Run-time support for parallel programs

Clapp, Russell Mace, Ph.D.

The University of Michigan, 1991

RUN-TIME SUPPORT FOR PARALLEL PROGRAMS

by

Russell Mace Clapp

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1991

It was midnight, midnight at noon
Everyone talked in rhyme
Everyone saw the big clock ticking
Nobody knew, nobody knew the time

Elegant debutantes smiled
Everyone fought for dimes
Newspapers screamed for blood
It was the best of times

Every place around the world it seems the same
Can't hear the rhythm for the drums
Everybody wants to look the other way
When something wicked this way comes

Sometimes they tie a thief to the tree
Sometimes I stare
Sometimes it's me

Everyone told the truth
All that we heard were lies
A pope claimed that he'd been wrong in the past
This was a big surprise

Everyone fell in love
A cardinal's wife was jailed
The government saved a dying planet
When popular icons failed

Every place around the world it seems the same
Can't hear the rhythm for the drums
Everybody wants to look the other way
When something wicked this way comes

Sometimes they tie a thief to the tree
Sometimes I stare
Sometimes it's me
Sometimes I stare
Sometimes it's me

–Sting

For Greta

ACKNOWLEDGEMENTS

Thanks to the following:

Trev for six long years of funding, and a careful review of my work,

Kunle for exchanging ideas, but especially for the tunes,

Buz for advice,

Dick for getting me started,

Jim S. for a new direction, and ACA for a ZS to play with,

Jarir for parties,

Dave (David) for PC software and instructions on how to use it,

Don for Unix help,

Chuck A. for more Unix help,

Joe for PC help, and pounding a few at Casey's

Paul for Apollo help, and being a "normal" office mate,

Ticky for giving me a job, and making sure I finished,

Rob and Al for tools, parts, and unlocking doors,

my committee for their time,

Dawn, Karen, Dolores, and Virginia for paperwork,

Chris for help in the early days,

Chuck J. for being himself,

my friends and family for not losing faith,

Mom and Dad for moral support,

and finally Greta, for putting up with it all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

The advent of the microprocessor has encouraged computer architects to design multiple processor computers. These machines, called *multiprocessors*, increase system throughput by running multiple programs simultaneously, or by decreasing the running time of each program by employing *parallel processing*. In order to improve the applicability of parallel processing, computer scientists have developed languages and compilers for *parallel programming*. However, practical parallel programming systems are still in a state of infancy, and their performance leaves considerable room for improvement. Many commercial systems incorporate ad hoc techniques for expressing parallelism, most involving machine dependencies [81, 19, 82, 80]. Programming languages for parallel systems have ranged from adapting sequential languages for parallel execution to the use of explicitly parallel concurrent languages such as Ada. Some success has been achieved in implementing *procedural* or *imperative* languages on shared-memory multiprocessors. However, in many cases, the difficulty of the problem has led to inefficient implementations.

1.1 Run-Time Support

Run-time support code includes both the operating system (OS) and the language specific *run-time system*. The run-time system is the code which supports the scheduling, synchronization, and communication needs of a parallel program which can be met without invoking the operating system kernel. Because the OS kernel is not invoked, the run-time system code executes in *user mode*. When the OS kernel is invoked, a privileged state is entered, and execution is said to occur in *kernel mode*. Because a large amount of state information must be saved whenever a transition to kernel mode occurs, it is more efficient to manage the run-time support needs of a parallel program in user mode.

The fact that low-level and ad hoc techniques for implementing parallelism have surfaced at the language level is a symptom of the inadequacy of current run-time support software. Although gains are being made in OS design for shared-memory multiprocessors [26, 2, 61, 23, 116, 127, 36], there is still no widely accepted standard for OS level support of parallel programs. Furthermore, even with the support for parallelism provided directly by the operating system, there is still a need for language specific run-time systems to support the execution of parallel programs. Because OS calls are relatively expensive as stated above, it is inefficient to invoke the operating system at every opportunity to exploit parallelism in a program. Instead, a user-level language run-time system should be called to manage the spawning and merging of program parallelism. Because this system may be invoked with very low overhead, it allows parallelism of both fine and coarse grains to be profitably exploited.

Run-time efficiency is a key factor in the design of a parallel processing system because it is the amount of run-time overhead that limits the degree to which the parallelism inherent in a program can be realized. In a simplified model where there is a constant sequential overhead involved in initiating a parallel execution thread and a program may be broken up into any number of parallel threads, it has been shown that there is a limit on the number of parallel threads that should be used in order to achieve maximum speedup[1] [106]. This model implies that the *granularity*, the number of executed instructions that comprise a parallel thread, decreases as the program is divided up into more and more threads. If the program is divided to the point where the time necessary to execute one parallel thread is

1 Speedup is the ratio of a task's fastest sequential execution time to its parallel execution time.

equal to the overhead required to initiate it, then the program's running time is the same as it is in the uniprocessor case, and any further reduction in the granularity will result in a speedup of less than 1. Clearly, lower overhead results in more potential parallelism and thus higher performance.

An efficient implementation of a parallel language should also allow parallel programs to execute on a single processor without a significant penalty from excess run-time overhead. In addition to enhancing portability and allowing development on single processor systems, this ability also provides reasonable efficiency when the program is run on one processor in a multiprocessor system. While this may not seem significant at first, it is important to note that most shared memory multiprocessors used in a commercial setting today are multiprogrammed as time-shared systems that allocate one processor per user program. This mode of operation has been termed as *multistream* execution [92]. In a situation where there are more parallel jobs than there are processors in the system, it may be desirable to restrict all jobs to a single processor. In this case, the speed of the program when run on a single processor should not be compromised by parallelism.

In addition to handling the case of a fixed number of processors with which to run a parallel program, the run-time support must also be able to cope with, and efficiently implement, the case where multiple parallel programs may be concurrently sharing a single multiprocessor system. This is essentially a multiprogramming model where each program is parallel and may be able to utilize multiple separate processors. This model has already been introduced on a commercial system [36], and we expect it to become more prevalent in the future [108, 65, 26]. Consequently, the run-time support must be flexible enough to provide a dynamic scheduling service to map program parallelism onto a number of processors that may fluctuate during the lifetime of the program. Of course, this run-time support must also be efficient to be of any practical value.

1.2 Research Goals

The purpose of the research described in this dissertation is to design, develop, implement, and evaluate run-time support structures appropriate for an efficient language specific run-time system. This run-time support code is designed in a manner consistent with the multiprogramming model described above and is intended for execution on shared-memory multiprocessor computers. Given our objectives and intentions, we limit this work to consider language models of parallelism consistent with procedural languages, e.g. FORTRAN and C, which enjoy widespread use in commercially available systems.

1.2.1 Design

The design of run-time support structures involves careful consideration of the model of parallelism provided at the language level. Much has been published regarding language models of parallelism that are consistent with the overall programming model considered in this dissertation [133, 134, 84, 7, 109, 73, 87, 106, 124, 97, 120, 77, 66, 93, 74, 27]. The majority of the work in this area is concerned with compiler detected parallelism in sequential languages or the explicit denotation of parallelism using loops, blocks, procedures, or processes as described in Chap. 2 below. For the most part, this work does not provide implementation detail, nor does it provide detailed performance analysis based on experimental testing. Our goal in this area is to build on this previous work in defining a set of constructs that provide useful models of program parallelism. Additionally, we are motivated by performance considerations to define our constructs so that their run-time support may be efficient. We demonstrate this efficiency in the other areas of our work discussed below.

1.2.2 Development

In this phase of our work, we aim to develop efficient run-time support structures for the language models of parallelism supported in our study. As a consequence of this goal, we must also define a reasonable division of labor between operating system and run-time system, and provide an interface between the two. Other work has been done in this area, but much of it has concentrated on extensive OS support or system libraries [23, 26, 116, 131] (or see [82] for a survey). Some research has focused on specifically tailored run-time support [107, 110, 124, 95, 54, 86, 32, 97, 36, 99, 44], but in most cases the detailed design of the implementation is either not provided or is proprietary. The other work that does provide implementation detail is primarily concerned with programming models not addressed in this dissertation. A more detailed survey of previous research is given in Chap. 3 below.

1.2.3 Implementation

With the designs for programming models and their associated run-time support code developed in the first two phases of our work, the next step is to implement them on a target multiprocessor system. This implementation is coded in assembly language to provide for maximum efficiency. Also, in order to: 1) compensate for the lack of privileged-level access to existing machines with the latest multiprocessor operating systems; 2) perform experimentation in a controlled environment with highly-accurate measurement capability; and 3) study the impact of architectural modification on the software, we target our code to the Astronautics ZS series multiprocessor, for which we possess an interpreter-driven register-transfer-level simulator. With this approach and a large amount of computing time at our disposal, we are able to investigate both parallel program fragments, or *kernels*, as well as complete programs. Also, this system allows us to support code with our optimized run-time system, which is not impacted by incompatible operating system code or convention.

Other work in this area using similar programming models and run-time designs has either been simulated at a much higher and potentially less accurate level [110, 37], or has been implemented by vendors on specific machines where only complete programs are studied and detailed performance data is not provided [32, 70, 36].

1.2.4 Evaluation

Given our design and implementation on a target system, this phase of the work is intended to provide an evaluation of the performance of our system. This evaluation is based on experimental testing using the parallel code and simulator described above. As another variable in our study, we modify the hardware architecture to gauge the impact of different memory system configurations which provide a cache for each processor in the system. The modified systems provide *cache consistency*[2] in hardware. We use this testbed to provide detailed timing and cache performance statistics in order to factor out the various sources of overhead due to parallelization. This overhead is broken down into three categories: 1) Static Load Imbalance, 2) Run-Time Scheduling and Synchronization, and 3) Cache Consistency Overhead. Our tests measure the relative impact of each of these overhead sources for a variety of hardware and run-time system configurations. With our run-time system and suitable language semantics, our tests demonstrate that cache consistency overhead is a performance bottleneck for fine-grained parallel programs executing on a shared-memory multiprocessor system.

Other research has been done to evaluate the performance of shared-memory multiprocessor systems. For the most part, this work has been intended to aid computer architects in improving multiprocessor design. As a result, the primary areas for research have been the design and performance of consistent caches, e.g., [62, 63, 118, 101, 132, 4, 35, 40, 98, 72] and hardware techniques for synchronization, e.g., [71, 10, 12]. The results of these efforts

[2] The cache consistency problem is introduced in the following chapter.

are typically reported in terms of program data sharing levels, the number of operations needed to support cache consistency, cache miss rates, or time needed to synchronize processors. What is typically not analyzed is the speedup of the parallel program, and how its performance is correlated to the factors mentioned above. Many studies have relied on the same set of parallel codes, which have typically been developed using the current state-of-the-art in parallel programming environments, that, as mentioned previously, has several shortcomings. This approach results in a programming model vastly different from the one used in this dissertation. While other studies report results based on experimentation with coarse-grain parallel programs, we report our experience with finer grains of parallelism.

Another major difference in the approach used in these other studies has been their reliance on address trace-based simulation. By tracing the execution of these programs and then later feeding it to a multiprocessor simulator, a significant amount of information about the program is lost. While the simulator may be able to model the actions of the memory subsystem, the notion of time is compromised, and it is difficult to accurately model synchronization or shared resource contention [25]. Our approach is centered on execution-driven simulation, where a register-transfer-level model of the entire multiprocessor system is implemented and instructions are interpreted as they are issued in order to accurately reflect run-time data and timing dependencies.

1.3 Summary and Outline

The "top-to-bottom" approach of our work is aimed at incorporating the desirable aspects of other related research in the areas of design, development, implementation, and evaluation into a single cohesive system. The main objective is to discover the most appropriate form and function of run-time support structures for fine-grained general-purpose parallel programs. This objective is met through the design and implementation of efficient run-time support code for useful models of program parallelism, and the evaluation of this system with detailed interpreter-based simulation for a multiprocessor system incorporating an existing processor. A comprehensive literature review and the details of this research are provided in the following chapters.

The work presented in this dissertation represents the first quantitative study of run-time support structures for shared-memory parallel programs. This study identifies three major performance barriers which impede the speedup of parallel programs, and quantifies their impact on the performance of a multiprocessor system. Our research also quantifies tradeoffs between parallel language semantics and implementation requirements, and uses results from the study of performance barriers to direct a new design for a parallel language model. By performing this research in the context of a complete system for parallel programming, we ensure that our results are relevant to a multiuser parallel processing system.

The remainder of this dissertation is organized as follows. Chapter 2 provides some background information and defines the terminology used. The following chapter reviews other research and states the scope of our study. Chapter 4 describes the evaluation methodology and the testbed used in our work. Chapters 5 and 6 describe the language model, run-time support design and implementation, and the experimental results for loop-based and procedure-based parallel language models respectively. Chapter 7 then describes the compiler and operating system issues concerned with merging these programming models into a cohesive parallel programming system. Finally, Chap. 8 provides a summary and the overall conclusions that we have drawn from this work.

CHAPTER 2

BACKGROUND

2.1 Introduction

In this chapter we present background information and definitions of relevant concepts pertaining to the implementation of an efficient parallel programming execution environment. The major areas covered are parallel language models, their run-time support, operating systems, and hardware architecture.

2.2 Parallel Language Models

The units of parallelism available in a program vary greatly depending on the language being considered. As stated above, we are considering parallel processing of programs written in procedural languages such as FORTRAN, C, and Ada. The wide range of language possibilities provides a corresponding range of parallel program units of execution. We can classify these units as: 1) loop iterates, 2) blocks, 3) procedures, 4) Ada-like tasks, and 5) OS-level processes. Each of these levels has been used as the basic unit of parallelism for several implementations. Loop level parallelism occurs when multiple iterations of a `do` or `for` loop may be executed simultaneously. This parallelism can be detected by a compiler, e.g., Kuck & Associates' KAP, Pacific Sierra Research's VAST, the University of Illinois' Parafrase-2, or PFC from Rice University, or expressed using parallel loop constructs such as `doall` or `doacross` [73, 87]. Parallel blocks are typically expressed using "parallel sections" where all statements in the section may execute in parallel [124]. Procedure parallelism occurs when multiple procedure instantiations can be executed in parallel. This parallelism can also be detected by a compiler, but this is usually too difficult, and system calls, compiler directives, or language extensions are used instead [82]. Program parallelism can also be explicitly stated using language level tasks or processes. Language examples include the *tasks* of Ada [21] and the *processes* of Concurrent C [66]. Operating systems such as Unix also provide processes that are managed by the OS kernel. Both language-level and OS-level processes are usually provided with mechanisms for process communication and synchronization.

2.2.1 Loops

Parallel loops are essentially `do` loops where the multiple iterations may be executed in parallel. The two basic categories of parallel loops are the `doall` loop and the `doacross` loop [106]. The parallelism is typically detected through data dependency analysis by compilers [84, 7, 109]. However, several languages and language extensions have been proposed that contain `doall` and `doacross` style loops, which explicitly state the parts of the loop that may be executed in parallel [73, 87, 95]. In a `doall` loop, there are no data dependencies between the iterations of the loop, so they can be executed in any order and in parallel. The `doacross` loop, on the other hand, has one or more dependencies between iterations which impose an order of execution. The iterations may still be overlapped, however, provided that appropriate synchronization and shared memory updating is provided to ensure data integrity. Because `doall` loops are free from the restrictions placed on the parallel execution of `doacross` loops, they provide the greatest opportunity for speedup in the execution of an otherwise serial program. For this reason, our experiments have focused on the execution of `doall` loops.

```
    doall 100 i = 1, n (shared (n,A,B,C), private(i))
       A(i) = B(i) * C(i)
100 continue
```

Figure 2.1. Example `doall` loop.

```
    doacross 100 i = 2, n (shared (n,A,B), private(i))
       A(i) = A(i-1) * B(i)
100 continue
```

Figure 2.2. Example `doacross` loop.

Figures 2.1 and 2.2 show simple examples of `doall` and `doacross` loops respectively. These examples use a syntax similar to that generated by the VAST preprocessor, which explicitly states which variables are to be shared, and which are to be replicated and kept private within each iteration. An alternative form of the `doacross` example shown in Fig. 2.2 would be to use a `doall` and denote the amount of overlap between iterations explicitly using synchronization statements expressed as a function of `i`. In the above example, we use the keyword `doacross` coupled with the loop index `i` to imply an order of execution of the iterations.

The advantages of loop parallelism include its familiarity and simplicity. By using loops, parallelism may be expressed in a sequential language. This allows a large base of programs and programmers to benefit from parallelism without switching to a new language. The simplicity of loops makes them easy to support efficiently. However, there are two main disadvantages to using loops for parallelism. First, there are many algorithms containing parallelism that cannot be effectively expressed using loops. Second, the parallelism of loops tends to be fine-grained, so that the overhead of parallelization becomes a major factor. If the number of instructions in a loop is not significantly greater than the number required to set up its parallel execution, simultaneous execution of the iterations is not profitable.

One way to avoid much of the overhead of parallelization is to use static data partitioning and loop scheduling, e.g., [75]. Static scheduling reduces overhead by simplifying the run-time support code and enabling the programmer to specify the highest levels of data locality and static load balance. Disadvantages include an inability to cope with load imbalance caused by dynamic effects. While the static approach may provide the best performance by providing the programmer with ultimate control over parallel execution, it also removes the flexibility present with a dynamic scheduling approach. As stated above, we are focusing on dynamically scheduled systems. Dynamic scheduling approaches for parallel loops are discussed in Sec. 2.3 below.

2.2.2 Blocks

Parallel blocks are supported by a parallel section construct, whose semantics allow the execution of each of its statements to proceed simultaneously. Figure 2.3 shows an example parallel section. In order to support parallel blocks of larger granularity, the statements within the parallel section may themselves consist of several statements [97, 87, 124]. In Fig. 2.3, this could be implemented by replacing each statement with a block which would, for example, be surrounded by the keywords BEGIN and END. The parallel blocks mechanism allows the programmer to specify *heterogeneous* fine-grained parallelism, as opposed to the *homogeneous* parallelism of loops. The implementation of parallel blocks is very similar to parallel loops, but is slightly more complicated as we show in Sec. 2.3.1.3.

```
do parallel (shared (X1,Y1,n,A,B,C,j,k))
   X1 = A(1) * B(1)
   Y1 = A(1) * C(1)
   n  = j + k;
end do parallel
```

Figure 2.3. Example parallel section.

While the parallel section construct is simple and straightforward, it has the disadvantages associated with parallel loops. The work of each block in a parallel section is fine-grained, especially if each block is only a single statement, e.g., the programmer specifies several independent calculations. This raises the issue of parallelization overhead, especially because the support of this type of parallelism is more complex than parallel `doall` loops. In order to overcome a problem with grain size, it may be necessary to serialize and coalesce multiple parallel blocks into larger code sections. Of course, this approach reduces parallelism and may diminish the usefulness of the construct.

2.2.3 Procedures

As with loops, procedures provide a familiar construct that can be used as a unit of parallelization. This parallelism can be either implicit or explicit. Implicit procedural parallelism is detected at compile time through inter-procedural dependency analysis [90]. Explicit parallelism is denoted by using compiler directives [82], language extensions [68] or run-time library calls [131, 8] that designate procedures as "tasks". Another approach is to use a parallel programming tool which encapsulates a user's sequential code in procedures, and provides a language level interlacing of run-time support code which is transparent to the user [123, 29, 89, 135]. Of course, it is also possible to use parallel language primitives that allow expression of parallel procedures [97, 120, 49].

Figure 2.4 shows an example of what a parallel procedure call might look like. In this example, n instances of MUNGE are created, each being passed the parameter DATA. Depending on the type of DATA and the specifics of the language, a wide variety of parameter passing semantics are possible. This will be evident in the discussions that follow in Chaps. 3 and 6. Consequently, a declaration of a parallel procedure may be as simple as a sequential procedure, or may require special keywords to identify its instantiation index, or, if applicable, its subset of DATA.

Procedures, like do loops, have the advantages of familiarity and simplicity. Also, the granularity of procedures is in general greater than that of loops, and may enable a greater efficiency in parallel processing. However, there are also some disadvantages.

```
parallel CALL [n] MUNGE(DATA)
```

Figure 2.4. Example parallel procedure call.

Although some success has been achieved, automatic detection of procedure parallelism is difficult, and in some cases may be impossible. If the program is written in a sequential language, chances for parallel execution may be lost because of the language's lack of expressiveness. In order to exploit larger amounts of parallelism, it is usually necessary for the programmer to restructure the code so that parallel calls of a procedure are explicitly stated and provided with different data sets. However, compiler directives and run-time library calls are somewhat clumsy, error prone, and difficult to debug [19]. Parallelizing overhead also tends to be greater in this case, because procedures require some local environment separate from the main program. This situation again raises the question of what is the minimum granularity needed to profitably exploit parallelism.

2.2.4 Language Processes

Processes are units of parallel execution that can be expressed at the language level. They may be scheduled independently from each other, but are usually provided with mechanisms for synchronization and communication. Examples include the *tasks* of Ada [57] and the *processes* of Concurrent C [66]. While the concept of language-level processes may have roots in earlier languages for systems programming [74] or distributed programming [27], they are also suitable for general purpose programming on shared-memory systems. In fact, the shared-memory model presented by the Ada language suggests that it is best suited for implementation on parallel systems rather than distributed systems.

Figure 2.5 shows an example of a language-level process using the syntax of an Ada task. In this example, the task executes a loop in which a message is accepted from another task. The sending task is blocked until the `end` statement following the `accept` is

```
task RECEIVER is
   entry MSG(DATUM: in INTEGER);
end;

task body RECEIVER is
   BUFFER: array(1..N) of INTEGER;

   begin
      for I in 1..N loop
         accept MSG(DATUM: INTEGER) do
            BUFFER(I) = DATUM;
         end;
      end loop;
   end;
```

Figure 2.5. Example of language-level processes: An Ada task.

executed. This allows synchronization to be specified at the language level as a side effect of the task communication mechanism.

The main advantage of using processes for parallel programming is their expressiveness. Because of the number of flexible primitives typically provided, processes may be coded to communicate and cooperate in a multitude of forms that allow the expression of a very large number of parallel algorithms. However, because of this flexibility, the run-time support for processes is more complex, and thus requires more overhead. Because they may be preempted by the run-time system scheduler, processes must maintain state, which includes structures such as communication queues and a working register set. This extra overhead requires that the grain size of a process be quite large in order to ensure efficiency.

2.2.5 OS Processes

Another form of process that can be used to exploit parallelism is not defined at the language-level, but is instead defined by the operating system. In its basic form, each user program, or *job*, is executed by a different operating system process. However, in Unix systems, it is possible for a process to execute a *fork*, which clones its address space and context for use by a new process. The fork returns a value which the code then uses to determine whether it is the child or parent process. Thus, although two identical processes are created, each has its own context, and may proceed to execute different paths through the same code. These OS-level processes may also communicate with each other using the message passing primitives provided directly by the operating system. This feature gives OS-level processes the same functionality that most concurrent languages provide. However, the run-time support for OS-level processes is provided by the operating system kernel.

The main advantage in using OS-level processes for parallel programming is that their run-time support is already provided by the OS kernel. The main disadvantages are that the model is implementation dependent on the operating system, it is difficult to use for programming large amounts of parallelism, and the overhead of forking a process is very high. However, there are applications where implementation dependencies are necessary, such as managing a set of physical resources. The overhead of a fork can also be worthwhile is the child process will execute for a very long time, e.g., until the next system reboot.

2.3 Run-Time Support

Because there are many forms of parallelism possible, there are many forms of run-time system support mechanisms for parallel programs. We can classify run-time system designs into two broad categories: 1) *microtasking*, and 2) *multitasking*. In various forms, microtasking can be used to support loops, blocks, and procedures. Multitasking can be used to support any of the parallel constructs, but its increased overhead makes it best suited for procedures and tasks. When multiple OS-level processes are used to implement a parallel algorithm, the run-time support is provided completely by the operating system kernel, and we will refer to this run-time support as *multiprogramming*.

