by processor synchronization and dynamic scheduling of computation to processors is inversely related to the size of parallel computation units. As parallel loops become smaller, both in the number of iterations and in the number of instructions within an iteration, the scheduling and synchronization operations are used more frequently to control the start and termination of computation. This thesis evaluates existing techniques and explores alternatives in the areas of architectural support for parallel execution, synchronization and scheduling algorithms, and compiler transformations for modifying parallel application characteristics.

In Chapter 1 the structure of parallel FORTRAN programs is explained in detail. This chapter provides a comprehensive explanation of which type of parallel constructs require what kind of scheduling and synchronization support. The implementation details of synchronization algorithms with different architectural primitives are also presented. Chapter 2 describes a trace-driven shared memory multiprocessor simulation platform. The simulation tool emulates the execution of different scheduling and synchronization algorithms and supports the *test&set*, exchange-byte, *fetch&add* primitives together with a synchronization bus.

The performance of four parallelized Perfect Club benchmarks, BDNA, FLO52, DYFESM, and ADM, is evaluated by using the developed simulation tool. The execution behavior of these programs is analyzed in Chapter 3. It is shown that both iteration scheduling and parallel task scheduling can be a significant source of overhead. For architectures in which an atomic *fetch&add* instruction is not supported, the execution of both the iteration scheduling algorithm and the task scheduling algorithm is a bottleneck and degrade the program performance. When the *fetch&add* primitive is supported by the architecture, the iteration scheduling overhead decreases but the task scheduling overhead remains significant. It is also shown that an architecture supporting the *fetch&add* primitive can achieve the performance of an architecture supporting a simple synchronization bus in executing these benchmarks.

Chapter 4 introduces a distributed task scheduling algorithm which is designed for scheduling parallel tasks. The proposed algorithm allows processors to acquire tasks from multiple queues and relies on nonblocking lock operations for correct operation. To support this algorithm, a nonblocking queuing lock algorithm is presented. The experimental results presented in this chapter show that the performance of programs containing nested parallel loops can be improved up to 50% with the use of a distributed task scheduling algorithm.

As stated in Chapter 3, the amount of overhead introduced by scheduling algorithms is a function of both the architectural primitives and synchronization algorithms, and the iteration and task granularity of applications. In Chapter 5, blocking transformations are used to modify program characteristics at compile time. The blocking transformation groups iterations of a parallel loop-nest together into a sequential thread resulting in an increase in both loop granularity and task granularity. In this chapter, first a formal representation for blocking transformations is introduced and conditions of validity for a transformation are defined. It has been shown that the changes to the source code required by a blocking transformation are identical to those of a parallel loop partitioning; therefore, the code transformation techniques described in [21] can be used.

The experiments whose results are presented in Chapter 3 show that blocking transformations can be very effectively used to enhance the granularity characteristics of parallel programs. For architectures which do not support synchronization busses or *fetch&add* primitives, even blocking a small number of iterations of a loop-nest together improves performance significantly. It is shown that the performance of a blocked program using the exchange-byte primitive can match the performance of an architecture supporting the *fetch&add* primitive. For architectures supporting a synchronization bus or the *fetch&add* primitive, there is very little gain in performance as a result of blocking transformations. For large block sizes, the loss of parallelism in the application due to blocking results in less efficient use of processors, hence performance degradation.

It is shown in this thesis that architectural support for parallel processing, run-time algorithms for scheduling and synchronization and the application characteristics interact in a complex pattern. A shared memory multiprocessor supporting a synchronization bus with the *fetch&add* primitive can execute automatically parallelized application efficiently. The performance of an application executing on such an architecture is limited by the amount of parallelism in an application.

An architecture supporting *fetch&add* in shared memory can suffer from task scheduling overhead in executing nested parallel loops. For this architecture, the performance of parallel programs can be improved by using distributed task scheduling algorithms. For the architectures supporting *test&set* or exchange-byte primitives, compiler restructuring of loop-nests is necessary to achieve performance to justify parallel execution.