2.3.1 Microtasking

A *microtask* is "created" when a processor allocates some subset of parallel work via execution of the run-time system code. This work is typically some number of loop iterations, a block, or a procedure. The run-time support for a microtask is provided as both code generated by the compiler backend and as lightweight routines linked together with the parallel program. This run-time support executes in user mode, thereby avoiding costly operating system calls which require expensive context-switching overhead. In the case of microtasking then, the concept of a task is defined at the run-time system level, as opposed to multitasking systems where the task is defined at the language level.

In a typical microtasking implementation, summarized in Fig. 2.6, all processors begin by executing some initialization code to acquire an individual stack area for "scratch pad" use. All processors then execute the scheduling code, which performs synchronization and arbitrarily assigns the mainline of the program to one processor. This processor begins execution while the others remain blocked or busy waiting. When a parallel construct is encountered, the running processor executes run-time support code to "set up"' global information that the other processors will use to acquire their own subset of the parallel work. This global information includes the starting address of the code and the number of iterations to be executed, if necessary. It may also contain stack, argument, and frame pointers if any of these areas are to be shared, and it may contain a count of available processors. This global information, which is shown in Fig. 2.6, may be thought of as a record entry in a global work queue. After queue initialization, a synchronization operation is performed enabling the other processors to begin accessing the queue. Each processor acquires some amount of work (according to the scheduling algorithm, discussed below) and modifies the queue record to indicate this fact. When all work has been scheduled, one processor acquires the code following the parallel construct and executes it and/or the set up code for the next parallel section. Data dependencies, which are accounted for at compile time, determine at what point this last processor synchronizes with the others executing the previous parallel section. Any processors left over that do not schedule any of the parallel work, or any of the code that follows, will continue to busy-wait or be blocked waiting for more work to be entered into the queue.

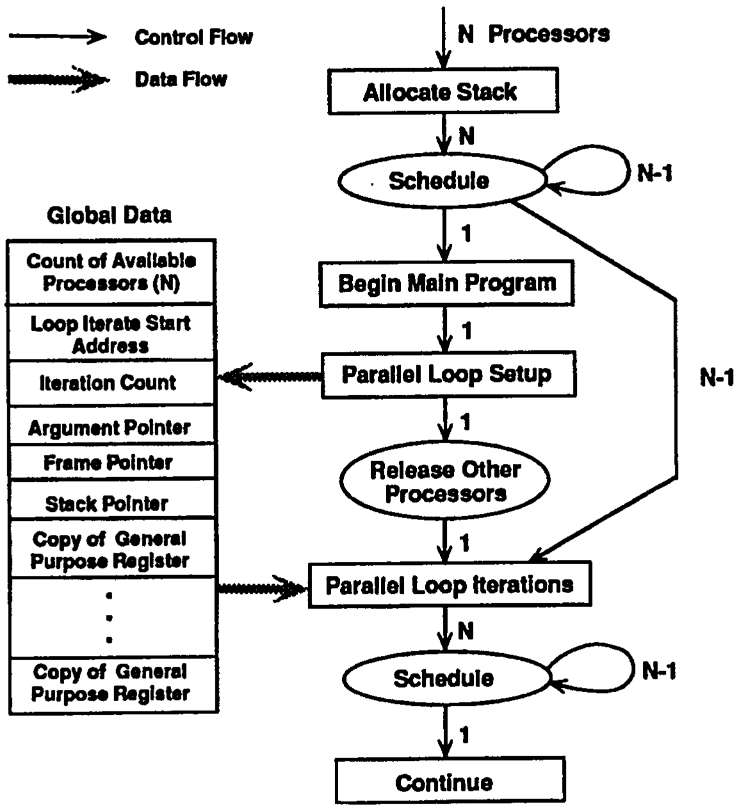

Figure 2.6. Global information and execution flow for a parallel loop.

2.3.1.1 Self-Scheduling

Allocating each processor's work by using the general scheme described above is referred to as *self-scheduling* [120, 110, 106]. This term is used because each processor enters the run-time system to allocate work without making a supervisor call into the operating system. Depending on the exact semantics of the parallel work to be scheduled, variations of self-scheduling which employ alternate techniques for partitioning work into microtasks are possible. In the case of parallel loops, these alternate scheduling techniques include *chunk scheduling* and *guided self-scheduling*. With chunk scheduling, the run-time code that sets up the execution of the parallel loop calculates an "optimal chunk size" by dividing the total number of iterations by the number of available processors. When processors perform a schedule operation, they allocate a number of iterations equal to the chunk size. If the number of iterations does not divide the number of processors evenly, one additional scheduling point is required to assign the "leftover" iterations[1].

If the scheduling times of different processors vary greatly, or if the different iterations of a `doall` loop execute a variable number of instructions, the simple self-scheduling approach tends to provide the best load balancing by evenly distributing many small microtasks. On the other hand, if the scheduling operation tends to have a high overhead relative to the number of instructions in each loop iteration, and/or the number of iterations is much greater than the number of processors, chunk scheduling reduces scheduling overhead by requiring fewer scheduling points. Guided self-scheduling is a technique designed to provide better overall performance by compromising between the extremes of best load balancing and fewest number of scheduling points. With guided self-scheduling, each processor computes its chunk size using the formula $\lceil \frac{R_i}{p} \rceil$ where R_i is the number of iterations remaining to be scheduled at step i and p is the number of processors [106, 110].

2.3.1.2 Scheduling Semantics

In the simplest case, microtasks are scheduled using *run-to-completion* semantics, which enable them to be scheduled only once before they complete. In the case where microtasks must synchronize due to data dependencies, e.g., `doacross` loops, they simply busy-wait until the synchronization allows them to continue. In order to avoid deadlock, these synchronization requests should be generated by the compiler instead of coded directly by the programmer.

[1] An alternative would be to allow slight variation in the chunk size so as to avoid the extra scheduling point. Either approach may be the most efficient, depending on run-time overhead and the effects of load balancing.

As a consequence of run-to-completion semantics, simple microtasks are, in general, unable to create additional parallel work themselves. Such a capability would require the creating microtask to become blocked while the newly created work was being executed. The only way to avoid blocking the "parent" microtask would be to alter the semantics of the parallel construct so that newly created work would not be scheduled until completion of all previously created work, including the parent microtask. This approach would result in a situation were work is created but is not executable until some future synchronization point is reached. While this change in semantics may be acceptable for some algorithms, it is inconsistent with implicit parallelism which is extracted from sequential programs. It also complicates the run-time system design by forcing it to filter out the work that can be scheduled from the work that has been created but is not yet ready to be scheduled.

As a result, most existing microtasking systems do not allow parallel constructs to be nested. This is not a serious restriction, however, since compile-time restructuring can coalesce loops and blocks to enable the creation of many more microtasks of a larger grain size [134, 106]. A benefit of this approach is that, in the case of parallel loops, the run-queue is required to hold only one record of parallel work at a time, providing for a very simple and efficient implementation. This allows the queue information to be stored at "well-known" locations, and avoids costly memory allocation and pointer chasing by the run-time system. Furthermore, these well-known locations need not reside in main memory and may be quickly accessed in the shared registers present in several multiprocessor supercomputers.

In a more complex form of microtasking, *run-until-block* semantics can be supported. By using a variable length run-queue and allowing the nesting of parallel constructs, microtasks may block either on synchronization variables or to create more parallel work. In either case, lightweight context switching will have to be supported as well as techniques for managing separate data areas for microtasks. This style of microtasking is more flexible, e.g., it allows recursion, but it requires a more complex implementation which may result in additional execution overhead. We refer to this type of run-time support as *extended* microtasking.

2.3.1.3 Heterogeneous Parallelism

The above discussion of microtasking is primarily aimed at the homogeneous parallelism found in loops or multiple calls to the same procedure. As mentioned above, the support for the heterogeneous parallelism of blocks is more complex. If each block is treated as a parallel loop of one iteration, the simple length one run-queue cannot be used, or the benefits of parallel blocks will be lost altogether. An alternate strategy is needed, where each "iteration" value assigned via self-scheduling is used to compute a unique starting address. A table look up technique can be inserted into the generated code at the beginning of the parallel section.

If extended microtasking is being supported, the alternate to the table look up technique of supporting parallel blocks can be used, where each block is treated as a single iteration loop. Also, because extended microtasking allows nesting of parallel constructs, there are more opportunities for the exploitation of heterogeneous parallelism by the programmer. This capability can be exercised in the form of nested parallel loops and/or nested parallel procedure creations in addition to simple parallel blocks.

2.3.2 Multitasking

Multitasking run-time systems support the execution of language tasks or processes as well as procedures that can be thought of as restricted forms of tasks. In addition to support for run-until-block semantics, multitasking run-time systems must also support more complex forms of context switching brought about by *preemption* semantics, where one task may block another running task in order to begin or resume execution. Furthermore, mechanisms must be provided to support intertask communication. Because of the need for preemptive scheduling, task control blocks (TCB's) must be kept for each task to hold its state and data [43]. These TCB's are then linked in the run-queue or communication queues. These requirements make the run-time system more complex, and overhead is higher than in the microtasking cases. However, the semantics of language-level tasks or processes allow the expression of many different parallel algorithms including those that require heterogeneity, recursion, or complex synchronization models associated with simulation or real-time systems.

Parallel languages which provide a process or task model that requires multitasking support are referred to as *concurrent* languages. When compared to microtasking systems, concurrent languages require additional run-time services. In addition to the scheduling and communication requirements listed above, some languages, e.g., Ada and Concurrent C, also require support for dynamic memory allocation, time-of-day clock, time controlled events, implementation of complex create/terminate semantics, handling and propagation of exceptional conditions up multiple call stacks, and handling of interrupt signals. While some of these services may be supplied to a microtasking system as library service routines, e.g., timing support and dynamic memory allocation, concurrent languages usually have explicit semantics regarding these services incorporated into the language definition. An analysis of the run-time system requirements for Ada can be found in [43], and the scaled down requirements for a simpler concurrent language intended for distributed-memory systems and based on C can be found in [45].

2.3.3 Multiprogramming

As mentioned above, multiprogramming refers to run-time support provided by the operating system kernel for multiple OS-level processes. This name is derived from it main application, where each OS-level process represents a different user program. However, multiple OS-level processes may be created by a single user program using the fork system call. In either case, the scheduling, synchronization and communication support for processes is provided directly by the operating system kernel.

Even though complete run-time support can be provided by the OS kernel, the semantics of an OS-level process and the overhead in creating and scheduling it has made it of limited use for implementing parallel algorithms. Other problems with the OS-level process include its inherent implementation dependencies, which make portability difficult. Despite these shortcomings, OS-level processes are still a necessary component in the support of parallel programs. They provide the interface to the resources of the hardware and operating system, e.g., disk drives and virtual memory. The role of the run-time system then is to provide as much user-level support as possible for scheduling, communication, and synchronization to the parallel program. This limits invocation of the OS and thus reduces system overhead. This layering approach to run-time support software is discussed further in Sec. 2.4 below.

2.3.4 Interprocess Communication

In each of the language models and run-time support approaches discussed above, there is some form of communication that occurs between different threads of execution, e.g., microtasks, tasks, processes, etc. This communication is referred to *interprocess communication* or simply *IPC*. For parallel loops or blocks, IPC is handled by the run-time system through shared global data used for providing microtask context as shown in Fig. 2.6. If procedures are supported as language units of parallelism, the exchange of parameters between the caller and callee of the procedure is another form of IPC. Concurrent languages typically provide language-level support for explicit message passing or rendezvous, which defines IPC between language-level processes. Finally, operating systems such as Unix provide IPC mechanisms, e.g., sockets and streams, which enable data to be exchanged between processes.

2.3.5 Summary

Table 2.1 classifies the main characteristics of the language models of parallelism and the associated run-time support mechanisms described above. The association between

Language Model	Run-Time Support	Scheduling Semantics	Interprocess Communication	Grain Size (instructions)
Loops, Blocks	Microtasking	Run-until-completion	Global data	<10,000
Procedures	Extended microtasking	Run-until-block	Parameters	≈10,000
Language Processes	Multitasking	Preemptive	Messages, Rendezvous	>10,000
OS-level Processes	Multi-programming	Preemptive	Messages	>>10,000

Table 2.1. Typical language model/run-time support combinations.

the language models and their run-time support is provided for typical implementations and those described in this dissertation. That is, variations are possible that allow extended microtasking support for parallel loops or even multitasking support for OS-level processes. Also, the run-time support for microtasking, extended microtasking, and multitasking still requires a layer of support provided by the OS kernel. This is discussed in more detail in Sec. 2.4 below.

In Table 2.1, the language models and run-time support mechanisms list near the top have the lowest overhead and are the easiest to implement. However, they are also the least flexible in terms of expressing parallel algorithms. The last two rows in the table represent programming models with greater flexibility, but this comes at the price of increased overhead and implementation complexity.

The IPC mechanisms required by the different language models similarly vary in flexibility, expressiveness, and implementation complexity. In each case, the IPC mechanism required is consistent with the scheduling semantics provided and the expected grain size of the execution thread. For example, the simple scheduling semantics and small grain size of a microtasking run-time system for parallel loops is consistent with the shared global data form of IPC provided. It is inappropriate to include a complex IPC mechanism that requires blocking for a communication request if there is no mechanism provided to block the execution thread. Also, requirements for fair servicing of IPC requests for multitasking and multiprogramming systems is consistent with preemptive scheduling, which interrupts execution threads and reallocates processors to waiting threads at regular intervals.

2.4 Operating System

The main role of the operating system is to allocate resources. In a parallel processing environment, programs must be able to acquire multiple processors and these processors must be able to share virtual memory. A typical approach to this problem is to allocate a fixed number of processors to the program [81]. In order to simplify the management of this allocation, context switches into or out of the program involve all processors simultaneously. The allocation of processors and support for shared virtual memory is implemented in the operating system using OS-level processes that are scheduled by the kernel. This processor allocation approach allows the self-scheduling to occur without OS intervention, since the processors access a shared run queue in user space to determine their share of the work.

One approach that might simplify run-time system design is to map program fragments or tasks onto OS processes one-to-one. However, there are two main problems with this approach: 1) the overhead associated with creating and scheduling OS processes is very great, and 2) the semantics of the language unit of parallelism may be inconsistent with the OS process. In the first case, the amount of state information these processes carry cause context switching to be expensive. This state may include open file descriptors and the virtual address translation mapping in addition to a status word, stack pointer, program counter and register set. This replication of state in each process makes it difficult to share resources between processes in a Unix-like OS. Because of the state they carry, these OS processes are referred to as *heavyweight*. In some new systems though, direct support for *lightweight* processes is available from the OS kernel, e.g., Mach [2]. However, even though these lightweight processes carry less state and take advantage of shared resources in a global location, the overhead of entering the kernel on their behalf is still too expensive for most forms of language parallelism, including language-level tasks [96]. Also, see the performance figures in [26].

In addition to performance problems, it is possible that language units of parallelism may not map semantically onto OS processes. Complex process models that accommodate interprocess communication and preemption semantics are not appropriate for simple microtasks. Further, in the case of concurrent languages, it is possible that the language defined semantics for creation, termination, and communication cannot be supported directly by the existing semantics of OS processes.

The performance problem coupled with the possibility of semantic problems effectively prevents a direct mapping of parallel language units to operating system processes. Therefore, the more common approach to supporting program parallelism is to use a language-specific user-mode[2] run-time system. Figure 2.7 illustrates the layering of software and hardware using this approach. This diagram shows which parts of the software have access to certain instructions provided by the hardware. Only the operating system kernel has access to privileged instructions such as those provided to manipulate the translation lookaside buffer. Instructions provided by the hardware for synchronization and/or communication are visible to both the run-time and operating systems. This allows the run-time system to perform scheduling and synchronization as needed. The application program may only access the remaining type of instructions, which are referred to here as "regular instructions". Many current systems allow application-level code to request synchronization through operating or run-time system calls. In an idealized system, however, synchronization and communication are implicit in the semantics of the language, and they are handled directly by the run-time system without programmer intervention.

2.4.1 Threads

As mentioned above, operating systems are appearing that support lightweight processes

[2] Recall from Chap. 1 that *user mode* refers to code that does not execute in the OS kernel.

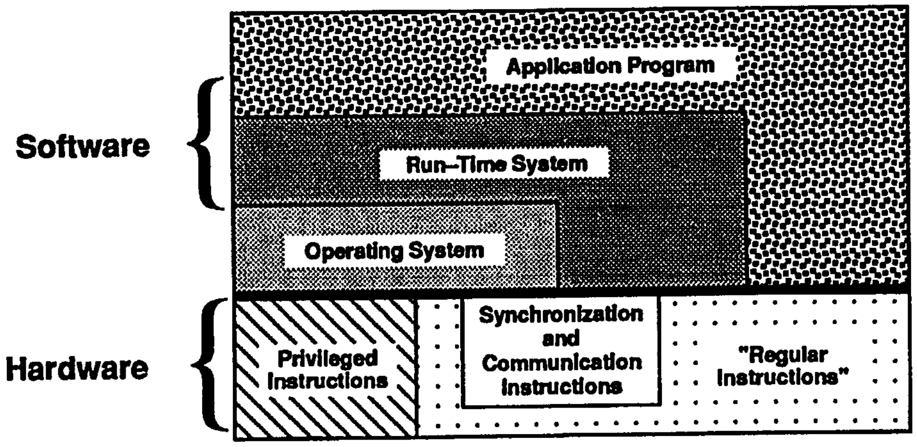

Figure 2.7. Software/hardware hierarchy.

and virtual memory sharing. In Mach, for example, a *task* acts as a heavyweight process and contains the open file descriptors and virtual memory mapping for a collection of lightweight processes referred to as *threads*. The threads are the units of execution that are scheduled by the OS kernel and share the resources of the parent task. In the case of Mach, a single task can be allocated to a parallel program, and the multiple execution streams of the program can be mapped dynamically onto several threads by the run-time system. Other options are possible in a Unix-like operating system, including the allocation of multiple heavyweight processes to a single program, e.g., Symunix [61]. In any case, the operating system processes (heavyweight or lightweight) that represent execution units can be regarded as *virtual processors* [26]. The language run-time system is then responsible for scheduling the units of program parallelism amongst the virtual processors. This allows the dynamic scheduling of parallel execution streams to occur in user mode with less overhead than the alternative of assigning each execution stream to an independent OS process.

2.4.2 Scheduling

With the model for scheduling program parallelism mentioned above, there are additional problems to consider regarding the OS scheduling of threads, i.e., virtual processors. While it has been suggested that parallel language units be mapped one-to-one onto threads [26], we believe that the frequent creation and termination of relatively small grain language units does not warrant this approach. The run-time system can schedule these units onto available threads much more quickly, as will be demonstrated in Chaps. 5 and 6.

Another issue regarding thread scheduling is the usefulness of *coscheduling* [23] or *gang scheduling* [65, 26]. With this type of scheduling, context switches into or out of the parallel program involve all threads allocated to that program. This approach is intended to avoid long delays and/or thrashing of the scheduler when all threads of a program are blocked at a barrier synchronization point while the last thread has been preempted or is blocked for some other reason. However, this strategy may cause unnecessary preemption of some threads and prevent the scheduling of any part of the program when the number of physical processors available is less than the number of threads allocated to the program. If threads are preempted or deallocated only during run-time system scheduling points, gang scheduling may be unnecessary, since the mapping of program units onto threads is handled by the run-time system scheduling mechanism. Furthermore, in the case of heterogeneous parallelism, the multiple threads of a single program may not need to synchronize at all, and gang scheduling provides no additional benefit. Also, depending on the structure of the code, it may or may not be desirable to use gang scheduling when a system call by one thread may cause preemption. The case of I/O system calls is discussed below. These issues regarding scheduling and preemption are discussed further in the context of a proposed parallel programming system in Chap. 7 below.

2.4.3 Dynamic Memory Allocation

A common service provided by operating systems is the allocation of additional memory space to a program at run-time. This function is implemented in Unix via a variety of routines including `malloc` and `sbrk`. In the case of a multiprocessor operating system, there are some new issues to consider. In addition to supplying additional memory space, it will most likely be necessary to make this space addressable by all threads or OS processes dedicated to the parallel program. This will certainly be necessary if the run-time system itself requested the storage. Also, multiple requests for storage from the same parallel program should be serviced in mutual exclusion so that multiple calls do not receive overlapping memory spaces. This type of dynamic memory allocation is supported in Dynix via the `shsbrk`, `shmalloc`, and `mmap` system calls [23].

Depending on its complexity, the run-time system itself may choose to manage dynamic memory allocation. If a large chunk is allocated from the OS at start-up, the run-time system can allocate memory from this pool for its own use, and manage deallocation as well. This avoids making many costly calls to the OS. However, the OS must still provide the capability described above, for those programs which allow user-level requests for storage via a system call and when the run-time system requires additional storage at run-time.

2.4.4 I/O Support

Support for I/O operations is another area typically managed by the operating system. While this support via a system call is straightforward for a uniprocessor program, there are several issues to consider in the case of a parallel program. If multiple threads are not synchronized and perform I/O operations using the same file descriptor information, problems may occur [23]. While these problems can be solved at the language level by synchronizing tasks or microtasks before performing I/O operations, it may be desirable in general to allow parallel I/O activity. Direct OS support for parallel I/O with shared file descriptors may be necessary [23]. This support may even require some form of built-in synchronization.

The other major problem related to I/O activity in parallel programs is the interface between thread/task states and OS-level scheduling. If a language-level task (or microtask) requests I/O activity, the associated OS-level thread making the call may become blocked until that activity is completed. While this may introduce some delay in the time it takes the program as a whole to reach its next synchronization point, the approach simplifies the duties of the run-time system. The operating system reinstates the thread and task (or microtask) at the point of suspension after completion of the I/O activity without any further action required by the run-time system. However, other possibilities exist which prevent the blocking of the OS-level thread, thus avoiding a temporary reduction in processor allocation for the parallel program. In Mach, for example, an asynchronous system call is possible where a thread may continue and check the status of the call at a later time. To incorporate this approach, the run-time system must be aware of and support the blocking of language-level tasks or microtasks for I/O activity. This will complicate the run-time support if run-to-completion semantics are expected.

Each of the above stated OS issues and their impact on scheduling and synchronization are discussed in the context of a proposed parallel programming system in Chap. 7.

2.5 Hardware Architecture

In addition to the view the operating system presents to the generated code of a parallel program, it is also critical to consider the hardware architecture. As stated above, we are primarily concerned with multiprocessors where each processing element has the ability to directly access a physically shared memory. Main memory can be centrally located, with access served by a bus, crossbar, or other interconnection network. Alternatively, the memory may be spread across multiple memory units local to each processor as long as each processor may access any other processor's local memory. In this latter configuration, the time to access memory may be variable, depending on the location of the memory unit relative to the referencing processor. Such a system organization has been termed Non-Uniform Memory Access (NUMA).

Caches may also be present that serve subsets of processors (typically one per processor) in order to, among other things, reduce the main memory bandwidth requirement. Because memory is shared, it is possible for multiple caches to contain different versions of the same data item, causing the *cache consistency* or *cache coherency* problem [119] which requires either a hardware-based or software-based solution. If caches are kept consistent via a hardware mechanism, they are transparent to the user program and operating system. Otherwise, software must maintain consistency, and these caches then become a form of *local memories*. Many methods have been proposed for both hardware and software controlled cache consistency. Surveys of approaches for hardware consistency appear in [17] and [4], while software solutions are proposed in [38, 56, 39, 41]. A brief survey of both hardware-based and software-based cache consistency schemes is given in [102].

In systems where a shared bus is used to access main memory, the cache consistency problem is commonly solved using a bus-snooping cache consistency protocol. In systems where there is no shared bus, when, for example, a crossbar or multistage interconnection network is used to connect the processor-cache subsystems to a shared memory, the cache consistency problem is usually solved in software or by using a central directory-based hardware scheme. Although other techniques such as a shared cache or snooping on a crossbar are possible, these approaches require complex hardware support and are not scalable to more than a few, i.e., more than a dozen, processors. Therefore, our experiments focus on software and directory-based hardware schemes.

2.5.1 Software Consistency

We can categorize software mechanisms for cache consistency into three general types that rely on: 1) non-cacheable data; 2) compiler-managed cache block consistency; or 3) write merging. The first type is the classical technique where shared data is made non-cacheable, typically on a page-by-page basis. The second type requires that the compiler utilize information about data dependencies and alignment of words in cache blocks to bypass the cache on a fetch if necessary, and to write through or flush blocks to main memory when needed. The third type allocates and zero fills cache blocks for result data areas, and then overlays these blocks at the conclusion of the parallel loop to form the final result in main memory. This last approach allows multiple processors to share cache blocks, but requires that no two words be shared between microtasks. This property has been referred to as *false sharing* [60], and allows the overlaying of result areas to be performed using simple OR instructions on each logical word.

Of the three approaches, the second has the most potential for fast execution as it incorporates the most information about the structure of the code and its data dependencies. However, this approach is also the most difficult to implement. The most sophisticated such scheme is called Fast Selective Invalidation and is summarized in [40]. In this scheme, the compiler tags each reference as one that either may retrieve the data from the cache, *cache read*, or must retrieve it from memory, *memory read.* This distinction is made based on how the data is accessed in the program. Data that is used in a read-only fashion is always marked cache read. If data is written at any point in the program, it is marked memory read in any parallel loop following the one in which it is written. However, it is still possible for a memory read to retrieve data from the cache. This is permitted if the block was written earlier in the same parallel loop, i.e., microtask, currently executing. This is supported by a special *change* bit in the caches for each block which is reset before and after execution of a microtask. The change bit for a block is set whenever that block is written. If the change bit is true, a later memory read may be retrieved from the cache.

Another scheme that requires compiler support, called Version Control, is also proposed in [40], but this scheme requires substantial hardware support. This scheme maintains version numbers and updates them at the beginning and end of each microtask, instead of resetting change bits. The version number is incremented at the end of a microtask for a cache block if it contains any part of an array that was written by any microtask associated with the current parallel loop. This version number, or CVN, is maintained by each processor in a separate private memory. When a block is cached, an additional cache block tag field called the birth version number, or BVN, is loaded with the current CVN, or CVN+1 if the access is a write. Later references to this block compare the CVN to the

BVN, and if the BVN is less than the CVN, the cache block is stale, and a memory fetch must take place.

In addition to requiring sophisticated compile-time data dependency analysis, the schemes described above also makes three key assumptions: 1) caches are write-through to main memory[3], 2) task migration is disallowed, and 3) the cache block size is one word. While the authors state that the scheme can be modified to support a change in these assumptions, it appears to be a non-trivial task to do so. For example, the scheme also requires that no two processors access the same cache block for a given parallel loop. This is a reasonable requirement if the cache block size is one word. However, with a multiword cache block, which is the norm in existing computer systems, it is quite likely for different processors to access different cache words in the same cache block (false sharing as introduced above). In order for this scheme to work, either microtasks have to be constructed that do not share cache blocks, or data must be aligned by the compiler so that only part of a multiword block is actually used. Of course, a single word cache block could be implemented in hardware, but this would seriously degrade performance by losing the advantage of spatial locality in other data references. This is a difficult problem to solve.

In our experimental system (described in Chap. 4), the cache block size is 16 words, and caches are copy-back to conserve interconnection bandwidth. It has been shown elsewhere that write-through caches do not in general perform well when compared to similarly configured systems with copy-back caches [128]. Also, the restriction on task migration is inconsistent with our model of parallelism and its associated run-time and operating system support. Finally, the scheme described in [40] also requires hardware support.

2.5.2 Directory-Based Hardware Consistency

The classic directory-based hardware cache consistency scheme is the one proposed by

[3] That is, data written to a cache is written to memory at the same time. A *copy-back* cache updates memory later when the updated cache block is to be replaced.

Censier and Feautrier in [34]. It is similar to the one originally proposed by Tang [126] but requires fewer cache tags and a smaller central directory. The directory is located in main memory and contains an entry for each cache block in the system. Each entry contains "location" bits to indicate which caches, if any, possess the block and a "modified" bit to indicate whether or not the main memory version is up-to-date. If the memory version is not up-to-date, the modified bit is set and the block may reside in only one cache. For the directory tags, each entry requires $P + 1$ bits for a P processor system.

While the Censier and Feautrier scheme allows sharing of clean lines, a modification to this scheme suggested in [4] restricts blocks to reside in at most one cache. The location bits can then be replaced with an index that indicates the cache where the block resides. Only $\log_2 P + 1$ bits are required for each directory entry. This scheme is based on the assumption that sharing levels are generally low, and it is better to conserve directory bits rather than support extensive sharing. However, as pointed out in [50] and reiterated in Chap. 5 below, this scheme is too restrictive as it also prevents the sharing of read-only data. Due to the performance problems with this modified scheme, a compromise scheme which allows blocks to reside in some fixed number of caches less than the total has also been proposed and tested [35]. This reference introduces terminology which refers to the Censier and Feautrier scheme as a *full-map* directory, while the modified schemes are termed as *limited* directories.

Another possibility for a directory-based consistency scheme is a distributed, or *chained*, directory. Chained directories introduce extra cache block tags that are used to form either singly or doubly linked lists of the same cache block in different caches. When a block is introduced into a cache, it is added to the list for that block. These lists are used to propagate invalidations from cache to cache if the line is updated by a processor. Additional techniques are required to implement this scheme in order to preserve the linked list structure in the case of block replacement. Pilot implementations of both doubly [78] and singly [129] linked list chained directories are underway.

A directory scheme that incorporates a broadcast mechanism has also been proposed in [16]. This scheme maintains only two bits per cache block directory entry, which indicate one of four states: 1) block not cached, 2) block clean in exactly one cache, 3) block clean in an unknown number of caches, and 4) block dirty in exactly one cache. Because the directory entry contains no cache pointers, this scheme broadcasts invalidations and write-back requests. While this approach is the most scalable of directory schemes, it is also potentially the most costly in terms of interconnection network bandwidth. Modifications that incorporate some fixed number of cache pointers into the directory (similar to the limited directory approach) may be useful in reducing the number of broadcasts needed.

Our experiments tested two versions of directory-based hardware cache consistency: 1) the full-map directory, and 2) the limited directory with only one cache pointer per block. Although the full-map scheme is not scalable to many dozens of processors, we believe, based on our simulation approach (described in Chap. 4) and other published results [35], that the results of these experiments give a good indication of the performance possible when using limited directories with more cache pointers or some version of a broadcast-based scheme. For both approaches tested, which we term *shared* and *private* hardware consistency respectfully, the directory information is used on cache misses or non-private hits to retrieve the up-to-date cache block and, if necessary, invalidate other copies and update the tags and directory. Further modifications to all of the above-mentioned schemes are possible, including the addition of a write-broadcast approach [47]. Such a scheme would update all caches on a write to a block instead of sending invalidations. However, the large bandwidth requirement of such an approach combined with its potential to update unused blocks makes this scheme unpopular with computer architects. Additional details of our implementation of cache consistency can be found in Chap. 4. Surveys of several central directory-based schemes can be found in [4] and [102].

2.5.3 Synchronization

Another major feature of the hardware architecture of shared-memory multiprocessors is the synchronization mechanism provided. This feature is a vital component in the design, as it enables software to be built that utilizes the multiple processors. A synchronization operation is usually based on an indivisible read-modify-write operation on either a main memory word or special memory hardware. Examples of such instructions include test-and-set [76], fetch-and-op [69], and compare-and-swap [76].

If a read-modify-write instruction is provided that operates on a word of main memory, a large number of synchronization variables that can be directly supported in hardware are available to the program, but at potentially great cost in access time. This approach also raises issues regarding cache consistency and synchronization because concurrent attempts to lock a memory word by multiple processors can result in a ping-pong effect, where ownership of a cache block changes frequently between processors and many invalidations are sent through the system [59]. Alternatively, special hardware synchronization locations provide for fast access, but they are limited in number. These hardware locations may be special shared registers [53, 18] or special memory areas [22].

In some systems with process-based programming models where large numbers of processes may be running on many processors, it has been shown that synchronization activity may become a bottleneck which seriously degrades performance [71]. To overcome this problem, basic hardware synchronization mechanisms are augmented with software techniques to implement synchronization with less hardware contention. Studies have shown a great performance improvement with the use of software enhanced synchronization techniques when contention for hardware locks is great [71, 10]. However, for the synchronization requirements of our programming model and run-time system on a 16 CPU multiprocessor, these software enhancements are unnecessary. This was largely due to the quick fetch-and-op synchronization directly supported in hardware in our testbed, which all but eliminated the need for critical sections in the run-time system. This is discussed further in Chap. 4 below.

2.5.4 Sequential Consistency and Weak Ordering

The concept of *sequential consistency* was originally introduced in [85] as a criteria to establish whether a multiprocessor could execute a parallel program correctly. The formal definition in [85] states:

> [Hardware is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

In [3], the above definition is restated as the following two conditions:

> 1) all memory accesses appear to execute atomically in some total order, and 2) all memory accesses of each processor appear to execute in an order specified by its program (program order).

When these conditions are met, it is impossible for a write from a given processor to be observed by another processor at a particular time unless it can be observed by all other processors at that time. While this is a desirable property, and one that is expected by programmers, it can be difficult to implement in high-performance systems where out-of-order execution is allowed on individual processors, or processor to memory interconnection networks allow memory accesses to be received out-of-order.

In order to overcome the implementation difficulties of sequential consistency, a weaker condition termed *weak ordering* was originally introduced in [58]. In [3], a revised formal definition of weak ordering was given as:

> Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.

The idea behind this weaker condition is to provide sequential consistency only if programs use synchronization primitives with critical sections to prevent race conditions. This enables the hardware to enforce sequential consistency only when synchronization instructions are encountered. Alternatively, the software can contribute to enforcing sequential consistency of synchronization operations by issuance of a special instruction to commit all outstanding memory accesses before the next instruction (a synchronization operation) can be issued. Such an instruction has been referred to as a *fence* [105, 3], and one is provided in the Astronautics ZS architecture called STORES [18]. In our simulation testbed, a similar instruction, called QUIET, is used to wait for all memory accesses to complete in addition to waiting for all processor pipes to empty. It is also important to note that completing memory accesses implies completion of any associated invalidations in a cache-consistent multiprocessor.

2.6 Summary

In this chapter we have provided background material and definitions that pertain to parallel programming language models, their run-time support code, the resultant role of the operating system, and the major architectural issues regarding the software's implementation on a shared-memory multiprocessor. In the next chapter we will examine several research efforts that have produced similar programming models, run-time systems, operating systems, and implementations on actual/simulated multiprocessor systems.

CHAPTER 3

RELATED WORK

Although a relatively young field, considerable work has been reported on parallel processing systems and their programming. Parallel programming models range from automatic compiler-detected parallelism in sequential languages, explicitly parallel procedural languages, to implicitly parallel functional and single-assignment languages. The run-time support design and implementation for these programming models has ranged from dynamic self-scheduling microtasking systems to multitasking systems and statically scheduled Single Program Multiple Data[1] (SPMD) systems. This run-time support has also been implemented both in user mode and through library calls to operating system routines. In addition to the wide range of programming models and their implementation, the targets for this software have ranged from small- and medium-scale shared-memory multiprocessors to large-scale distributed-memory multiprocessors as well as heterogeneous multicomputers, distributed systems of networked personal workstations, systolic arrays, and dataflow machines. In most cases, some degree of performance evaluation of the resulting software/hardware system has been done.

As stated in Chap. 1, our focus is on procedural languages which may provide both explicit constructs and parallelizing compilers to take advantage of the fine- to medium-grained parallelism found in loops and procedures. Our run-time support conforms to the dynamically self-scheduled microtasking model. Furthermore, we restrict our target to

1 In an SPMD system, a copy of a single program is executed on each processor. Each copy executes with its own set of data, which is usually a combination of statically partitioned data and that exchanged via messages with other processors at run-time.

be a multiprocessor with either shared memory or a transparent implementation of shared memory by the operating system. With this context in mind, we present a review of related research efforts in the rest of this chapter.

3.1 Parallel Language Models

In the following paragraphs, we describe related efforts in the design of procedural languages for parallel processing which incorporate parallel loops and procedures. For completeness, we also briefly describe concurrent and distributed languages as well as tool-based parallel programming systems.

3.1.1 Parallel Loop-Based Languages

Most of the work done in this area has focussed on automatic detection of parallel loops in sequential FORTRAN-like programs. This work began at The University of Illinois in the 1970s and resulted in the Parafrase project [84]. This was the context for Wolfe's thesis [133], which formalized the concept of data dependency and defined its relationship to parallelizing sequential program loops. Similar work was also performed at Rice University [7].

Both of these efforts have continued, and formed the starting point for numerous other researchers who have also contributed to the state-of-the-art in compiler-detected program parallelism. Many papers have been published describing the identification of parallel loops [134, 103, 109, 20, 91], the invention of the `doacross` parallel loop that accounts for inter-iteration synchronization [55, 125], and techniques for the analysis of interprocedural data-dependencies to enable compilers to parallelize loops containing procedure calls [30, 31, 90]. However, although these techniques have proved beneficial in compiling code for processors with vector units, the current state-of-the-art in compiler-detected program parallelism, by itself, is not a substitute for parallel programming, as the results of empirical studies have found [117].

Particular problems present in the case of automatically detected parallelism in compilers are interprocedural data-dependency analysis and the presence of pointer type variables. In the first case, when a compiler is trying to determine whether or not a loop may be executed in parallel, it must be able to detect any data dependencies that inhibit parallelization. While this task may be straightforward in many cases, it is complicated when procedure calls are present in the loop being analyzed. In this case, the compiler must determine whether or not there are any data dependencies within the called procedure that prevent parallelization. This situation is a problem when the compiler does not have access to the source code for the called procedure at the point of the call. This problem is aggravated by separate compilation, which is becoming more commonplace in newer programming environments which incorporate object-oriented design and other advances in software engineering.

The second problem is data dependency analysis in the presence of pointer type variables. Pointer types allow variables to be declared that hold address values which refer to other objects. Their presence in a program complicates data dependency analysis, since it is typically not possible to determine which object a pointer type variable will refer to at run time. Thus, it may be impossible to determine if any data dependencies exist, and parallelization is effectively prevented. This problem has largely been ignored since there is no pointer type in the FORTRAN language. However, these variables are commonplace in other procedural languages, e.g., C, and have proved to be a valuable programming tool. This problem must be addressed if compiler-detected parallelism is to be profitably exploited for a large base of sequential programs.

While advances in automatic parallelization of sequential programs may successfully address these problems in the future, other efforts aimed at exploiting the parallelism of the looping construct have involved explicit programmer encoded parallelism. Two notable proposals based on FORTRAN include Cedar FORTRAN [73] and the work of the Parallel

Computing Forum (PCF) [87]. Other proposals which include parallel loops and are based on Pascal or C include Simultaneous Pascal [124, 95], EPEX C [97], and PCP [28]. We briefly describe these proposals below.

3.1.1.1 Parallel FORTRAN

The Cedar FORTRAN language [73] was designed for use on the Cedar Multiprocessor under development at The University of Illinois. The Cedar-1 machine is a collection of four clusters of eight vector processors, each cluster being an Alliant FX/8 which shares a global memory with other clusters. The configuration may be extended to include many more clusters. The language is intended to take advantage of the hierarchical shared-memory architecture. It also has its roots in past efforts to vectorize and parallelize FORTRAN, as described in [73]. (For example, array/vector extensions are adapted from the proposed FORTRAN 8x standard [9].)

Cedar FORTRAN provides constructs for both `doall` and `doacross` loops. Both loop types allow declaration of local variables for each iteration of the loop. `doall` loops do not have an order of iteration execution specified for their iterations, but `doacross` loops must be scheduled *horizontally*, that is, in the order specified by the iteration index. For `doacross` loops, synchronization primitives are provided and must be specified by the programmer to control the amount of overlap allowable between iterations of the loop. Cedar FORTRAN also contains "macrotasking" extensions similar to those provided by Cray FORTRAN [86, 82, 19]. These library routines allow the spawning of tasks which use a specified procedure as a code body. These tasks are also assigned some number of processors to execute them and are termed "cluster tasks" in Cedar FORTRAN.

Each of the two parallel loop types may be further described with implementation directives called "confined", "spread", or "cross-cluster." Confined loops are confined to the processors of the cluster task in which they are executed. Alternatively, spread loops are executed by the processors of multiple cluster tasks. The intent of these first two implementation directives is to nest confined loops within spread loops, so that the hierarchical structure of Cedar may be mirrored in the program. However, cross-cluster loops are also defined which enable the compiler to combine the processing resources of multiple cluster tasks and perform the mapping of loop iterations onto processors, thus relieving the programmer of this effort.

Cedar FORTRAN also contains several synchronization primitives, as well as techniques for partitioning data into shared and private common blocks. Shared data includes variables local to cluster tasks and `global` data. Global data resides in the global store shared by all clusters, but is demand paged to the clusters that reference it. In fact, all shared data is demand paged to the clusters that request it. For synchronization, a large assortment of primitives are provided including the `lockon/lockoff` routines for mutual exclusion and `evpost/evwait` routines for barrier and producer/consumer style synchronization provided by Cray FORTRAN. Cedar FORTRAN also includes the `with` statement for specifying a critical section. The `with` statement specifies a resource that must be acquired before the statements following it may be executed. The resource for the `with` statement is a variable declared as `sync`. These variables include tag fields which may be used by `with` statements and updated in a way to order the synchronization of processors.

The PCF proposed parallel FORTRAN [87] is similar to Cedar FORTRAN in many ways, but also contains several differences. PCF FORTRAN does not contain `doall` or `doacross` loops *per se*, but instead provides a `PARALLEL DO` statement, with which both of the aforementioned loop types may be implemented. PCF FORTRAN also provides a `PARALLEL SECTIONS` construct as well as a `PARALLEL CALL` facility for parallel procedure calls (discussed below in Sec. 3.1.2). `PARALLEL SECTIONS` allows specification of groups of statements that may execute in parallel. PCF FORTRAN also provides a wide variety of synchronization primitives as well as techniques for declaring local and shared data of different classifications.

As with the Cedar FORTRAN cluster task, PCF FORTRAN provides a notion of a larger unit of encapsulation which handles the execution of the parallel constructs. In PCF FORTRAN, this unit is called the *virtual processor*. The loop iterations or other units of work created by the parallel constructs (which we call microtasks) are referred to as *chores*. In order for the programmer to specify parallelism, he or she must create several virtual processors and then create chores within them. In order to guarantee that multiple virtual processors contribute in the execution of a single parallel construct, the construct must be declared as `SPREAD`, and the programmer must ensure that each `SPREAD` construct is encountered in the same order in each contributing virtual processor. Virtual processors are scheduled by the implementation onto physical processors, but more than one processor may not simultaneously execute the same virtual processor. Thus, multiple chores created by a single virtual processor without the `SPREAD` form of the parallel construct are restricted to a single processor. These chores may then only execute in parallel if the processor provides additional parallelism through multiple instruction streams, e.g., HEP and Horizon, or vector processors. In essence, the programmer is required to manage the amount of parallelism seen at the run-time support level as well as the logical program-level parallelism.

Figure 3.1 is taken from [87] and shows what is referred to as the Advanced Machine Model. The boxes labeled "state" refer to the notion of *state* in PCF FORTRAN, which represents the values of all variables and arrays as well as open file descriptors at a particular point in time in the program. The state provides a context for a virtual processor and the chores that are mapped to it. A chore may have its parts executed within multiple states on different virtual processors using the `SPREAD` construct as described above.

3.1.1.2 Parallel Pascal and C

As mentioned above, other parallel procedural language proposals have contained constructs that incorporate parallel looping structures. Simultaneous Pascal [124, 95] is one which provides a `forall` construct as well as a `fork/join` mechanism. The `forall` is essentially a `doall` construct, and includes a mechanism for declaring variables local to

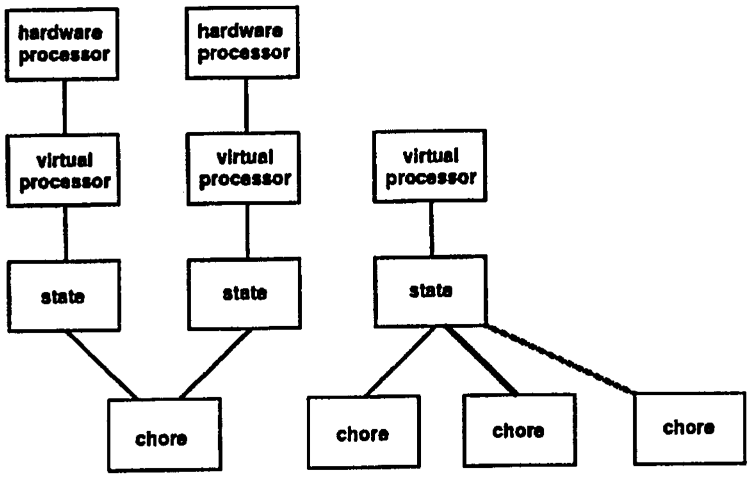

Figure 3.1. PCF FORTRAN advanced machine model.

the parallel loop. The `fork/join` construct contains some number of statements between the keywords `fork` and `join`. Each of these statements may execute independently and in parallel. Simultaneous Pascal also provides a `using` construct for declaring variables with the scope of a single statement, a `locking` statement for defining, locking, and releasing semaphores, and a `traverse` statement, for operating on a dynamically created data structure in parallel.

The parallel constructs of Simultaneous Pascal emphasize fine-grained parallelism, particularly with `fork/join` and `traverse` providing single-statement parallelism. No mention is made in the references about parallelism at the procedure-call level. Also, Simultaneous Pascal does not specify language-level constructs for managing the mapping of parallel program fragments onto processors or virtual processors. This task is left to the SpoC run-time system, which is also described in the references. We discuss this run-time support below in Sec. 3.2.4.3.

Another procedural parallel language which contains constructs for parallel loops is EPEX C [97]. Although this extension of C is centered around parallel procedure calls (which are discussed in the following subsection), `PARFOR` and `PARSECT` constructs are also provided for coding parallel loops and sections. Conventional C `for` statements that are nested in parallel sections may be specified to execute "cooperatively" in the same manner as a SPREAD DO in PCF FORTRAN. EPEX C also provides a DOTOG (do together) construct which is analogous to the fork/join of Simultaneous Pascal. A BARRIER statement is also provided for synchronization. The scope of BARRIER is all parallel "processes" within the parallel section where the barrier is executed. EPEX C also provides a mechanism for declaring shared and private data.

PCP is another C-based parallel language which contains support for parallel loops. PCP supports parallel loops and parallel procedure calls with the splitall and split constructs. However, the semantics of split are not only to indicate potential parallelism in the program, but also to subdivide the number of processors associated with a current thread and assign them to newly created sub-threads. Thus, split defines the amount of real parallelism based on the number of processors available, not on the semantics of the program. PCP also defines support for lock/unlock synchronization, barrier synchronization, and a fetch_and_add operation.