Blocking transformation is a powerful means of modifying program characteristics. The restructuring of nested loop structures makes it possible to enhance the behavior of an application. These uses can range from reducing interprocessor communication in message passing parallel programs to increasing the utilization of caches or other components of memory hierarchy in sequential programs. Such uses of blocking transformations are a current area of research. Blocking transformations also result in well-structured inner loops of a known number of iterations. This property allows the accurate prediction of resource requirements of loop-nests which can be utilized by superscalar/VLIW compilers for code optimizations.

## REFERENCES

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

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

[3] D. Chen, H. Su, and P. Yew, "The impact of synchronization and granularity on parallel systems," Tech. Rep. CSRD Rpt. no. 942, Center for Supercomputing Research and Development, University of Illinois, 1989.

[4] P. A. Emrath, D. A. Padua, and P. Yew, "Cedar architecture and its software," *Proceedings of 22nd Hawaii International Conference on System Sciences*, vol. 1, pp. 306–315, 1989.

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

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

[7] Sequent Corp., *Balance(tm) 8000 Guide to Parallel Programming*, July 1985.

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

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

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

[11] G. S. Sohi, J. E. Smith, and J. R. Goodman, "Restricted fetch&$\phi$ operations for parallel processing," *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 410–416, 1989.

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

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

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

[15] E. D. B. III, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 295–307, 1986.

[16] 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*, vol. 2, pp. 200–204.

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

[18] N. S. Arenstorf and H. F. Jordan, "Comparing barrier algorithms," *Parallel Computing*, no. 12, pp. 157–170, 1989.

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

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

[21] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, pp. 836–845, 1986.

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

[23] C. D. Polychronopoulos, "Multiprocessing versus multiprogramming," *Proceedings of the 1989 International Conference on Parallel Processing*, vol. 2, pp. 223–230, 1989.

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

[25] H. F. Jordan, "Hep architecture, programming and performance," in *Parallel MIMD Computation: HEP Supercomputer and Its Applications,* J. S. Kowalik, Ed. Cambridge: MIT Press, 1985, pp. 1–40.

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

[27] H. F. Jordan, "Interpreting parallel processor performance measurements," *SIAM Journal of Scientific and Statatistical Computing*, vol. 8, no. 2, pp. s220–s226, March 1987.

[28] G. Cybenko, J. Bruner, S. Ho, and S. Sharma, "Parallel computing and the perfect benchmarks," Tech. Rep. CSRD Rpt. no. 1191, Center for Supercomputing Research and Development, University of Illinois, 1991.

[29] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua, "Experience in the automatic parallelization of four perfect benchmark programs," Tech. Rep. CSRD Rpt. no. 827, Center for Supercomputing Research and Development, University of Illinois, 1991.

[30] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputing applications," Ph.D. Dissertation, Department of Computer Science, Rice University, May 1989.

[31] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.

[32] S. G. Abraham and D. E. Hudak, "Compile-time partitioning of iterative parallel loops to reduce cache coherence traffic," *Journal of Parallel and Distributed Computing*, vol. 2, pp. 318–328, 1991.

[33] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," *Proceedings of the 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pp. 39–50, 1991.

[34] W. Pugh, "The Omega test: A fast and practical integer programming algorithm for dependence analysis," *Proceedings of Supercomputing '91,* 1991.

[35] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, October 1991.

[36] U. Banerjee, "Unimodular transformations of double loops," *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pp. 192–219, California, 1990.

[37] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1200, December 1986.

[38] J. Peir and R. Cytron, “Minimum distance: A method for partitioning recurrences for multiprocessors,” *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 217–225.

[39] C. King and L. M. Ni, “Grouping in nested loops for parallel execution on multicomputers,” *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 31–38.

# VITA

Sadun Anik received the B.S. degree in electrical and electronics engineering from the Middle East Technical University, Ankara, in 1986, and the M.S. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1990. His research interests are in parallel processing, computer architecture with emphasis an on performance considerations in parallel program execution. He is joining Applied Vision in San Mateo, California, in September 1993.