3.1.2 Parallel Procedure-Based Languages

While not as widely studied as parallel loop-based languages or languages for concurrent and distributed computing, several language proposals have been made that are based on the concept of parallel procedures. Perhaps the first such proposal was the extended FORTRAN used for the HEP multiprocessor [120]. This system allowed individual procedures to be created and executed in parallel. A CREATE statement caused a subroutine to execute in parallel with its creator. A RESUME statement in a called subroutine caused its caller to resume execution and run in parallel with the called subroutine. Created subroutines or called subroutines that executed a RESUME statement terminated asynchronously with their creators when a standard FORTRAN RETURN statement was executed. Synchronization between parallel procedures was not implemented with barrier or fork/join constructs, but instead was controlled by access to asynchronous variables. Each memory word in hardware contained a "full/empty" bit, and asynchronous variables provided the programmer with a method to access that bit. Basically, a read of an "empty" variable was suspended until a producer filled it.

Another system that includes parallel procedures is the EPEX C environment developed for the IBM RP3, described above and in [97]. Parallel procedures are created in EPEX C with the `parcall` statement. All available processors execute the `parcall` to create a new parallel procedure until one execution of the `parcall` returns `FALSE`. At this point, no additional parallel procedures, or *paraprocs* are created by this `parcall`, and, when all those already created have completed, the creating procedure or "parent" is allowed to proceed. Each parallel procedure is passed a private copy of the arguments, so if a data structure is to be shared by multiple paraprocs, it must be passed using a pointer variable. There is no data consistency checking or protection between multiple paraprocs, so data integrity must be guaranteed by the programmer using `parcall` control synchronization as well as the `BARRIER` statement.

The number of paraprocs created by a `parcall` is determined by the code of the parallel procedure, which eventually must return `FALSE`. However, additional `parcalls` may exist in paraprocs, allowing a "tree" of paraprocs to be created. Only leaf nodes of the tree may be executing at any given time though, because parent paraprocs must suspend until all children complete. EPEX C defines all parent stack frames to be visible to children, effectively requiring the run-time system to support a *cactus stack* (discussed below, see Fig. 3.2). Thus, any paraproc can refer to any ancestor's local variables.

As stated above, a parallel procedure capability is also provided by PCF FORTRAN [87]. The PCF constructs differ from EPEX C in two major ways. First, as is the case with HEP FORTRAN, a parallel procedure call results in the spawning of only one parallel procedure. Consequently, in order to execute procedures in parallel, the creator does not block at the point of the parallel call. However, unlike HEP FORTRAN, PCF provides a `WAIT ALL CHORES` form of synchronization which enables a creator to suspend until all chores it has created have completed. This construct allows a parent to block until all parallel procedures created have finished, since a parallel procedure call results in a chore creation. PCF FORTRAN is similar to EPEX C in that shared data accessed from parallel procedures must be protected by the programmer. However, it is unclear from the language definition as to whether a cactus stack must be implemented.

Other run-time system packages have been developed that enable parts of a sequential language program to be executed in parallel. These "system" calls are usually language independent, and require explicit calls to be inserted into the original program. This approach is similar to providing support for parallelism at the operating system level, which we believe requires excessive overhead. The generality of these run-time systems also prevents many assumptions about program structure from being made, which in turn prevents speed enhancing optimizations from being made. An overview of these run-time environments is given in [82] and [19].

Another parallel language proposal that we have made uses the parallel procedure idea for distributed-memory multiprocessors (DMMs) [45]. However, in that system, additional constructs are provided in the language that reflect the message passing hardware and operating system. We consider the procedures in this case to be more like "full-fledged" processes, since they have the capability to send and receive messages to/from other processes. Also, as a performance consideration, the location of processes within the DMM can be controlled at the language level.

3.1.3 Concurrent and Distributed Languages

As stated previously, there are a wide variety of programming models available for parallel processing systems. Even when considering only procedural languages, the possibilities include not only languages which incorporate parallel loops and procedures, but also concurrent languages, languages for distributed-memory multiprocessors (or *data parallel* languages), and languages for programming distributed systems.

A concurrent language is one which incorporates a tasking or process model, and typically requires a multitasking run-time system. Examples include Ada [57], Concurrent C [66], and Linda [67]. These languages are intended to be independent of the underlying architecture, leaving the responsibility of supporting the language specification to the run-time system implementation. However, the designs of Ada and Concurrent C are best suited to shared-memory multiprocessors while the design of Linda is intended to facilitate implementation on distributed-memory multiprocessors. It should be noted, though, that some work has been done to support Ada on distributed-memory machines [43, 79] as well as Linda on shared-memory machines [33].

Languages for distributed-memory multiprocessors include concurrent languages with special memory models and constructs for message passing [45] as well as data parallel languages such as Parallel Pascal [112] and C* [114]. While the former model is similar to a concurrent language, the data parallel languages are in-step with the SPMD programming style and provide constructs that enable a single code fragment to run on each processor and access its own data area. Message passing is either implicit or denoted in an intuitive manner, relieving the programmer from coding calls to low-level message passing routines.

In addition to languages for distributed-memory multiprocessors, other languages have been proposed specifically for programming distributed systems, i.e., groups of complete computer systems connected via some local area network. Examples of such languages include SR (Synchronizing Resources) [13], NIL [104], and Lynx [115]. These languages are much like concurrent languages, but they typically provide mechanisms for fault recovery and dynamic reconfiguration of logical communication channels.

3.1.4 Tool-Based Parallel Programming Systems

In addition to compiler-detected parallelism and explicitly parallel languages, work has also been done in the development of tools for specifying program parallelism. These tools usually provide a graphical interface, and enable programmers to construct parallel programs by specifying sequential code fragments and their dependency on each other. The sequential code fragments are usually encapsulated as procedures, for example CODE [29] and Poker [123], but tools are also available for working with loop-level parallelism, for example Pat [15]. These tools are primarily concerned with helping the programmer understand parallelism while providing an opportunity for rapid-prototyping of "portable" parallel programs. Less emphasis is placed on performance, although Pat does interface with the Cray microtasking library (described in Sec. 3.2.4.2 below). Similar work has been done for distributed-memory multiprocessors, where the emphasis is on performance and on relieving the programmer from specifying calls to low-level message passing routines. These are typically intended for use with homogeneous SPMD programs. Examples include Task Grapher [89] and Hypertool [135].

3.1.5 Summary and Analysis

The above subsections have provided a summary of a representative set of language features which enable the programming of parallel loops, sections, procedures, and language-level processes. For the work described in this thesis, we have used the language model for `doall` loops as they would be determined by parallelizing compilers which provide iteration-local variables where necessary. However, in some cases we found that the parallelization tools available to us were unable to locate all potential loop parallelism. This may be overcome by using one of the parallel-loop based languages described above. In addition, our approach is one independent of the specifics of the shared-memory architecture. We also require the run-time management of parallelism to be the job of the run-time system and not the programmer. For those reasons we disregard proposals using the concepts of cluster tasks, spread do loops, and processor group splitting as described above.

In addition to loops, our run-time system design is also intended to support parallel procedures. In order to implement this support efficiently, the semantics of parallel procedures must be carefully defined. Consideration must also be given to the expressiveness of these resulting parallel procedure semantics. With this in mind, we define a set of constructs to enable the programming of parallel procedures in Chap. 6. These constructs resemble those described above, but are still significantly different from other proposals. By providing a create statement which may spawn any number of parallel procedures without blocking the parent, we provide a way to express more parallelism than the proposals described above. However, we also restrict the form of synchronization to create and merge statements in order to provide a language and run-time system environment which prevents deadlock. The issue of deadlock is not seriously addressed in any of the imperative parallel language proposals described above. Complete details of our parallel language constructs are given in Chap. 6.

3.2 Parallel Run-Time Support

As stated in Chap. 1, other researchers have developed and implemented run-time support mechanisms for parallel programs. However, much of this work has focussed on OS-level support or system libraries, which we have characterized as an inefficient approach to supporting parallelism. On the other hand, some studies have described language specific run-time systems, but most have either focussed on theoretical issues or have not revealed proprietary implementation information. The remaining work on language specific run-time systems has been done in non-commercial research environments, where experimental systems have been described in moderate detail. In the following subsections we will briefly describe those other research efforts, placing particular emphasis on the experimental systems. We begin with library and OS-level run-time support, followed by user-mode language-specific multitasking and microtasking run-time systems. It is the microtasking systems with which we are most interested, since that is the focus of this dissertation.

3.2.1 Library Defined Support

A survey of library-based multitasking run-time systems for FORTRAN can be found in [82]. This article describes experience using these systems, not their implementation or performance. Two microtasking systems, from Cray and IBM, are also included in the survey, and we discuss these further below.

The Uniform System for the BBN Butterfly Machines [131, 19] is defined as a library of run-time support routines. However, these routines execute in user-mode for the most part, thus avoiding the high overhead of operating system calls. The system assumes that one operating system process is available for each processor. A similar run-time system is the Chameleon system developed at the University of Washington [8]. This system includes language directives that enable data partitioning among parallel code sections. However, the specification of parallelism and its implementation is similar to the Uniform System, which we describe in more detail below.

The Uniform System is centered around a set of code and data elements that define the portions of a program that may execute in parallel. A *task generator* is a data structure that describes an amount of parallel work. The data structure contains a *task generation procedure*, a *task work procedure*, data, and fields for bookkeeping. The task generation procedure is essentially the code that is executed in order to schedule the next instance of a task associated with the task generator. It also generates the parameters and performs any other initialization for the task being scheduled. The task work procedure is the code that actually forms the body of the task. The data in the task generator is for all tasks created in conjunction with the generator to share, and the bookkeeping fields are used to keep track of the progress of all these tasks. Parallel activity is started by placing a task generator into shared memory. However, the reference makes no mention of how or where multiple generators are placed in memory. Processors spin on local locks waiting for parallel work to be created. Local locks are updated in a tree-like fashion when work is created, i.e., one processor updates locks for two other processors, and so on.

The Uniform System library also provides a collection of routines called *activators* for creating and initializing task generators. There are a number of specific activators that allow predefined "types" of generators to be created, as well as a *general purpose activator* for specifying any type of generator. For example, a call to the specific generator `GenOnI (task_work, range)` indicates that a task generator should be created with task work procedure `task_work` and, as part of the task generation procedure, each task should be given a unique actual parameter equal to an integer between 0 and `range`. Essentially, the name of the activator called encodes the method for which parameters will be prepared for the generated tasks. As a result, this also determines the number of tasks created. Other activators enable passing of the elements of a lower triangular of a matrix or generating only as many tasks as processors. Clearly, these activators are designed with specific applications in mind.

Calls to the general purpose activator specify a task generation procedure, a task work procedure, a pointer to common data, generator initialization and finalization procedures, a flag indicating whether the generator is synchronous or asynchronous, a flag indicating whether the generator is abortable, and a processor limitation. Initialization and finalization procedures allow generator specific code to be executed before creation of any tasks and after completion of all tasks. Synchronous generators cause their creator to block until all generated tasks have completed. Asynchronous generators allow the creator to continue after the generator has been created. An abortable generator may be removed and cease to create tasks before all have been generated. A processor limitation allows restriction of the number of processors that may work on a generator, regardless of the number of tasks to be created. Specific activators assemble parameters based on their function and in turn call the general purpose activator.

Because of the Non-Uniform Memory Access (NUMA) architecture of the Butterfly, several additional routines are present in the Uniform System to dynamically allocate memory in different partitions across the local memories of each processor and in shared memory. The reference also shows several examples of how a compiler can translate parallel FORTRAN constructs into calls to the Uniform System.

3.2.2 Multiprocessor Operating System Support

In Chap. 2 we stated that multiprocessor operating systems that allow parallelism in user programs are appearing. Examples include Carnegie Mellon University's Mach [26], The University of Rochester's Psyche [116], and Sequent Computer Systems' Dynix [23]. Each of these systems allows programmers, through system calls, to access the functions of thread or lightweight process creation, interprocess communication, shared memory allocation, thread scheduling options, and synchronization. However, this approach, while fully functional, is cumbersome and inefficient, incurring system call overhead for basic operations. While some optimization is possible, for example, the reuse of created threads for other program tasks, these run-time support mechanisms are still best used by a language specific user-mode run-time system. One effort aimed at reducing the high system call overhead by supporting multiprocessor OS primitives outside of kernel mode when possible is underway at the University of Washington [11]. Still, this approach is OS oriented, and can benefit from language specific run-time support which avoids any calls or interrupts to the OS, whether in or out of kernel mode.

3.2.3 Multitasking Systems

As mentioned above, multitasking systems have been built for multiprocessor computers. These run-time systems typically support language-level processes or procedures as restricted forms of processes, both with preemptive semantics. As mentioned in Chap. 2, these systems also require considerable support for timing, priorities, interprocess communication, and synchronization.

In addition to the multitasking systems described in [82], Other multitasking systems exist which are defined by high-level language semantics and therefore are transparent to the user. Commercial implementations of parallel Ada exist for both the Sequent Balance [99] [2] and Encore Multimax [64]. Research work has been done toward implementing Ada and Ada-like run-time systems on distributed-memory multiprocessors [51, 48, 79, 45]. Work has also been done implementing Concurrent C [52] and Linda [33] on both shared-memory multiprocessors and distributed systems. A description of Ada run-time system components appears in [43], Concurrent C is described in [52], and Linda is described in [67]. In the case of Ada, additional work has been published regarding the performance of individual language features, with some discussion of their relation to run-time system components [42, 46].

3.2.4 Microtasking Systems

3.2.4.1 Cedar Project

A significant amount of work in the area of dynamically self-scheduled microtasking run-time support has been done by Polychronopoulos at The University of Illinois as part of the Cedar Project [106, 110, 107, 108]. This work has focussed primarily on scheduling techniques for the iterations of parallel loops and has led to the development of guided self-scheduling, introduced in Chap. 2. This research has also described a decomposition of parallel programs into task graphs and discussed the relation between nodes in the graph, the synchronization it requires, and the combining of nodes to reduce scheduling points where possible. This is discussed in more detail in Chap. 5.

The work by Polychronopoulos has also contributed a theoretical framework for quantifying the tradeoffs in different scheduling algorithms. A result of this theory is a proof

[2] A parallel Ada system also exists for the current Sequent Symmetry series.

that guided self-scheduling provides optimal performance. However, this proof relies on several simplifying assumptions, e.g., synchronization and scheduling times are constant, that do not necessarily hold true in real systems. While simulations back up this claim, they embody most of the assumptions and are at a sufficiently high-level that the issue of data reference locality is not accounted for. In later chapters, our experimental testing shows mixed results for guided self-scheduling, but it still appears to be a useful technique.

3.2.4.2 Commercial Efforts

Implementation work on microtasking run-time systems has been done for commercial systems by Cray[3] [54, 82], IBM [32, 82], and Convex [36]. Other computer system vendors have also been developing microtasking-style run-time systems including Alliant [127], Tera [130], Sequent [100], Apollo (Hewlett-Packard), Stardent, and Encore. While the first three companies have revealed some information on the structure of their run-time systems, they have revealed little about actual implementation details of their proprietary systems. Most published material centers around the programming requirements to use these systems, which are typically some combination of automatically compiler-detected parallelism and the addition of compiler directives to explicitly denote parallelism. These systems emphasize loop-level parallelism, although most will allow procedure calls inside microtasks if compiler directives are supplied that allow data dependencies to be ignored. These systems are also usually limited by disallowing the nesting of microtasking constructs, enabling an efficient implementation using run-until-completion scheduling semantics. The above references, with perhaps the exception of [36] and [127], also say little about the operating system role, except that it must supply multiple processors and shared memory to the parallel program. Performance of these systems is also discussed very little, except for some isolated loop results on a two processor system discussed in [32] and a report of

[3] From the date of the citations it appears that Cray coined the term microtasking.

megaflop rates on a Cray Y-MP system with up to 8 processors executing a hand parallelized and vectorized version of the Perfect Club benchmarks [1, 70]. Development that addresses some of these above issues, especially nesting and suspension of microtasks, is undoubtedly underway at these companies. However, a comprehensive account of these efforts and their results has yet to be published.

As stated above, Convex has published the most to date about the structure of their run-time system and its interface to the operating system [36]. For the C240 system, the Unix kernel was modified substantially to take advantage of a scheduling mechanism that makes extensive use of special purpose shared or "communication" registers. There are a fixed number of "sets" of communication registers, and each set can "mount" a Unix process by copying key elements of its context including pointers for virtual memory addressing. Each set of registers also includes a set of "thread identifiers", which are used to set up addressing of private storage for each thread. In this case, a thread is what we refer to as a microtask. With this context information mounted, any processor can contribute to the parallel work of any process by executing one of its threads. Appropriate values from the communication register set are used to ensure proper virtual memory addressing. The reference does not discuss the situation where a process is waiting to be mounted and no space is available in the register set, or if processes may be unmounted before they have completed.

The Convex approach, called Automatic Self-Allocating Processors (ASAP), is one where multiple process contexts are simultaneously resident with respect to each processor. This is a more flexible approach than other efforts which rely on mounting multiple process contexts simultaneously, the first being the HEP multiprocessor [120], with a newer version under development called Horizon [130]. These other systems are finer grained, with a single processor fetching successive instructions on the behalf of different execution threads.

3.2.4.3 Experimental Run-Time Systems

Experimental run-time systems that can be considered as microtasking have been described in more detail [124, 95, 97]. These run-time systems were developed in conjunction with the Simultaneous Pascal and EPEX C language efforts described above. The language and run-time system for Simultaneous Pascal has been termed SpoC, for Simultaneous Pascal on Concert, where Concert is the Harris Corporation's Concert Multiprocessor. Some performance data has also been reported for SpoC. In contrast, it is unclear how much development was actually done on the EPEX C system described here, since it was designed for the IBM RP3, an experimental machine which did not reach production status. Both the Concert Multiprocessor and the IBM RP3 are NUMA machines with a hierarchical arrangement of local and remotely accessible memory as described for the BBN Butterfly above.

SpoC

The SpoC run-time system supports the parallel `forall` loops, parallel statement execution, and parallel pointer manipulation described above. The reference [124] implies that the created microtasks are scheduled using run-to-completion semantics, but since nesting of parallel constructs is allowed which requires a parent microtask to wait for child completion, run-until-block semantics are actually used. However, the references say nothing about the blocking and resumption of microtasks and their effect on the scheduling algorithm.

Each microtask created in the SpoC system is referred to as a thread. Each thread is provided with a *frame*, which is essentially an execution frame commonly associated with a procedure call. Each frame contains standard procedure call information required by Pascal, in addition to information such as a parent frame pointer, an outer scope pointer to support addressing of variables at multiple nest levels, a pointer to a synchronization counter, a pointer to a "semaphore" that protects that counter, an address where the code to implement a join resides, and a variable length portion for thread local data. Frames are essentially preallocated memory areas that are grouped into multiple linked lists called *frame pools*. Each frame pool is associated with a particular processor, so multiple processors do not contend for the same memory during frame allocation, unless the allocating processor's list of frames are all being used. This last condition requires locking to ensure mutual exclusion during frame acquisition.

After frames are acquired and initialized, they are placed in the *work pool* which is essentially a run-queue. A processor schedules work from a frame by loading its values into its registers where possible. For the `fork/join` construct, a frame is created for each statement in the construct. For `forall` loops, one frame is created for the loop, and each thread schedules a frame, acquires a unique loop index, and then reinserts the frame into the work pool. No detail is provided about how this scheduling and reinsertion process can be overlapped by multiple processor schedules, or how the thread-dependent local data area of the frame is managed for multiple iterations. The `traverse` construct is similar to `forall` in thread creation, except a frame is created for each independent branch of the data structure to be traversed. For example, to traverse a binary tree, each node creates two frames to expand the traversal down each subtree. Each scheduled thread also has access to a local stack for procedure calls and interrupt handling, however, no description is given as to how that stack is managed in relation to suspension and rescheduling of threads.

Although the SpoC frames contain considerable context to support the sharing of variables at multiple nest levels among other things, at least one example program has been run with favorable speedup results [95]. The reference reports results for a Mandelbrot image processing application and a Gaussian elimination application. Two versions of each program were produced using two different parallelization strategies at the language-level. Speedup for Gaussian elimination peaked at about 7, while the two versions of Mandelbrot peaked at 20 for 21 processors and 58 for 63 processors. The reasons given in the reference for speedup limits include data locality problems and excessive thread context, although they claim that run-time system and scheduling overhead is not significant. The approach suggested to address these problems is based on hardware modifications, including the addition of a thread-context coprocessor and a structured memory controller which incorporates information about the nature of SpoC data structures.

The SpoC references make no mention of the operating system role or its interface to the system. Based on the description of the hardware environment in [124], it appears to be a system which enables a program and run-time system to be downloaded to a collection of "bare" processors by the host computer, effectively making Concert a single-user parallel computer.

EPEX C for RP3

The EPEX C system for the IBM RP3[4] describes a run-time system to support the constructs for program parallelism described above [97]. However, no performance data is reported, and the system may not have been fully implemented. The system is divided into two major functional areas, 1) the scheduler, and 2) a memory allocation facility that supports a cactus stack.

Parallel procedures in this system (or simply microtasks) are scheduled from a central ready queue of entries kept in priority order which contain a pointer to code, a pointer to parameters, flags indicating readiness and completion, and a lock for ensuring mutual exclusion of access. These entries are placed in the queue by the parent, and all available processors access this entry to call the procedure until one call returns FALSE. When this happens, further calls do not occur, and the parent waits for all other calls to finish before removing the queue entry and proceeding. While other processors can execute any available work in the queue, a parent procedure must wait for completion of children after the call

[4] This system is distinct from the EPEX FORTRAN for the IBM 3090 mentioned in the microtasking references cited above.

to the child that returns FALSE. However, since nesting is supported, multiple parents may be blocked at any given time, wasting processing resources.

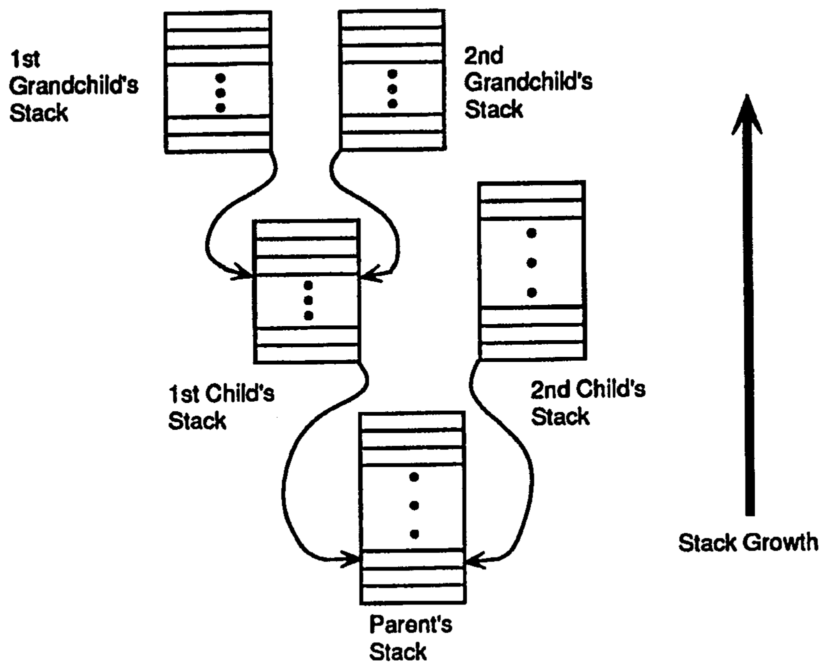

Figure 3.2. Example cactus stack.

In addition to scheduling a microtask by reading information from the queue entry, a creation of a branch on the cactus stack must also occur in the run-time system. The cactus stack allows multiple procedure calling frames to be linked together, so that variables of parallel procedures can be referenced at multiple nest levels. An example cactus stack is shown in Fig. 3.2. Although [97] says little about how these branches are created or managed, it does point out that a parent procedure cannot complete and deallocate its branch until all of its children have completed. This is necessary since child procedures may reference the local variables of the parent.

Only a few implementation details of this EPEX C system are given in the reference. Also, performance data is not supplied apart from a few comments about the performance of certain run-time system functions. These include the suggestion that programmers strive for large grain procedures complete with synchronization statements. The idea is to limit parallel procedure calls which are costly, containing several hundred instructions to create.

The wasted time by scheduled parents waiting for child completion is also an issue which encourages a limit on the amount of parallel procedures actually created. Deadlock is also mentioned as a potential problem, but nothing is said about preventing it. The authors do make the point, however, that this is an experimental system which has not been designed with efficiency goals in mind. There is also an implicit assumption that a fixed number of processors are assigned to a parallel job and no further discussion of the operating system interface is given.

3.2.5 Summary and Analysis

As described in the above paragraphs, a notable amount of research and development work has been done in the area of parallel run-time systems, including microtasking systems on shared-memory multiprocessors. However, in each case, there is either a deficiency of detail in describing the implementation, a limit on capabilities, a dearth of performance data, or design inefficiencies. In this dissertation, we will address these issues, and present an efficient run-time system design which will support run-until-block semantics for both parallel loops and procedures with nesting, without wasting processing resources with resident blocked microtasks. We present a new stack-based scheme for managing blocked microtasks, which also allows sharing of data between ancestors and descendants without the complexity of a full cactus stack implementation. We also provide an approach to frame allocation and initialization that enables fixed-length frames to be allocated quickly. In each case, we provide working assembly code which explicitly defines our implementation. And, in addition to extensive performance data, explained in the following section, we also discuss the design for an interface to a multiprocessor operating system.

3.3 Multiprocessor Performance Evaluation

As mentioned in the introduction, numerous performance evaluation studies of various types have been done for shared-memory multiprocessors. However, for the most part, these studies have used very different program models from our work, where code is parallelized and run-time support implemented using ad hoc techniques and/or the SPMD style. As a result, these programs conform to a large grain parallel programming model. In fact, many of the studies referenced below have borrowed codes from the same set of of parallelized CAD applications for circuit layout and verification. Also, these studies have relied primarily on trace-based simulation, which calls into question the accurate representation of synchronization activity in the case where the simulated architecture is different from the one where the traces are collected [25].

Most detailed performance studies of parallel programs executing on shared-memory multiprocessors have differed greatly from our work in at least one aspect. These studies have concentrated on different style architectures, e.g., bus-based systems [62, 63], used traces from one system to simulate a generic machine [118], or used unrealistic assumptions about cache block size, e.g., one word, cache size, e.g., infinite, or cache write-back implementation, e.g., assume sectored in a non-sectored system [101, 132, 4, 35, 40, 72]. Such assumptions eliminate the problems that must be solved in real computer systems with the implementation of cache consistency in the presence of false sharing. Another study which does incorporate execution-driven simulation and an architecture similar to ours, also uses a different programming model (as stated above) and presents data in terms of cache access times instead of program running time and speedup [98]. Also, other studies have been done on real multiprocessor systems that are concerned with synchronization performance [71, 10, 12]. However, these studies rely on synthetic benchmarks to evaluate the performance of several synchronization-related activities. Finally, other studies have been done which incorporate a fine-grained parallel model similar to ours and have used real program execution to measure speedup [70, 8, 14, 32, 124]. These studies, however, do not report performance data in the detail we investigate.

One study that has combined a fine-grained programming model with detailed hardware architecture directed performance evaluation is described in [128]. This study analyzes the impact of the cache coherence protocol and synchronization on the performance of two fine-grained parallel applications and a multiuser workload. This work was done using hardware-based performance monitoring tools on the Sequent Symmetry bus-based shared-memory multiprocessor. Data is reported on bus traffic, cache miss rates, and program speedup. Synchronization is via cached-based locks, which also participate in the coherence protocol. The bus utilization measurement is of primary importance in this study, as it determines the scalability of the system. This shared resource is used to transfer data between a processor cache and main memory, or between two caches. This data traffic also includes synchronization related activity. The cache miss rate is a major determinant of bus utilization. This study says little, though, about the structure of the parallel code, and states explicitly that it is not focussed on application or run-time support tuning. As a result, the reference notes that the overhead of parallelization has a significant impact on performance.

While the studies mentioned above are valuable and provide insight into the performance of parallel programs on multiprocessor systems, our goals in this phase of our research differ in two major ways. First, we are concerned with the performance of fine-grained parallel programs which use microtasking run-time systems. In order to measure this performance, we have designed and implemented a run-time environment and language constructs for creating the necessary parallel programs. Second, the results of our performance studies are intended to unveil the interaction between speedup of a parallel program and the sources of run-time overhead attributed to both hardware and software mechanisms. While other studies attempt to project the raw performance of a cache memory subsystem or a synchronization mechanism, our work is aimed at measuring these components of run-time overhead in the context of a given fine-grained parallel program, in addition to the overhead associated with the mechanisms employed by the run-time system. Our experiments measure the individual impact of each source of overhead for a set of parallel programs, and attribute to each source a percentage slowdown of these programs versus ideal speedup. As stated in Chap. 1, these sources of overhead are broken down into three categories: 1) Static Load Imbalance, 2) Run-Time Scheduling and Synchronization, and 3) Cache Consistency Overhead. A combination of software restructuring and execution-driven simulation has enabled us to perform the necessary tests, which vary both the cache consistency technique and the run-time system design. Our testbed is based on the Astronautics ZS series, a shared-memory multiprocessor with a crossbar interconnect between processors and memory. This system also provides several sets of shared registers which enable fast synchronization and enhance the performance of the run-time system. Also, although some simplifying assumptions are inevitable when simulating a prototype system, care has been taken in our work to ensure that these assumptions do not misrepresent the impact of false sharing or the overhead associated with synchronization.

3.4 Summary

In this chapter we have surveyed other research efforts concerned with design, implementation, and performance evaluation of parallel programs and programming systems on shared-memory multiprocessor systems. We have classified the work into three major categories: programming languages, run-time systems, and performance evaluation. Overall, we see that each study or research project is concentrated on one or at most two of the major areas. One of the goals of our work is to study the issues relevant to all three areas in the context of one study. In the following chapters, we explore each of these areas as they relate to experimentation with a fine-grained, microtasking-based, parallel processing system.

CHAPTER 4

MOTIVATION AND APPROACH

In this chapter we motivate our experimental study, and describe our testbed environment.

4.1 Motivation

As stated in the last chapter, we are interested in exploring the performance potential associated with loop-based and procedure-based parallel programming models. Our aim is not only to verify the efficiency of our run-time support code, but also to examine the performance characteristics of the complete system, and to identify performance bottlenecks.

In our studies examining loop parallelism, we were able to examine incremental performance degradation due to different parts of the system. Because the loops used in this set of experiments were considered in isolation, we were able to exploit our *a priori* knowledge about the code's execution on a uniprocessor system to aid us in determining the effects of parallelization overhead. Together with our ability to eliminate (or "short-circuit") overhead from some parts of the system in the simulator, we were able to identify the incremental overhead contribution from three major "barriers", or sources of "impedance", to parallel program speedup: 1) load balancing; 2) run-time scheduling and synchronization; and 3) cache consistency overhead. The evaluation of the performance based on the relative overhead amounts due to each category enables us to identify performance bottlenecks based on application code characteristics. It also helps us verify the efficiency of our run-time support code.

In contrast, we are only able to determine an amount for cache consistency overhead for our parallel procedure tests, because of the large programs used and the highly dynamic nature of our parallel procedure codes. However, the testbed described below and the techniques used for parallel program construction and cache consistency simulation apply to all of our experimental testing.

4.2 Performance Barriers

As stated above, there are three barriers that impede the performance of parallel programs which we categorize as: 1) load balancing; 2) run-time scheduling and synchronization; and 3) cache consistency overhead. We discuss these in greater detail next.

4.2.1 Load Balancing

A parallel processing system has a perfectly balanced load when all processors are actively participating in a parallel computation and no processor is left idle. This ideal is rarely attained. For example, load imbalance can be caused in the execution of a single parallel loop when the number of iterations is not an exact multiple of the number of processors available: some processors are left idle while waiting for the "leftover" iterations to complete. This *static imbalance* is aggravated when the ratio of iterates to processors is small and the grain size of each iteration is fairly large. Such factors increase the number of idle processors and the time they must wait. The situation can be further hampered by the effects of cache misses and memory bank conflicts, which effectively vary the execution time of each iteration, and also by the run-time scheduling algorithm, which may distribute iterations unevenly amongst the processors. On the other hand, it is possible for this *dynamic imbalance* to improve load balancing, by compensating for static imbalance. However, even in this case, the running time of the program is still increased. In our tests, we identify the impedance due to the static imbalance that decreases load balancing and therefore efficiency. The effects of dynamic imbalance is measured in the context of the other overhead categories described below.

4.2.2 Run-Time Scheduling and Synchronization

The run-time support code that is necessary to enable parallel processing of `doall` loops or parallel procedures is an additional source of overhead that degrades performance. In addition to the instructions that must be executed as part of this code, another major source of overhead in the run-time support is synchronization. Although this effect is minimized in our experiments by the shared "semaphore" registers of the ZS (described below), long idle times for processors waiting to synchronize can still be observed. For each microtask, the time to execute the run-time support code and perform any necessary synchronization must be small compared to the time to execute the application code of the microtask, or the potential benefits obtained from parallelization will be lost.

4.2.3 Cache Consistency

The third major impedance limiting the performance of parallel loops is the overhead required to keep caches consistent. As is clear from the description of cache consistency techniques given above in Chap. 2, rather elaborate schemes are required that have the potential to incur large amounts of overhead. Depending on the technique used, large variations in cache consistency overhead can be observed when processor reference locality is altered due to the run-time scheduling technique. The presence of cache consistency techniques may also degrade performance significantly when a program is run on a single processor and consistency actions are unnecessary.

An issue related to cache consistency is the choice of cache line size. While a large line size may enable a single processor system to take advantage of spatial locality in data references, it may also increase the amount of false sharing present in a multiprocessor system.

Increased false sharing results in an increase in the invalidation rate and, consequently, a decrease in the cache hit rate. Cache misses that occur as a result of a previous invalidation are referred to as *invalidation misses* [62]. The choice of line size has a lesser effect when software consistency is used because a "worst-case" static view to the problem is taken that is largely independent of run-time data movement.

4.3 Measuring Performance Barriers

Using our simulator testbed (described below), we are able to obtain detailed and highly-accurate performance measurements for the execution of parallel loops and procedures. Also, through simulator modifications, we can effectively short-circuit certain system effects in order to factor out the contribution to the overall performance degradation due to the three performance barriers listed above in the execution of parallel loops. To achieve this, we make the following three kinds of measurements.

4.3.1 Inherent Parallelism

In order to evaluate the performance of the overall parallel processing system, we first determine the best parallel performance possible for a given `doall` loop. This "inherent" parallel performance is determined by executing a subset of the iterations on a single processor with no cache consistency actions or run-time support code. The subset of iterations selected for execution are those that represent one processor's allocation with the best static load balancing for a given number of processors and fewest cache misses. By comparing this performance rating to what would be possible with perfect linear speedup[1] we can compute the impedance due to static imbalance only. This is a function solely of the number of iterations and the number of processors, hence the term inherent. It can be used,

[1] That is, N processors run the program in T/N time, where T is the time required to run the program on one processor using the best sequential algorithm.

in general, to establish an upper bound on parallel performance for any program containing parallel loops.

4.3.2 Assumed Consistency

The next step in the overhead measurement process is to determine the impedance due to the run-time scheduling and synchronization functions. For each scheduling algorithm tested, we construct the parallelized code for the `doall` loop by inserting the run-time support code at the assembly language level. By executing this code on the simulator without any support for cache consistency (assumed consistency), we can determine the impedance due to run-time scheduling and synchronization by subtracting out the running time for the inherent parallel case described above. The scheduling effects include overhead in executing run-time code, additional load imbalance caused by non-optimal dynamic schedules, and additional cache misses due to memory reference locality. The impedance produced by synchronization in this configuration includes the time to execute synchronization instructions and the time lost by memory bank conflicts among the processors.

4.3.3 Complete System Simulation

In the next phase of measurement, we add the support for cache consistency. Simulation under these conditions gives us the performance of the complete system as it would actually appear. By comparison to the assumed consistency case, we get a measure of the impedance due to cache consistency support. This impedance is caused both by explicit actions to enforce consistency and by an increase in cache miss rates due to invalidation misses. At this stage of simulation we can also measure the impact of the major cache parameters, e.g., cache line size and set associativity, on both pure and invalidation miss rates, which in turn affect overall performance.

4.4 Experimental Testbed

4.4.1 The ZS Series

The Astronautics ZS series of computers are based on a proprietary 64-bit processor designed with numeric applications in mind. The processor is heavily pipelined and is capable of issuing two instructions per clock period. Memory accessing and floating point are decoupled by using distinct instruction issuing streams for each. Memory accesses are also buffered using processor queues for integer and floating point loads and stores. These features permit dynamic scheduling between addressing and floating point functions, and successfully hide memory latencies in many cases [122]. This type of architecture is referred to as Decoupled Access/Execute (DAE) [121].

The ZS-1, a uniprocessor, was completed (both hardware and software) and has been operational for some time. The ZS hardware is constructed to support systems containing up to 16 processors, but multiprocessing software is incomplete. ZS-series multiprocessors use a set of shared "semaphore" registers for low-overhead interprocess communications. Simulations reported in this dissertation use accurate timings derived from the actual ZS multiprocessor hardware.

Each processor in the ZS system is provided with a 128K byte data cache. The interconnection network in the ZS multiprocessor system is essentially a crossbar network. The data path is four words (256 bits) wide, and is optimized for 16 word (one cache block) transfers. To support references to non-cacheable data, smaller transfers, down to one byte, can be accommodated, but the timing for smaller transfers is the same as for a full 16 word transfer.

ZS Series Multiprocessor Hardware	
Clock period	45ns
Data cache size	128K bytes
Data cache organization	2-way set associative
Data cache block size	128 bytes, 16 double precision words
Memory size	128M bytes
Memory access time	675ns
Processor/Memory interconnect	crossbar
Simulator error	5%

Table 4.1. ZS hardware and simulator characteristics.

4.4.2 The Multiprocessor Simulator

The multiprocessor simulator is a register-transfer-level simulator and program interpreter. Based on the instruction and address information supplied by the interpreter, the busy times of the functional units, load and store queues, cache, main memory, registers, and pipelines are modeled. The states of these subsystems are advanced each clock period in accordance with any dependencies that are present. A file of system parameters is used by the simulator to define cycle time requirements for the functional units, memory access, and the queues. In addition, parameters such as cache line size and data associativity, functional unit requirements for each instruction, and the processor clock speed can be varied. Except where noted below, the parameter values used in the ZS multiprocessor simulations included a processor clock speed of 45ns, a 2-way set associative data cache of 128K bytes with a block size of 128 bytes, and an 8-way interleaved main memory with an access time of 15 clock periods. The simulator reports a single processor megaflops (MFLOPS) harmonic mean rating of 4.0 for the 24 Livermore FORTRAN Kernels on the ZS-1. Table 4.1 lists the major characteristics of the ZS hardware and the multiprocessor simulator.

In the uniprocessor case, we found the simulator timing results to be consistently faster than actual running time on the ZS-1 by about 5%. This discrepancy is due to the omission of address translation faults and operating system scheduling interrupts in our simulations. In order to avoid different versions of a "warm cache" between the different hardware configurations tested, the data cache for each processor is flushed before execution of the timed loop. This provided each test case with identical starting positions.

The parallel programs used for our experiments make use of the ZS's shared semaphore registers for synchronization. There are 32 sets of registers, each set consisting of 32 registers of 32 bits. Several fetch-and-op instructions are provided that operate on the semaphore registers. Instructions are also provided for the first 8 semaphore registers in each group that enable processors to block until the value of a particular register becomes either positive or negative. Together with the ability to read and write these registers, these instructions make the semaphore registers ideal for holding addresses, implementing barriers, and computing indices for a microtasking run-time system.

4.5 Parallel Program Construction

Our parallel programs were developed using sequential code and a knowledge of data dependencies and parallelization boundaries. To detect candidate codes for parallel loop execution we used the Kuck & Associates KAP data dependency analyzer together with the hand parallelization diaries used by Grassl in [70]. However, we found some instances of parallelism which were possible but not detected. We added this parallelism into our codes. These tools were very helpful in detecting the parallelism using a loop-based language model.

Our parallel procedure codes also used both a loop-based program (where loops are essentially converted into procedure calls) and a recursive code for a Quicksort program which has independent sub-problems at each recursion. A sequential FORTRAN Quicksort program could not benefit from the tool for automatic detection of parallelism because it implements recursion at that language level using several GOTO statements and an array managed as a stack. This structure gives the appearance of dynamic data dependencies which in fact do not exist. This prevents the parallelizer from making useful assumptions about the code structure which allow parallelism. Instead, the Quicksort program had to be hand parallelized at the language level using our parallel language constructs (introduced in Chap. 6) and knowledge of the algorithm.

The sequential base codes used for the programs were compiled on a ZS-1, and then disassembled. At the assembly source level, we added instructions to implement the self-scheduling based run-time support. This code conversion was debugged using a ZS-1, which we have on-site. Of course, some bugs only manifest themselves when the code is executed on multiple processors. These conditions were detected by verifying the results of the simulations. Although bugs of this sort are time-consuming to locate, they occurred infrequently and were hence not a significant problem.

Because each processor executes the same code, the run-time support code is responsible for synchronizing these multiple threads of execution and assuring that each processor executes a unique subset of the total work to be done. The run-time support code added to the parallel programs was written to be independent of the number of processors used to execute it. This code only assumes that a count of the number of available processors is provided in one of the semaphore registers.

The simulator initiates the execution of each processor at the entry point of the object file. The start code of this object is essentially a piece of run-time system code that synchronizes the processors and dispatches all but one to the self-scheduling kernel. The last processor executes the entry code of the program itself, and eventually gets to the code that sets up the first parallel execution, thus enabling the other processors to schedule work. These other processors have no environment of their own worth saving, and simply use registers and share the main stack environment when necessary. The details of this implementation vary somewhat between loop support and procedure support, and this is described in further detail in Chaps. 5 and 6 respectively.

Implicit in this style of execution is the notion of one operating system process or thread per processor. Hence, our number of virtual and physical processors is the same. Also, since no system calls are made, and no other parallel programs are present in the system to cause preemption, there is no OS-level context switching necessary. Almost all programs tested have a running time of less than one second in real-time, which is a reasonable value for an observed time between interrupts on one processor in a multiprocessor system[2]. Further, the simulator does not model address translation faults or interrupts, which also allows us to avoid entrance to the operating system. Traps are not intended to occur in our system, so they simply create an error condition which halts the simulator. Because the run-time support is supplied at the user-mode and is either generated inline with the code or linked together with it as a set of small routines, there is never a need in our simulations to enter the operating system. The modeling of system calls and OS-level context switching would require a port of an appropriate multiprocessor operating system to the ZS. An approach to merging our parallel run-time system code with a multiprocessor operating system is discussed in Chap. 7 below.

4.6 Implementing Cache Consistency

In our experiments, we evaluated two software and two hardware consistency schemes. The software schemes consisted of 1) making result data non-cacheable, and 2) using local memory management instructions to implement write merging. The ZS provides mechanisms to make pages of virtual memory non-cacheable as well as several instructions for allocating and flushing cache blocks. The hardware schemes we evaluated are two variants of a central directory-based approach to cache consistency. In order to evaluate the hardware schemes accurately, the ZS simulator was modified to incorporate them. These consistency techniques are described in more detail below.

[2] Although an interrupt may occur as often as once every millisecond to check for schedulable OS-level processes, these interrupts may only be felt by one processor, possibly idle and not allocated for use by a parallel program. Hence, an observed time between interrupts of 1 second is not unreasonable.

4.6.1 Software Consistency

While the first software cache consistency scheme is straightforward, the second warrants some further explanation. It utilized the "allocate block" and "flush block" instructions to manage a local copy of the final result data in each cache. Each processor allocated enough blocks to hold the entire result array at a temporary virtual address distinct from those used by the other processors and distinct from the final result area. As a side effect of the allocate block instruction, each word in each block is initialized to a value of zero. Each processor then proceeds with its share of the parallel loop iterations, performing one or more scheduling operations depending on the self-scheduling technique used. After all iterations have been scheduled, the first processor to complete its work proceeds by flushing its entire result area to main memory at a virtual address reserved for the result using multiple flush block instructions. Each successive processor to complete its iterations then, in mutual exclusion with other processors, reads each 64 bit word from the result area, performs an "OR" operation between that word and the corresponding word stored at its temporary location, and places the result at the result area's virtual address. After each word is read and updated, the cached result area is flushed to main memory.

An alternative approach to software consistency would be to flush blocks as they are written in the body of the loop. Techniques for this style of consistency have been proposed in the literature [40, 56]. However, due to the fact that loop iterations are allocated dynamically with our run-time support, and that the compiler-produced code accesses words in blocks in a non-sequential fashion, we would have to completely rewrite the generated code, realign all program variables, and restrict iteration allocation in order to ensure consistent results. This problem would be simplified if individual words could be flushed instead of blocks, or if the word size and cache block size were the same (as assumed in [40]). However, reducing the cache block size to such a small value would eliminate the positive effects of spatial locality obtained with a multiword block. A policy of prefetching one word blocks may compensate for this problem [88], but this requires sophisticated compilers and architectural modifications that are beyond the scope of our study.

Several properties of the codes we simulated enabled us to use the temporary result area approach to consistency. The most important property is that while any number of processors may need to access the same logical block, no two processors require access to the same word within a given block. (As mentioned in Chap. 2, this property is referred to as false sharing.) This creates a situation where each cache contains a temporary result area with words that contain either zero or a final result, and, no two caches contain a final result in the same logical word. This enables the OR instruction to be used to merge the temporary result areas. This approach to data consistency has been used in various forms in other systems, e.g., the Myrias SPS-2 [24].

If the temporary result areas are too large to remain in the cache along with the other referenced data for the duration of the loop iteration executions, each processor must allocate its temporary result area at a different virtual address. If block replacement in the cache then effects the result area, its values will not be corrupted by collision with another processor's temporary results. This approach may require that a large virtual memory space be available to the parallel program.

4.6.2 Hardware Consistency

The hardware cache consistency schemes we evaluated are based on the central directory approach proposed in [34]. While this approach is similar to the one originally proposed by Tang [126], it requires fewer cache tags and a smaller central directory. Surveys of directory-based cache consistency techniques can be found in [4] and [102].

The hardware consistency scheme implemented requires a central directory with an entry for each block. This entry contains several bits that indicate which caches contain the block and whether or not the main memory copy is up-to-date. In one scheme, originally proposed in [34], each directory entry contains one bit for each cache in the system to indicate the block's presence in that cache. There is also one bit for each entry that indicates whether main memory is up-to-date with a cached block. If it is not, this bit is set, and the block may reside in only one cache. This consistency scheme does not require any additional cache tags than those already provided by the hardware. These include a valid bit, a dirty bit, and least recently used bits. Figure 4.1 shows the tags associated with a cache block at both the processor cache and main memory.

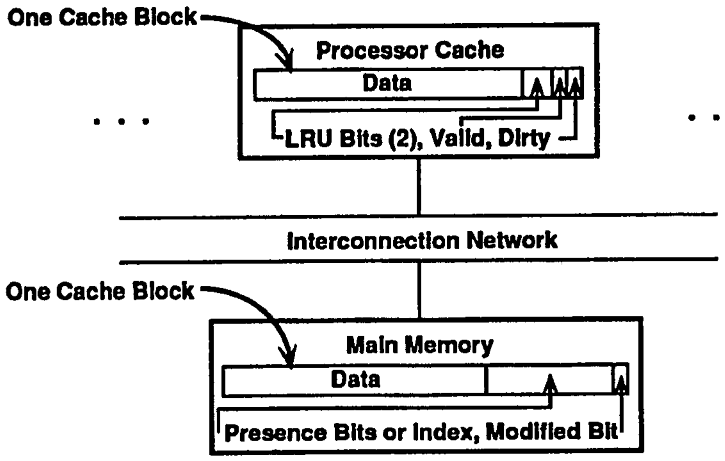

Figure 4.1. Block tags for directory-based cache consistency schemes.

A write miss makes a block dirty and consequently exclusive to that cache. The directory must be consulted to invalidate any other cached entries. A write hit must also consult the directory and propagate invalidations if the block is not already dirty (and thus private). The modified bit in the directory must also be set when a write to a block is performed. Read hits and dirty write hits may proceed without accessing the directory. A read miss requires that the block's presence bit be set in the directory for the requesting cache to indicate that this block is now present in that cache. Read and write misses must also supply the up-to-date value of the block to the requesting cache. If the modified bit is set in the directory on a cache miss, main memory must read the block from the cache containing the up-to-date value and clear the associated dirty bit before supplying the data to the requesting cache.

A variation of this scheme proposed in [4] was also tested. Instead of providing a presence bit for each cache in each directory entry, only enough bits are provided to encode an index to one cache in the system. If there are *P* processors in the system, this alternate scheme reduces the number of presence bits from P bits to $log_2 P$ bits as shown in Fig. 4.1 and mentioned previously in Chap. 2. However, this scheme provides exclusive access for each cached block and requires an invalidation whenever a processor accesses a block that is cached elsewhere in the system.

4.6.3 Simulating Hardware Consistency

As mentioned above, our experiments for evaluating hardware cache consistency techniques required some modifications to the simulator, effectively changing the architecture of the ZS. We also made several simplifying assumptions to avoid a detailed redesign of the hardware. The first assumption is that no race conditions exist between checking local cache tags and accessing the central directory. The simulator updates the cache tags, the directory, and performs invalidations at the point of the cache miss. The processor in this case then idles the required number of cycles to simulate the time taken to perform these updates. Any subsequent accesses by other processors always see the most up-to-date cache tags and directory values. Because individual words are not shared in our test programs without explicit synchronization, subsequent block accesses by other processors need not wait for previous ones to complete before updating cache tags and directory values. The first processor will cache the block and load the referenced word into the load queue, but the block will no longer be valid. Overall, the effect of eliminating race conditions is to force an arbitrary serialization of events in a manner that is easy to implement. It relieves the simulator from re-examining requested transactions a second time when they reach the directory, possibly modifying their effect, e.g., causing or canceling an invalidation, based on the values present in the directory.

Although cache tags and directory entries are updated instantaneously, processors must wait additional cycles before accesses are complete if the requested memory bank is busy or the block to be accessed is dirty in another cache. We assume that the directory is interleaved across the memory banks so that the block requested resides in the same memory bank as its directory entry. This provides mutual exclusion for directory entries, since only one processor may access a given memory bank at any one time. We also assume that the directory can be updated, any necessary invalidations can be sent, and main memory can be read all in the time it takes to perform a main memory access. If the requested memory block is dirty in another cache, we assume that memory can be updated and the value supplied in one additional main memory access time. Since the requesting processor has control of the main memory bank when the modified bit is checked, we do not queue the request to write the up-to-date cache block back to main memory and update the directory.

Our race-free implementation does not model the increased network traffic for invalidations, nor does it suspend a writing processor to wait for an invalidation acknowledgement. The lack of additional network traffic modeling makes our results slightly optimistic, but we believe this is not a serious omission, since our experiments are not characterized by heavy invalidation traffic.[3] Also, because invalidations are not acknowledged, our system performance is slightly more optimistic than it should be for modeling a weakly-ordered system. (Recall from Chap. 2 that a weakly ordered system is one which guarantees a consistent value of a shared memory word provided access to it is protected by explicit hardware synchronization. However, the simulator does implement a weakly-ordered system with the help of the QUIET instruction contained in the programs, which stalls the processor until all pipes are empty and all memory operations are complete, and the performance gain by our simplified modeling of cache invalidations should be slight at best.

These simplifications to the modeling of hardware cache consistency make the simulation of the shared consistency scheme equivalent to how we would simulate the scalable broadcast-based scheme proposed in [16]. This is so because invalidation traffic is not modeled, and write request traffic is modeled as one memory access time to read the value from the appropriate cache. While the broadcast-based scheme certainly requires more

[3] Except in the case of private hardware consistency, where performance is already significantly degraded by excessive invalidation-misses. See the cache hit rates in Tables 5.9 and 6.2.

interconnection bandwidth and will also suffer more network contention, our simulations of the shared hardware scheme provide an idea of how the broadcast-based scheme could perform in an ideal situation. Also, as mentioned in Chap. 2, there are other more scalable directory-based schemes that may closely match the performance of the shared hardware scheme.

Finally, the simulator provides interlocks and memory bank arbitration so that banks are accessed in mutual exclusion. Bank conflicts will add delays to the completion of requests, and these times are effectively added to the base memory access times we assume for directory accesses. While our assumptions about race conditions, the ability to send invalidations, the ability to read up-to-date blocks in other caches, and the ability to add a directory to the memory system most certainly simplify our experiments, we believe that these tests do reflect the additional delays brought about by cache misses due to invalidations and memory accesses to modified blocks.

4.6.4 Major Assumptions

The major assumptions made in order to perform our simulations are summarized in Table 4.2. As stated above, we believe these assumptions are reasonable for determining relative performance of different cache consistency techniques and dynamic scheduling algorithms. The overall impact of these assumptions will be to over predict performance by a small amount. However, since most of our results are presented in terms of speedup, the predictions will be accurate, since single processor tests make the same assumptions.

We can summarize the impact of the major assumptions listed in Table 4.2 individually. The absence of OS interaction is consistent with the relatively short running times of the programs, and the absence of other work on the system. This provides a consistent, repeatable testbed environment which provides an upper bound on performance. The times to consult the directory and implement a copy-back due to a cache miss to a modified block (implied copy-back) are simulated using average memory access time as a baseline, but

Major Assumptions for Multiprocessor Simulator
No OS interaction.
1 OS process per processor.
Central directory part of memory.
Directory access time is 675ns, same as memory.
Data cache copy-back time for a miss to a modified block is 675ns.
Directory and cache tags are updated instantly.
Implied copy-back network traffic is not modeled.
Invalidation network traffic is not modeled.
Invalidations are not acknowledged.
Bank conflicts are modeled for directory access.

Table 4.2. Major assumptions for simulation.

any further delays caused by network contention or race conditions in updating cache or directory tags are omitted. However, further delays caused by memory bank conflicts on the directory access are modeled by the simulator. The overall impact of these modeling assumptions are that the reported execution times are slightly optimistic. However, the execution times reported by the simulator are equivalent to that observable in a real system when no race conditions or network contention occur. This is not unreasonable given the frequency of cache misses and contention on a crossbar interconnection network.

4.7 Summary

In this chapter we have described our methodology for filtering the impedance due to different sources of overhead found in a parallel processing system. Our simulation testbed has also been thoroughly described. In the next two chapters, we describe our design, implementation, and testing of loop-based and procedure-based parallel programs and their run-time support.

CHAPTER 5

LOOP PARALLELISM

5.1 Introduction

In this chapter we discuss the performance issues relating to loop-level parallelism and evaluate their impact based on the results of simulations run using the testbed described in the previous chapter. We begin by describing the loop model of parallelism used in these experiments. The implementation of the run-time support for these programs is then described. The following sections discuss the loops used in our experiments, and provide detailed performance data from the simulation testbed. These sections also examine the performance breakdown into the three major overhead categories discussed in the previous chapter. The relative performance of the different cache consistency techniques and scheduling algorithms is also examined.

5.2 Language Model

In each of our example programs, the loops used were `doall`, so there are no cross-iteration dependencies requiring synchronization. This allowed us to use the run-to-completion semantics compatible with the simple microtasking model, which enabled the use of the length one run queue described in Chap. 2. However, this approach prevented the use of nested parallel loops, which prompted us to use a policy of only parallelizing outermost loops, as described in [106].

The parallel loops used in our experiments are all based on existing code from sequential programs. Because we compiled the code with an optimizing compiler for a uniprocessor configuration, the ZS-1, some modifications had to be made to the resulting assembly source in order for the program to be parallelized. These changes were minor for the first three tests which each involved a single `doall` loop kernel. The most common change required improving the compiler's register allocation so that private copies of local data for each iteration were not kept at identical memory locations on the stack. If there were not enough registers to complete this task, a local stack area was set up for each microtask to store local data. These techniques prevented the need to declare local variables as arrays at the language level.

For the last parallel loop code simulated, a complete program taken from the Perfect Club Benchmark Suite [1], more significant changes were required at the language level. The TRFD program uses a single large common block to store all arrays. In some cases identical array slots were used to store local data for multiple iterations of a parallel loop. To overcome this, the common block was expanded, and the array indexing pattern was modified. The impact of this expansion is described in Sec. 5.8 below.

For the three kernel tests, the identification of the desired `doall` loop was straightforward. However, the selection of `doall` loops in the TRFD program was much more difficult. In addition to the array overlaying technique described above, this program also contained array references where the indices were themselves array references. Further, these second-level array references were to arrays whose values were not initialized until run time. Consequently, our attempt to use a data dependency analyzer came up very short in identifying the maximum possible level of parallelism based on loops[1]. However, with the help of a hand parallelization diary kept as a part of the work described in [70], we were able to parallelize all major outermost loops[2]. As a result of this parallelization strategy, the

---

1 A study which identifies this type of shortfall can be found in [117].

2 Except the loop around the entire program, since this would require a nesting capability. This is discussed further in Chap. 6

data memory area had to be increased (as described above), and some integer variables that were incremented inside of loops had to be initialized with a proper starting value before entrance into the parallel loop. This modification essentially makes these values private to each microtask.

5.3 Run-Time Support

5.3.1 Scheduling

The test programs for our experiments were parallelized by allocating microtasks that executed some subset of the iterations of the outermost `doall` loop. We tested three different dynamic scheduling policies, all based on the concept of processor self-scheduling. The first scheme, which we simply call self-scheduling, creates one microtask for each iteration of the outermost `doall` loop. The second, which we refer to as chunk scheduling, computes an "optimal" chunk size by dividing the number of iterations for the outermost `doall` loop by the number of processors available. If the numbers do not divide evenly, an extra "chunk" of the leftover iterations is also created. The third technique, called guided self-scheduling, allocates a number of iterations equal to $\lceil \frac{R_i}{p} \rceil$ where R_i is the number of iterations remaining to be scheduled at step i and p is the number of processors [106, 110]. The run-time support code for each of these techniques can be found in Appendix A.

For self-scheduling, one processor places the starting address and the maximum iteration value in separate semaphore registers. At the starting address, just before the loop body, the beginning iteration value for a microtask is converted into a "base value" to be used for the loop. This base value could be, for example, a base address for array accesses, or an iteration count modified to reflect a non-unit stride. This step is required since loop iterates are scheduled with N indices uniquely assigned a value between 1 and N, even if the loop index does not begin with 1 or has a unit stride. The code at the end of the loop is also modified. Because each microtask only executes one loop iteration, the code of the loop body is simplified by eliminating the test at the end of the outermost loop that normally would determine whether or not the last iteration for that microtask had been executed.

After reading the semaphore registers and before loop execution, another semaphore register is decremented. When this counter reaches zero, it indicates that the semaphore registers can be reloaded with values pertaining to the next parallel loop for execution. While the code to load the semaphore registers before the loop is executing, all other idle processors are waiting to enter the scheduling code. When the semaphore register holding the total number of iterations to be executed is written and becomes greater than zero, the waiting processors begin executing a fetch&decrement operation on the register to acquire a unique index for that microtask.

The chunk scheduling code is similar, but requires more steps. The code that loads the semaphore registers before the loop must also compute the chunk size. This results in a longer setup time for the loop during which other processors participating in the execution of this loop must wait. Also, each microtask must compute both beginning and ending iteration values based on the chunk size and its scheduling index. Because each microtask may execute a number of iterations, the code to test for completion of the iteration range must be restored to the loop body. This code was removed in the self-scheduling case.

The code for guided self-scheduling is a little more complex. As part of the process of acquiring a unique index for each microtask, several computations must be made to get the proper chunk size as well as the starting and ending values for the iteration range. Because the global number of iterations remaining is the basis for these computations, they must be performed in mutual exclusion until this global value can be updated. This effectively creates a critical section of several assembly language instructions to perform a self-scheduling operation. In addition to holding addresses and key index values, the semaphore registers are also used to implement a binary semaphore that ensures mutual exclusion for the scheduling operation. While the chunk scheduling approach also requires a chunk calculation, it is performed only once before any microtasks begin. For guided self-scheduling, a chunk calculation must be performed at each scheduling point, thus increasing the scheduling code critical section from one fetch&decrement instruction to a lock acquisition and several integer arithmetic instructions.

The run-time support code in some cases involved additional instructions beyond that required for the scheduling techniques. In some codes, several values and addresses used "globally" by all iterations are computed before entrance into the loop. These values are computed by one processor during the loop setup and then loaded into additional semaphore registers. Each microtask then reads this "mini-context" into its processor's own register set and/or stack space before it begins execution. If there is a shortage of semaphore registers, pointers to a globally shared data area or stack frame may be used instead. Alternatively, each microtask may compute its own mini-context. This last approach is desirable if it can be completed in less time than copying semaphore registers.

5.3.2 Microtask Graph

As an aid to visualize the transformation of the parallel loop code into microtasks, we can construct a microtask graph which is similar to the task graphs presented in [106, 133, 107]. The microtask graph shown in Fig. 5.1 provides detail down to the basic block level. A rectangle represents a sequence of statements that constitute a basic block as defined in [6]. The rectangles with rounded corners represent parallel loops or a collection of microtasks associated with a parallel loop. These units are referred to as *tasks* in [106]. The dashed arcs between basic blocks represent actual execution dependencies determined by the data dependencies in the source code and the length-one queue implementation. The other solid arcs represent branches (both conditional and unconditional), and the dotted arcs represent branches that are possible with chunk or guided self-scheduling, where multiple loop iterations may be scheduled at once.

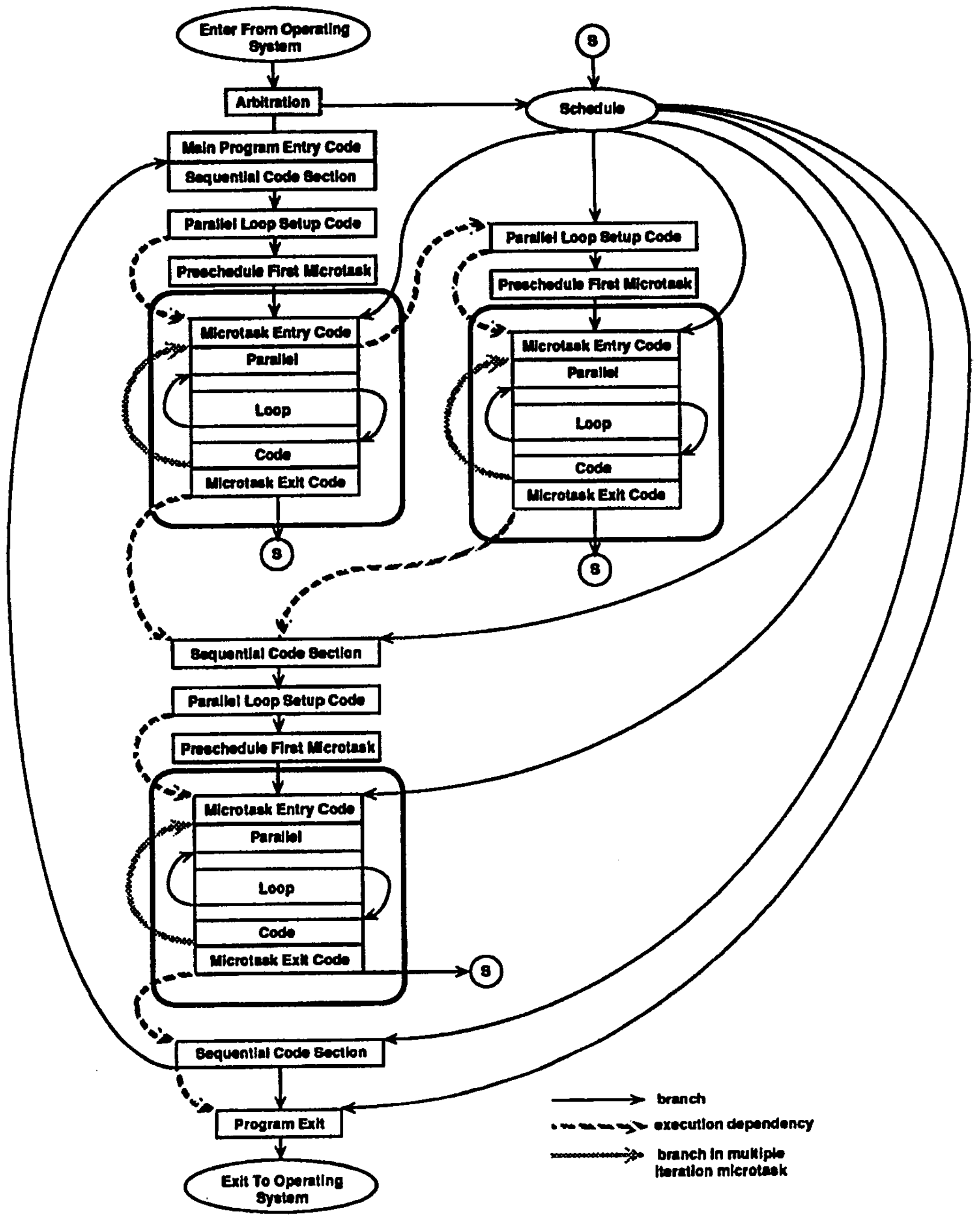

Figure 5.1. Microtask graph for parallel loop program.

In Fig. 5.1, the basic blocks, in groups of one or more, are labeled with the part of the program they represent. The blocks labeled as "arbitration", "setup code", "preschedule", "entry code", or "exit code" are essentially run-time system units to be generated in-line with the program's sequential code. The sequential code is represented by the blocks labeled "parallel loop code", "sequential code section", or "main entry code". The ellipse labeled "schedule" represent execution in the self-scheduling code of the run-time system.

As stated above, the squares with rounded corners in Fig. 5.1 represent tasks which are a collection of microtasks. Although execution dependencies may exist between blocks contained in different tasks, branches can only exist within tasks, between task code and the schedule code, or between blocks not contained in tasks. This last condition allows execution of entire tasks to be repeated. This condition would occur if, for example, an outer loop existed around several inner parallel loops. When task execution is repeated, we say that multiple *instances* of the task will exist during the lifetime of the program. When an execution dependency exists between a block within a task and a block outside that task, the block at the base of the arc must be executed for the last time per that instance of the task before the block at the end of the arc may be executed.

In order to explain the meaning of the graph more clearly, a description of the execution of the code represented in Fig. 5.1 by multiple processors follows. All processors enter the program from the operating system, and the first piece of code selects one processor to continue while dispatching the others to the self-scheduling kernel code. The single processor then continues with the program execution, allocating and initializing variables if necessary. Any sequential code at the start of the program is then executed. In this example, a sequential loop surrounds two procedure calls, and the procedures contain parallel loops. The procedure calls and returns are simply embedded in the code labeled "sequential code section" in the diagram. After the sequential code, the setup code for the first parallel loop is executed. After the setup, the other processors waiting in the scheduling code are now able to begin allocating the microtasks that constitute this parallel loop. This is indicated by the dashed execution dependency arc between the setup code and the microtask entry code. Instead of entering the scheduling kernel itself, the processor that sets up the parallel loop preschedules the first microtask and begins execution immediately.

In this example, the second parallel loop is not data dependent on the first. However, since only one parallel loop may be set up in the semaphore registers at a time (as a consequence of the length-one queue), there is an execution dependency between the microtask entry code of the first parallel loop and the set up code for the second parallel loop. This requires that the last microtask for the first parallel loop be entered before the next loop may be set up. Set up and scheduling of the second parallel loop proceeds as before. Also, there is an unconditional branch at the end of the microtask exit code to the self-scheduling kernel, as indicated by the circle labeled "S".

Before the third parallel loop shown in Fig. 5.1 can begin execution, the first two must complete, and the following sequential section must be executed. This is indicated by the execution dependency between the microtask exit code of the first two parallel loops and the following sequential code. Another execution dependency exists between the end of the third parallel loop and the sequential section that follows it. At this point, either the program proceeds with an exit, or a branch back to the beginning of the program is executed.

The execution dependencies are shown in the graph to indicate where multiple processors may participate in execution of the program. As shown in Fig. 5.1, execution dependencies point to the beginning of basic blocks where processors may branch from the scheduling kernel. These arcs indicate where synchronization between the body of the generated code and the scheduling kernel is necessary. They do not indicate branches. Blocks that are not pointed to by an execution dependency arc are instead pointed to by a branch arc. This implies a sequential execution dependency, where the block is executed by the same processor as the one that executed the previous block. This condition also holds for blocks shown as being adjacent to other blocks that lie underneath it. Also, we should note, that even though this microtask graph can be created at compile time and used to guide run-time support code generation, it does not represent a run queue that can be created statically. The code for each loop is still generated in program order. Consequently, each loop set up section is visited in program order, and the run queue is created and managed dynamically as described above.

5.4 Source Programs

To date, we have measured the performance of three parallel loops. The first two are different versions of a simple matrix multiply based on the 21st Livermore FORTRAN Kernel (LFK21) [94]. The third program is the Gauss-Legendre subroutine (Gauleg) from the Numerical Recipes [111]. It is used to compute a set of weights of a polynomial function over a given interval that can be used to compute the integral of the polynomial.

5.4.1 LFK21

We produced two versions of generated code for LFK21 by interchanging the order of the do loops. The first version of the code moves the innermost do loop to the outermost do loop in order to encourage the compiler to unroll inner loops and to make parallelization of the program easier. This resulted in the FORTRAN code shown in Fig. 5.2. The value of n used for the experiments described here was 100. It is this outermost loop limit that determines the number of microtasks that will be created, e.g., 100 in the simple self-scheduling case.

```
      dimension PX(25,101), CX(25,101), VY(101,25)

      do 15 j = 1, n
      do 15 k = 1, 25
      do 15 i = 1, 25
         PX(i,j) = PX(i,j) + VY(i,k) * CX(k,j)
   15 continue
```

Figure 5.2. Code for version 1 of LFK21.

This version produced the fastest code in the sequential case for several reasons. In addition to enabling the compiler to unroll the innermost loop, the indexing patterns in this case produced a unit stride in each of the three arrays. With the large (128 byte) cache line size, this version warmed the cache quickly and produced a minimum number of cache misses. The compiler could not accumulate partial results in registers, because the innermost loop accesses a different element of the result matrix on each iteration. However, this did not degrade performance, because the low number of cache misses combined with the ZS's dual instruction issue capability enabled the average number of clock periods per instruction (CPI) to approach 0.6 while producing a MFLOPS rating of 12.4. The number of instructions dynamically executed for each outermost iteration was 3485, of which 55% referenced memory, i.e., were loads or stores.

Although the version of the code shown above is the fastest in the uniprocessor case, its heavy use of memory caused performance problems when running on multiple processors with consistent caches. To provide a contrast, we tested an alternate version in which the "k" and "i" `do` loops were interchanged. We chose this restructuring to keep the variable length `do` loop as the outermost one. This allows a parallelization strategy that creates `n` microtasks (for self-scheduling) of a fixed granularity, the same technique as used in the previous version.

In this new version, each microtask computed the final result for one column of the PX array, which enabled partial results for each element to be accumulated in a register. It also enabled different microtasks to avoid accessing values of the PX array that were in the same cache line. The disadvantage of using this version of the code is that it produced a non-unit stride access pattern for the VY array. Although this version of the code produced only 4 additional data cache misses and 2 additional instruction cache misses for `n=100`, it ran quite a bit slower than the previous version, averaging slightly less than 1 CPI and 8.6 MFLOPS. The number of instructions executed for each outermost loop iteration was 3280, with 38% of those accessing memory. The source of the slowdown for this version of LFK21 was a reduction in instruction execution overlap (pipelining) due to busy registers during the multiplies and adds.

5.4.2 Gauleg

The Gauleg subroutine contained more instructions, but the code referenced shared data

Iter.	Avg. Inst. per Iter.	Mem. Ref.	CPI	MFLOPS
5	782	7.9%	1.7	7.0
16	1589	3.8%	1.5	9.2
25	2244	2.7%	1.5	9.6
50	4063	1.5%	1.5	10.0
160	11475	0.5%	1.5	10.4

Table 5.1. Sequential code performance and characteristics for Gauleg.

structures much less frequently (see Fig. 5.3). The `doall` loop for this code was the `DO 12` loop. After a detailed calculation, each iteration of the `doall` loop wrote four result values into different single precision (4 bytes per element) floating point arrays (`X` and `W`). Although many other variables were used inside the loop, each one was either a read-only parameter (`N, X1, X2, XM, XL`), or a variable local to that iteration (`I, J, P1, P2, P3, PP, Z1, Z`). This enabled extensive use of registers and kept the percentage of memory references low – ranging from 7.9% for 5 iterations down to 0.5% for 160 iterations. (The number of iterations in the Gauleg subroutine corresponds to the number of weights that are calculated.) The execution time of each iteration varies and cannot be determined at compile time, because Newton's method is used and its convergence rate is data dependent. The net result is that microtasks execute a varying number of instructions regardless of the scheduling algorithm used. Each iteration of Gauleg averaged between 782 instructions for a 5 iteration version to 11,475 instructions for a 160 iteration version. The performance of the ZS system on this program for various numbers of iterations is given in Table 5.1.

5.5 Inherent Parallel Performance

The inherent parallel performance for LFK21 versions 1 and 2 with `n=100` is summarized in Table 5.2. In the table, P is the number of processors used, S is the speedup, E, or S/P, is the efficiency, and MFLOPS is the total number of megaflops achieved using all

```
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*4 X1,X2,X(N),W(N)
      PARAMETER (EPS=3.D-14)

      XM=0.5D0*(X2+X1)
      XL=0.5D0*(X2-X1)
      DO 12 I=1,(N+1)/2
        Z=COS(3.141592654D0*(I-.25D0)/(N+.5D0))
1       CONTINUE
          P1=1.D0
          P2=0.D0
          DO 11 J=1,N
            P3=P2
            P2=P1
            P1=((2.D0*J-1.D0)*Z*P2-(J-1.D0)*P3)/J
11        CONTINUE
          PP=N*(Z*P1-P2)/(Z*Z-1.D0)
          Z1=Z
          Z=Z1-P1/PP
        IF(ABS(Z-Z1).GT.EPS)GO TO 1
        X(I)=XM-XL*Z
        X(N+1-I)=XM+XL*Z
        W(I)=2.D0*XL/((1.D0-Z*Z)*PP*PP)
        W(N+1-I)=W(I)
12    CONTINUE
```

Figure 5.3. Code for Gauleg.

the available processors. Recall from Chap. 4 that this performance rating is extrapolated from the results of executing a subset of iterations on a single processor with a minimum of cache misses and no run-time support overhead or synchronization. The subset of iterations is chosen based on an optimal static schedule that obtains the best possible load balance.

In the case of Gauleg, the optimal static schedule accounted for the fact that different iterations of the `doall` loop were of differing granularities. The inherent parallelism for each version of Gauleg is summarized in Fig. 5.4. The MFLOPS rating for this code ranged from 7.0 using 1 processor for 5 iterations to 161.4 using 16 processors for 160 iterations.

It is clear from the tables and chart that, even though there is optimal load balancing of outermost loop iterations, there is still a limitation on speedup. This limitation is primarily due to the fact that optimal load balancing still leaves some processors idle some of the

	Version 1			Version 2		
P	*S*	*E*	MFLOPS	*S*	*E*	MFLOPS
1	1.0	1.00	12.4	1.0	1.00	8.6
2	2.0	0.99	24.7	2.0	1.00	17.2
4	3.9	0.98	48.9	4.0	0.99	34.3
8	7.4	0.93	92.2	7.6	0.95	65.4
12	10.6	0.88	131.0	10.9	0.91	93.9
16	13.3	0.83	165.7	13.9	0.87	120.0

Table 5.2. Inherent parallel performance for LFK21.

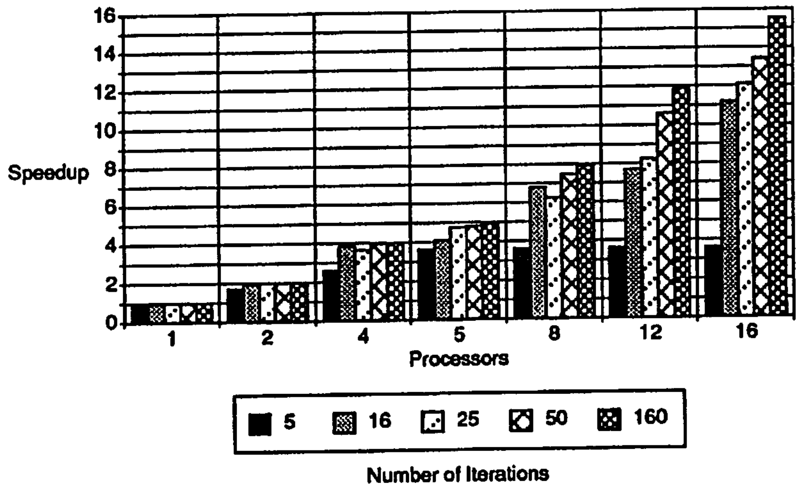

Figure 5.4. Inherent parallel performance for Gauleg.

time. However, the largest instance of Gauleg does come close to 100% efficiency. Another factor that affects the speedup when using larger numbers of processors is the increased number of overall cache misses. If only one processor is used, the cache is warmed by the initial iteration so that subsequent iterations do not miss the cache. On the other hand, if the number of processors used equals the total number of iterations, each processor will be allocated one iteration and it will have to start with a cold cache.

5.6 Performance With Assumed Consistency

Figures 5.5 and 5.6 show the loop speedups for the parallelized version of each program and problem size tested together with inherent parallelism for comparison. The speedups for LFK21 show performance close to inherent parallelism for guided self-scheduling and self-scheduling. Chunk scheduling performance is significantly worse, due to its inferior dynamic load balancing. Figure 5.5 also shows the speedup of the second version of LFK21 when it is calculated based on the single processor performance of version 1. This compares the speedup of version 2 to the "best sequential algorithm". However, we also compute the speedup for version 2 as compared to itself, in order to demonstrate the performance possibilities for other codes with similar memory referencing patterns and grain size. In fact, it is the relatively low percentage of memory references per microtask that enable the speedups of version 2 (when compared to itself) to be slightly greater than version 1. The results for tests with n=50 and n=25 show the same trends, but the speedups using 8, 12 and 16 processors are about 18% less for n=50 and 30% less for n=25 (see Appendix B).

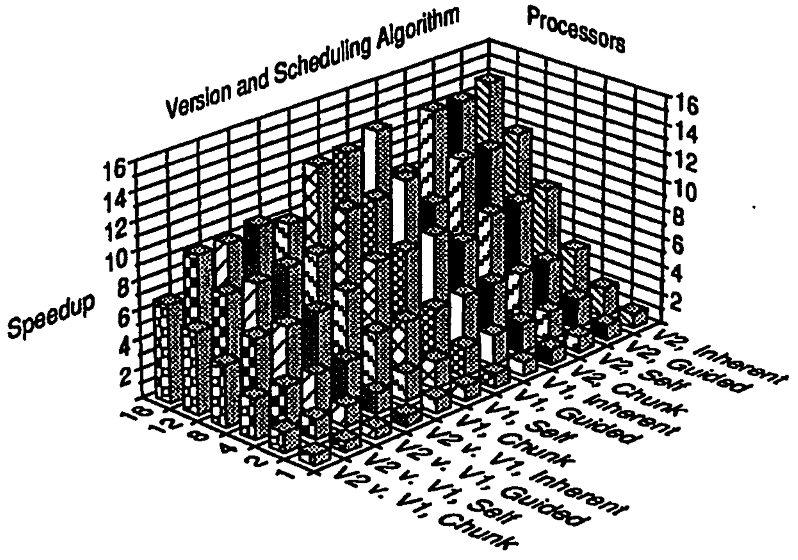

Figure 5.5. Speedups for both versions of LFK21, n=100, assumed consistency.

The speedups for Gauleg shown in Fig. 5.6 indicate that self-scheduling provides the best run-time performance. For problem sizes of 50 and 160 iterations, this dynamic scheduling technique is close to inherent parallelism. The performance of chunk and guided self-scheduling for all sizes of Gauleg is less than self-scheduling. This is due to

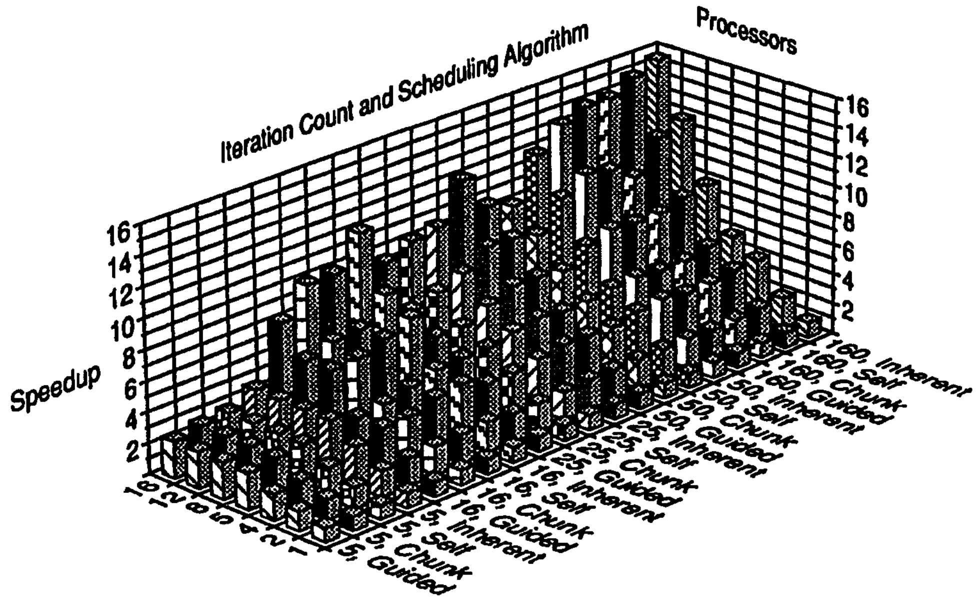

Figure 5.6. Speedups for all versions of Gauleg, assumed consistency.

the imperfect dynamic load balancing of these techniques, which is further aggravated by the variable-sized grains of the iterations of Gauleg.

5.7 Complete System Performance

5.7.1 Overview

Figures 5.7 through 5.10 show the speedups of both LFK21 versions with n=100 and two instances of Gauleg for all scheduling algorithm and cache consistency technique combinations. In Figs. 5.7 and 5.8, there is a great variation in performance depending on the consistency technique employed. For the first version of LFK21, the hardware consistency techniques are much better than the software techniques. The performance of the non-cacheable data approach is heavily degraded by the high number of memory references in each microtask. The write merging technique (software consistency) is slowed by the time it takes to merge the fairly large result areas for the double precision result array.

The "adjusted" technique refers to a calculated result for write merging, where it is assumed that the data overlaying can be performed pairwise among processors in parallel, requiring $\log_2 P$ steps. Because the iterations of LFK21 are all the same number of instructions, we can compute the adjusted performance of write merging by adding a base time equal to that required for all microtasks to complete the computation to the time required for merging assuming the pairwise approach. Although the pairwise approach would require additional synchronization, memory space, and possibly hardware support, we calculate this value as an upper bound on performance. However, as is clear from Figs. 5.7 and 5.8, this adjustment to write merging does not provide much improvement.

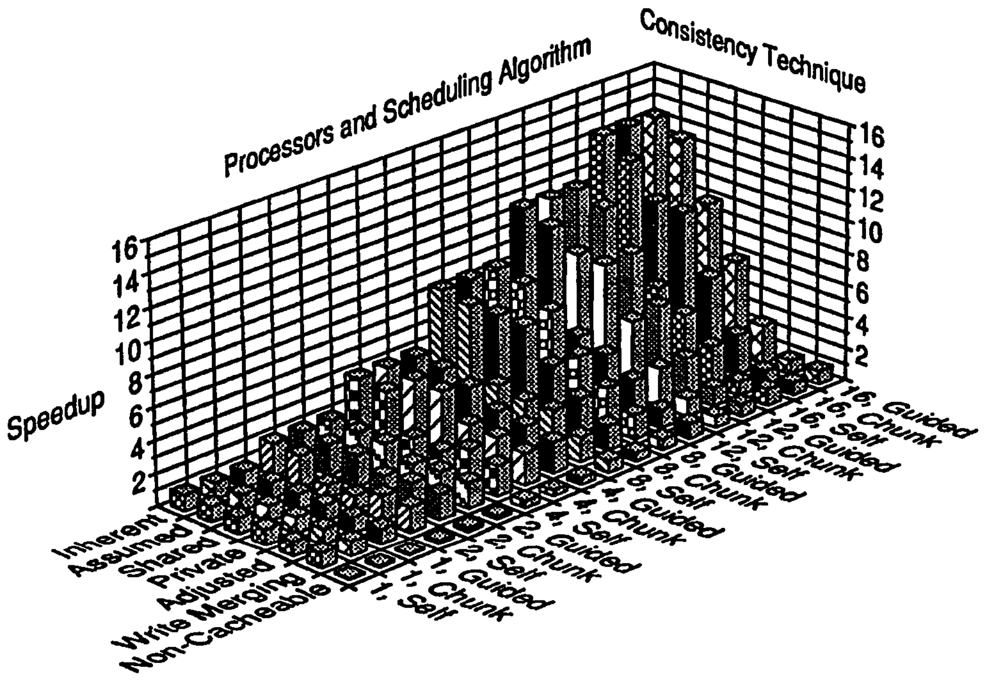

Figure 5.7. Speedups for LFK21, version 1, n=100.

The results for version 2 of LFK21 shown in Fig. 5.8 are quite similar, except that the speedups for the non-cacheable data approach are much greater, and the speedups for private hardware consistency are much worse. For the other techniques, the speedups are slightly greater than they were for version 1. The increased speedups in each case are due to the reduced number of memory references for version 2, and, the reduced performance

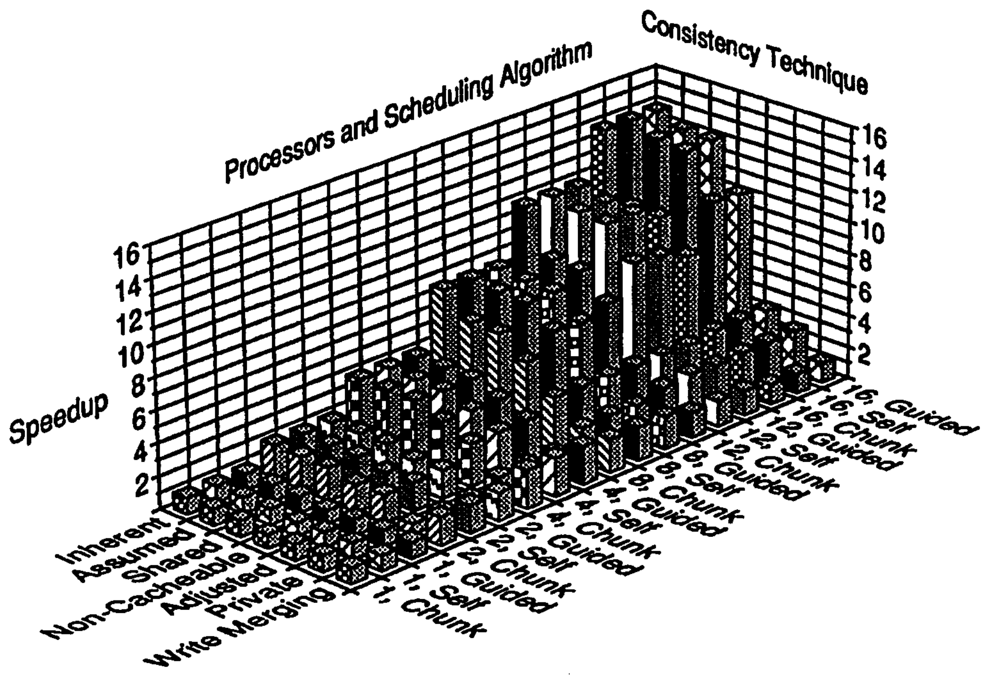

Figure 5.8. Speedups for LFK21, version 2, n=100.

of private hardware consistency is due to the non-unit stride referencing pattern for one operand in the multiply. This causes multiple references to the same block by each processor at different times in the computation. Because private hardware consistency invalidates all other copies when any processor references the block, there are many more invalidation misses occurring, even on the read-only operands of the multiply. Despite the improved speedups for the other consistency techniques used with this version of the code, it is important to note that version 1 still provides the best overall performance in terms of MFLOPS in most cases. The exceptions are discussed in Sec. 5.7.3.2 below.

The speedups for the versions of Gauleg shown in Figs. 5.9 and 5.10 are much less dependent on the cache consistency technique used. Shared hardware consistency and non-cacheable data perform the best, but the other techniques are not far behind. The only exception to this trend is for the smaller program sizes, 16 iterations in Fig. 5.9, when using all 16 processors. The overall improvement in speedup when compared to LFK21 is a consequence of the low percentage of memory references in Gauleg. It is also worth noting that, in contrast to the LFK21 codes, the performance for all the consistency techniques

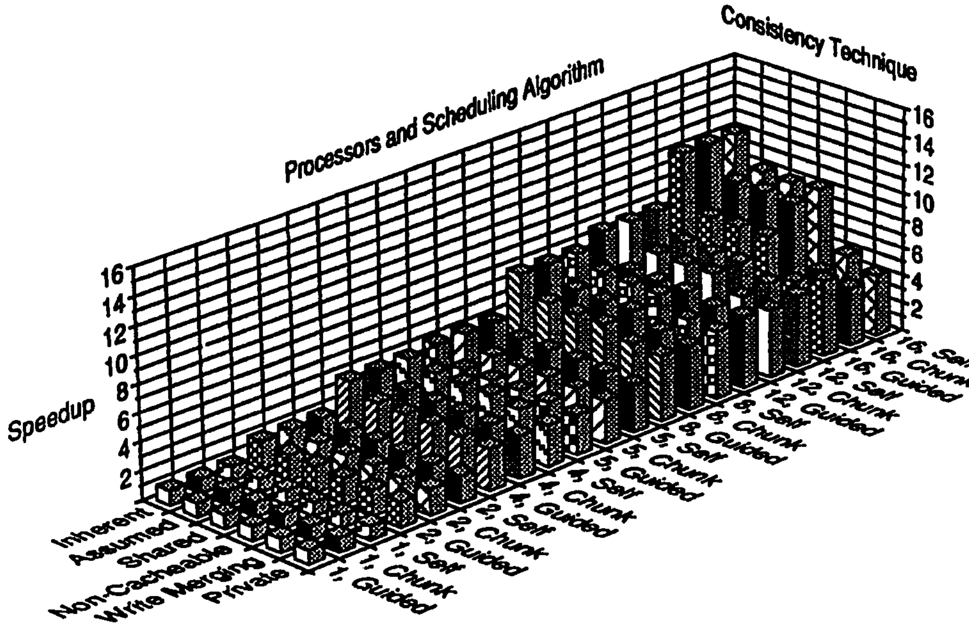

Figure 5.9. Speedups for all versions of Gauleg, 16 iterations.

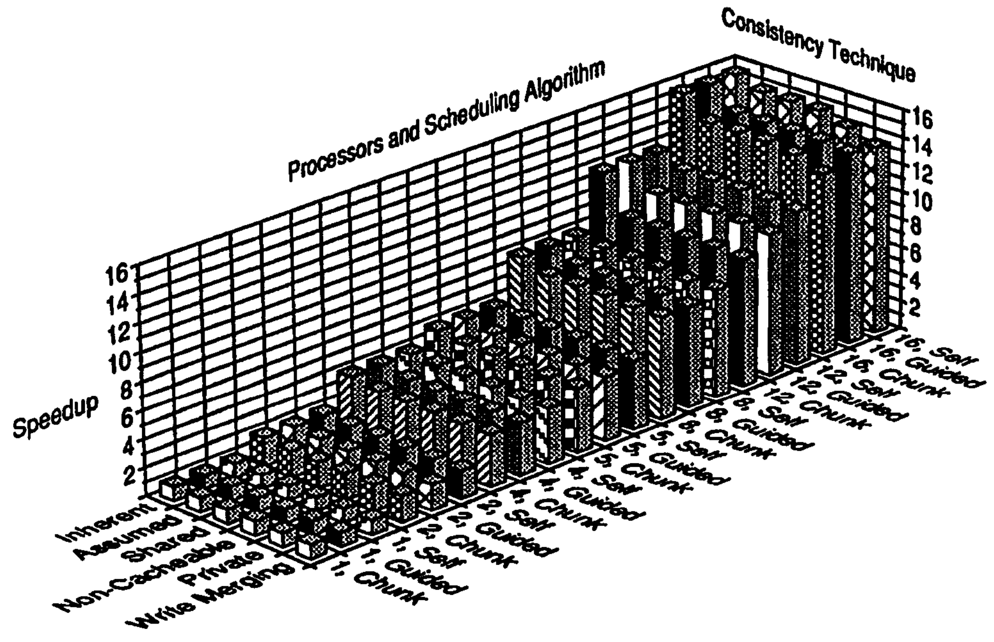

Figure 5.10. Speedups for all versions of Gauleg, 160 iterations.

is very close to that of assumed consistency, which indicates that run-time scheduling and synchronization overhead is greater in this case than cache consistency overhead.

5.7.2 Software Consistency

Perhaps one of the most interesting results was observed when cache consistency was enforced by making result data non-cacheable. Because the first version of LFK21 references memory often, the speedups of the code for n=25, n=50, and n=100 in this case are never greater than 1. The second version of LFK21, however, writes to the result array much less frequently, and the speedups are about 20% less than shared hardware consistency in most cases. In the case of Gauleg, memory references are infrequent, and the performance using non-cacheable data is almost identical to that using shared hardware consistency. These results demonstrate that while non-cacheable pages may be a reasonable technique for maintaining consistency in some cases, it can be totally inappropriate in others. It is also important to note that code should be optimized for fewer memory references if non-cacheable data is to be used. This optimization may require loop restructuring as was done for version 2 of LFK21.

The performance of the codes implementing cache consistency with temporary result areas and cache management instructions was poor overall. For version 1 of LFK21, the speedup never exceeded 2, and for version 2, it never exceeded 2.4. The "adjusted" results (described above) for both versions of LFK21 were slightly better. The speedup calculated versus the sequential running time of version 1 of LFK21 was never greater than 2.5, while the speedup calculated versus the sequential running time of version 2 of LFK21 was never greater than 3.5. The results did show an increase in speedup as more processors were employed, whereas the original code that updated the final result area one processor at a time showed its best performance at 2.4 with only 4 processors.

The performance of Gauleg using write merging was much better, approaching hardware consistency for larger iteration counts running on fewer processors. However, this consistency technique was still the worst performer for all tests using Gauleg. This result suggests that non-cacheable data is the best approach to software consistency when memory accesses are infrequent.

5.7.3 Hardware Consistency

For the hardware consistency tests, the simulator was modified as described in Chap. 4. Since cache consistency was enforced in hardware, there was no need for the programs to use any special cache management or data alignment instructions. The programs used for these tests were identical to those used in the assumed consistency tests.

5.7.3.1 Private

For both versions of LFK21 and each value of n, the performance of the private hardware scheme for cache consistency was similar to that of adjusted software consistency. For the first version of LFK21, the performance of hardware consistency was worse for fewer numbers of processors, and better for greater numbers of processors. For the second version of LFK21, the private hardware consistency technique always performed worse than the software results, with maximum speedups being less than 2.7 and 2.0 when compared to both versions of the sequential code.

For the tests involving Gauleg, the performance of private hardware consistency was close to that of cached write buffer software consistency. Although they were slowest, these two techniques did perform quite well when compared to the others, except when 8 or more processors were used on 16 iterations or less (see Fig. 5.9). This result is a direct reflection of the percentage of memory accesses made by the different problem sizes of Gauleg.

It is interesting to note here the difference in performance of the various scheduling algorithms. In the other tests where consistency is assumed or main memory traffic is otherwise reduced, self-scheduling has a slight performance edge for larger numbers of processors because of its improved dynamic load balancing. This is also true for all experiments involving the Gauleg codes, where variable length iterations present a need for better dynamic load balancing. However, in Fig. 5.7 we see that chunk scheduling and guided self-scheduling perform better in the LFK21 version 1 experiments using private hardware consistency. Because they allocate chunks of iterations greater than 1, multiple adjacent iterations are allocated for each schedule. This provides additional spatial locality on each processor which reduces cache misses. This result is more visible for the first version of LFK21, where memory is accessed frequently. However, this locality effect is not an issue with the Gauleg code, due to the relatively low number of memory references.

The major problem with the private hardware consistency technique is the slowdown caused by the prevention of sharing of read-only data. This is especially evident for the second version of LFK21, where each of the microtasks access the elements of one of the operands in the same order in a non-unit stride fashion. Since an access to any block must invalidate any other copies of that block, there are many more cache misses using this consistency technique.

5.7.3.2 Shared

The performance of the shared hardware consistency scheme was better than any of the other techniques, and approached the performance of assumed consistency for both Gauleg and the second version of LFK21. The shared hardware consistency mechanism allows any cache block to reside in more than one cache. As long as the block is not modified, no consistency actions are required. This approach avoids the difficulties encountered by the private scheme with read-only data. With sharing of read-only data possible, the effects of reference locality were less pronounced with shared hardware consistency. Locality did have a major impact on version 1 of LFK21, where guided self-scheduling and chunk scheduling outperformed self-scheduling (see Fig. 5.7).

Tables 5.3, 5.4, and 5.5 show the running times in microseconds for both versions of LFK21 using shared hardware consistency. These tables show the volatility in performance as the problem size changes. As n increases, the number of microtasks created increases, while the grain size of each microtask remains the same. For n=25, we see that, depending on the scheduling algorithm, version 2 outperforms version 1 for larger numbers of

	Version 1 Times			Version 2 Times		
P	Chunk	Self	Guided	Chunk	Self	Guided
1	2656	2646	2603	3683	3745	3681
2	1420	3105	1387	1941	1977	1934
4	800	1697	909	1068	1080	1062
8	493	1032	628	639	634	627
12	394	744	637	498	490	489
16	740	742	632	377	364	405

Table 5.3. LFK21 run times for n=25, shared hardware consistency.

	Version 1 Times			Version 2 Times		
P	Chunk	Self	Guided	Chunk	Self	Guided
1	5247	5235	5141	7324	7460	7322
2	2660	6193	2638	3686	3780	3716
4	1526	3294	1685	2090	1983	1964
8	910	1968	959	1219	1090	1082
12	705	1353	794	941	793	798
16	606	1120	732	812	653	651

Table 5.4. LFK21 run times for n=50, shared hardware consistency.

processors. When n=50, the performance of the two versions is about equal for 12 and 16 processors. For n=100, version 1 has better performance for all scheduling algorithms using shared hardware consistency except for 16 processors, where the performance of the two versions is about equal. This result demonstrates that as the number of microtasks increases relative to the number of processors and efficiency levels rise, the factors that dominate uniprocessor performance also begin to dominate multiprocessor performance.

5.7.3.3 Cache Line Size

In an attempt to reduce the amount of false sharing and the resultant invalidation misses, we tested the two versions of the LFK21 code with different cache line sizes and set associativity for assumed, shared, and private hardware consistency. While the change in associativity

	Version 1 Times			Version 2 Times		
P	Chunk	Self	Guided	Chunk	Self	Guided
1	10427	10412	10216	14605	14891	14604
2	5250	12358	5182	7327	7529	7364
4	2663	6512	3027	3689	3785	3741
8	1738	3685	1665	2390	1992	1976
12	1324	2641	1313	1814	1400	1403
16	1122	2067	1126	1528	1111	1102

Table 5.5. LFK21 run times for n=100, shared hardware consistency.

had virtually no effect, there were some variations caused by changing the cache line size. For version 1 of LFK21 with n=100, the performance using self-scheduling was improved when line sizes of 8 words (64 bytes) and 4 words (32 bytes) were used. However, this improvement did not match the performance of guided self-scheduling when using a 16 word cache line size. For version 2 of LFK21 with n=100, all scheduling algorithms ran faster when using an 8 word or 4 word cache line size when using private hardware consistency. This increase was not enough to compete with shared hardware consistency with the original 16 word line size. All tests with a line size of 2 words resulted in reduced performance. Overall, the attempts to reduce false sharing and increase performance with a smaller line size were unsuccessful. This is due in part to the unit strides present in these codes and their reliance on spatial locality.

5.7.4 Scheduling Algorithms

Our experiments did not suggest that any of the three scheduling algorithms tested was clearly the best. The results for LFK21 using assumed consistency suggest that self-scheduling and guided self-scheduling perform about the same, with chunk scheduling running slower. Self-scheduling provides the most speedup for 16 processors, while guided self-scheduling runs faster on fewer processors. Since self-scheduling allocates only 1 iteration for each schedule, it provides the best dynamic load balancing. However, guided self-scheduling can come close in load balancing, since smaller chunks are allocated during later schedules. Also, since chunk scheduling and guided self-scheduling allocate more iterations per schedule on average, they have fewer scheduling points at run-time. For this reason, these approaches may incur less scheduling overhead than self-scheduling, even though their scheduling code requires more instructions to compute chunk size and loop bounds.

The results for Gauleg, however, indicate that the superior dynamic load balancing of self-scheduling outweighs the reduction in scheduling point overhead. This result occurs because of the variable length instruction counts in the iterations of Gauleg. When chunk scheduling or guided self-scheduling is used on Gauleg, some microtasks are aggregates of several bigger iterations, while others are aggregates of several smaller iterations. This causes a situation where load balancing is upset because some microtasks complete long before others.

For the LFK21 tests using cache consistency, the performance of the scheduling algorithms depended on the amount of memory accesses in the code. For the first version of LFK21, where the number of memory accesses is 55% of the instructions, the performance of guided self-scheduling was best, with chunk scheduling being next (see Fig. 5.7). As mentioned earlier, this is due to the spatial locality in memory referencing that occurs when adjacent iterations are executed on the same processor. For the second version of LFK21, where memory accesses make up 38% of the instructions, guided self-scheduling and self-scheduling are about equal in providing the most speedup (see Fig. 5.8). Similar results were also observed for `n=25` and `n=50`, except that chunk scheduling performed best for version 1, and self-scheduling was best for version 2.

These results suggest that different scheduling techniques be used depending on the characteristics of the code inside the parallel loop. However, the benefit gained by choosing the appropriate scheduling algorithm may be quite small if the other sources of run-time overhead dominate the execution time. In the following subsection, we examine the impact

	Version 1				Version 2			
P	Slowdown	S.L.I.	S.S.	C.C.	Slowdown	S.L.I.	S.S.	C.C.
1	1.5%	0.0%	58.2%	41.8%	0.8%	0.0%	9.4%	90.6%
2	3.0%	17.8%	60.9%	21.3%	1.6%	12.5%	44.9%	42.7%
4	20.3%	7.8%	16.4%	75.7%	3.2%	18.6%	53.2%	28.2%
8	32.3%	23.9%	15.3%	60.7%	9.1%	59.7%	30.4%	10.0%
12	56.6%	24.4%	12.7%	62.9%	16.2%	63.0%	27.9%	9.1%
16	79.1%	25.2%	10.5%	64.4%	21.6%	69.5%	25.3%	5.2%

Table 5.6. LFK21 overhead, guided self-scheduling, shared hardware consistency.

of each of the three performance barriers in our experiments, and identify the bottleneck in several different configurations.

5.7.5 Overhead Breakdown

Tables 5.6 and 5.7 show the breakdown of impedance from each of the three major performance barriers for the best performing combinations of each parallel loop. The column labeled "slowdown" indicates how much slower the code ran than would be ideally possible with linear speedup. The overhead responsible for this slowdown comes from the three main impedance groups described above in Chap. 4. Each of the next three columns of the table show the percentage of total overhead attributed to static load imbalance (S.L.I.), run-time scheduling and synchronization (S.S.), and cache consistency (C.C.). Because these three columns show the amount of total overhead attributable to each category, their sum for any given number of processors (P) is 100% (± 0.1% in roundoff error).

The figures in Table 5.6 indicate that cache consistency support is the main source of overhead in the parallel processing of LFK21, version 1. The next highest contribution comes from static load imbalance. As stated in Chap. 4, there may be additional overhead due to dynamic load imbalance depending on the performance of the run-time scheduling algorithm. In this case, however, the overhead due to run-time scheduling and synchro-

	16 Iterations				160 Iterations			
P	Slowdown	S.L.I.	S.S.	C.C.	Slowdown	S.L.I.	S.S.	C.C.
1	3.9%	0.0%	99.5%	0.5%	0.4%	0.0%	100.0%	0.0%
2	6.7%	23.0%	71.5%	5.5%	0.5%	1.5%	97.5%	1.1%
4	12.1%	37.0%	54.6%	8.4%	1.4%	21.7%	77.3%	0.9%
5	38.1%	57.7%	39.6%	2.7%	1.3%	55.2%	42.5%	2.3%
8	32.9%	52.9%	37.7%	9.4%	3.3%	28.9%	70.3%	0.8%
12	99.9%	55.5%	40.3%	4.1%	7.1%	14.1%	84.9%	1.0%
16	77.7%	55.4%	42.2%	2.4%	7.3%	39.1%	60.2%	0.7%

Table 5.7. Gauleg overhead, self-scheduling, shared hardware consistency.

nization is relatively small when 4 or more processors are used, indicating that dynamic load imbalance is not significant.

The impedance breakdowns for version 2 of LFK21 are quite a bit different than version 1. As more processors are used, the main source of overhead shifts from cache consistency support toward static load imbalance. However, it is important to note that the total overhead (slowdown as a percentage of ideal performance assuming linear speedup) in this case is much less than for version 1. The reduced number of memory references in version 2 has changed the bottleneck of the system to static load imbalance, which is based on the structure of the code itself.

The impedance breakdowns for Gauleg with problem sizes of 16 and 160 iterations appear in Table 5.7. The most interesting columns in this table are the ones which show the percentage slowdown as compared to the ideal case of linear speedup for 160 iterations. The greatest amount of potential performance that is lost is only 7.5% . Because the amount of overhead is so small, the percentage breakdowns in the other columns are volatile. Run-time scheduling and synchronization is the bottleneck in this case, but the overhead contributed by it is quite small. For 16 iterations, the slowdown is greater, but it is largely a result of static load imbalance. This occurs because there are only 16 microtasks created which are of differing granularities.

5.8 Multiple Loop Experiments

5.8.1 TRFD

As mentioned in Sec. 5.2 above, our tests to measure the performance of multiple parallel loops in a single program is based on the TRFD code from the Perfect Club Benchmark Suite. This program is a "kernel" of the calculation used in a two-electron integral transformation [1]. The code is based on a restructured mathematical equation which requires on the order of n^5 floating point operations where n is the number of basis functions used in the transformation. The code for the TRFD program used in our experiments appears in Appendix C, and its structure is summarized by Fig. 5.11. It has been modified to remove statements associated with timing, computing FLOP rates, and printing the results. However, our uniprocessor tests using the parallelized code did include these other statements, in order to help verify the code's correctness.

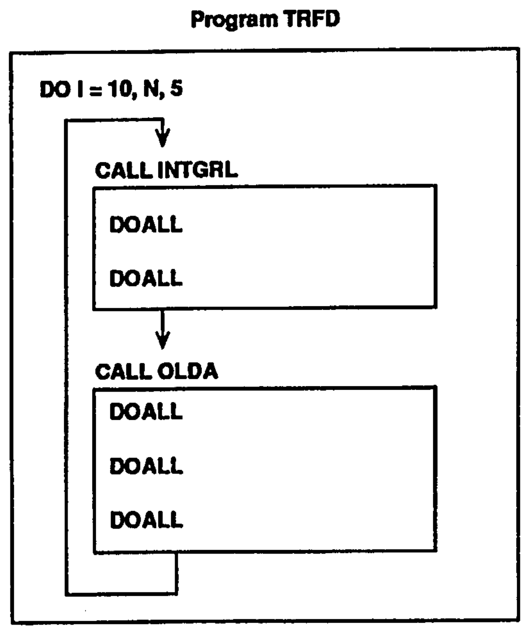

Figure 5.11. Structure of TRFD code.

The TRFD code in Appendix C shows the parallel loops as `doall` loops. There are 5 parallel loops in total, two in the `INTGRL` subroutine and 3 in the `OLDA` subroutine. The mainline of the code is a `DO` loop which repeats the entire calculation (one call each to `INTGRL` and `OLDA`) for different values of n. Although the iterations of this outermost `DO` loop could be done independently, there is little or no advantage to creating a small number of microtasks of very large but very different granularities. Also, this outer loop is provided simply to repeat the test for different values of n as part of the benchmark. It is not part of the transformation calculation.

The microtask graph for this program is similar to the one given in Fig. 5.1, except that there are two more parallel loops before the end of the program. There are also execution dependencies between the end and beginning of each of the last three parallel loops (all contained in the `OLDA` subroutine). As indicated in the graph, the first two parallel loops (both contained in the `INTGRL` subroutine) may have their microtasks execute simultaneously. The bulk of the floating point operations are contained in the third and fifth parallel loops (the first and third in the `OLDA` subroutine). Consequently, the microtasks associated with these loops are of a much larger granularity than those found in our loop kernel tests. These loops also have greatly varying dynamic instruction counts between iterations. The iteration counts are $n * (n + 1)/2$ for the third loop and just n for the fifth.

As mentioned in Sec. 5.2 above, the TRFD program overlays memory space for several arrays in one large common block. In our version, this memory area was expanded so that different iterations of a `doall` loop would not be data dependent. For this program, written for the Perfect Club with n varying between 10 and 40, the increase in memory was from 8M bytes to 16.8M bytes with our version. This increase was not a problem for the large main memory of the ZS-1 (128M bytes). However, in an effort to reduce the simulation time of the program, our tests only considered values of n equal to 10 or 15, which required less than 4M bytes of memory.

5.8.2 Experiments

Because of the size and complexity of the TRFD program, some experiments done with the loop kernels could not be repeated. These include the tests for inherent parallel performance and the write merging approach to cache consistency. The parallel loops in TRFD contain several inner loops, sometimes with a nest level of 5. Combined with the nature of the iteration in each loop, it was too difficult to determine (in a reasonable amount of time) the exact pattern of execution for each iteration of each loop. This prevented us from determining an optimal static load balance. However, we were still able to determine the amount of overhead due to cache consistency, with the exception of the write merging technique. Also, if the previous experiments are any indication, the performance of non-cacheable data in these tests does not suggest that the performance of write merging would perform much better.

5.8.3 Performance

Figures 5.12 and 5.13 summarize the speedups realized with parallel versions of TRFD for $n = 10$ and $n = 15$. The performance of each cache consistency technique is overlayed in the figures, so that the incremental improvement of one cache consistency techniques over another can be seen directly. Recall that, for $n = 15$, the program actually repeats the entire calculation for $n = 10$. However, the time spent executing the loops with $n = 15$ greatly exceeds the time spent for $n = 10$ (about 8:1).

The speedups for TRFD are comparable to LFK21, especially when $n = 15$. Although we cannot be certain, this may suggest that the load balancing situation is similar, even though TRFD does contain several short serial sections. The memory referencing percentage of instructions is similar to version 2 of LFK21 at 37%. However, this similarity did not help the performance of non-cacheable data as a consistency technique. Clearly, TRFD makes more of its memory references to shared writable data. While the performance of

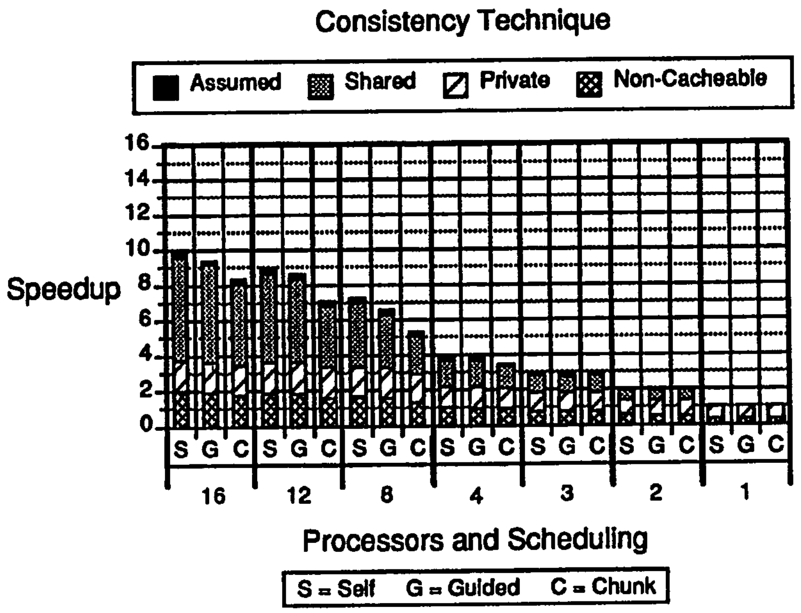

Figure 5.12. TRFD speedups for $n = 10$.

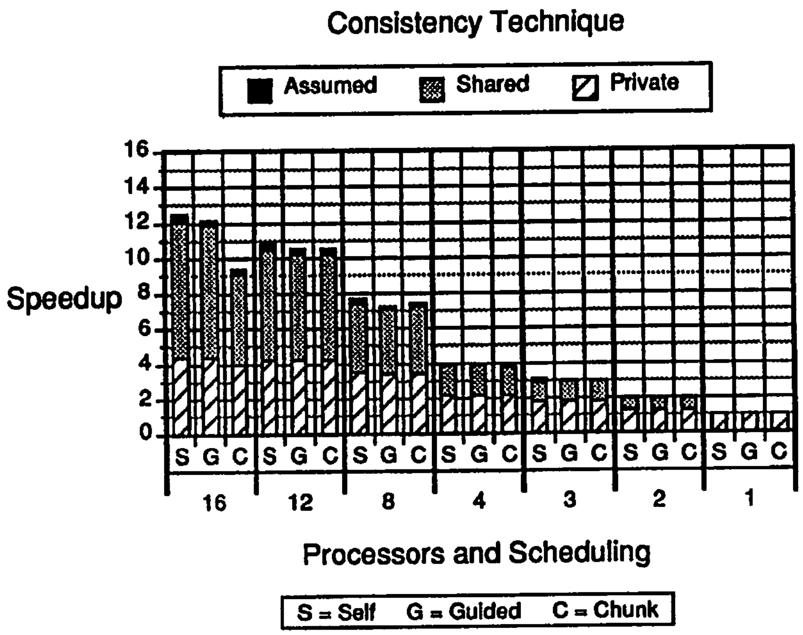

Figure 5.13. TRFD speedups for $n = 15$.

private hardware consistency is similar to LFK21 version 2, the performance of shared hardware consistency is better for TRFD, almost as good as assumed consistency. For this program, read accesses must far outnumber write accesses to shared writable data. Table 5.8 shows the amount of overhead with self-scheduling for cache consistency as a

	TRFD, $n = 10$			TRFD, $n = 15$	
P	Shared	Private	Non-Cacheable	Shared	Private
1	0.00%	0.00%	244.75%	0.09%	0.00%
2	3.54%	46.67%	253.80%	2.90%	57.57%
3	4.18%	60.01%	269.39%	3.26%	67.66%
4	4.34%	74.98%	275.16%	3.28%	80.30%
8	4.55%	118.75%	328.75%	3.39%	118.42%
12	3.98%	147.36%	379.74%	3.34%	154.41%
16	3.78%	170.85%	396.36%	3.10%	184.30%

Table 5.8. Cache consistency overhead for TRFD, self-scheduling.

percentage of running time for assumed consistency. The overhead with chunk and guided self-scheduling is slightly lower, but these codes also produced less speedup. This is discussed further below.

In examining the speedups for TRFD, we see nearly linear speedup until more than 8 processors are employed. For $n = 15$, 12 and 16 processor configurations perform significantly better than the case where $n = 10$. This is due to the increased iteration counts of the compute-intensive parallel loops. Except for the tests with 1 or 2 processors, the results also show the best performance for self-scheduling followed by guided self-scheduling and chunk scheduling respectively. This result is due to the load balancing benefits of self-scheduling in the presence of large and variable grain microtasks. Indeed, the data cache hit rates for these tests show a slightly higher hit rate for chunk scheduling, even though it produced the smallest amount of speedup. Table 5.9 shows the overall hit rates for each combination of scheduling and cache consistency techniques for 16 processors.

5.9 Conclusions

While our investigation studied several aspects of the performance of parallel loops on a multiprocessor mini-supercomputer, it should not be assumed that the results observed in

	TRFD, $n = 10$			TRFD, $n = 15$		
Scheduling	Assumed	Shared	Private	Assumed	Shared	Private
Self	99.4%	98.6%	72.8%	99.6%	99.1%	72.7%
Guided	99.4%	98.8%	73.3%	99.7%	99.3%	73.8%
Chunk	99.5%	98.8%	74.0%	99.7%	99.2%	73.0%

Table 5.9. Data cache hit rates for TRFD, 16 processors.

our tests will be identical to those obtained on other multiprocessors or with different parallel programs. We can, however, make the following conclusions based on our experiments.

5.9.1 Load Balancing

Static load imbalance is a major source of impedance to speedup. Although cache consistency support is often the bottleneck in a parallel processing system, load balancing is a major factor that may be more easily addressed. While our experiments demonstrated that high efficiencies are possible with `doall` loops, if a source program has a significant fraction of sequential code between these loops, overall program efficiency will be quite low. By modifying the run-time system to permit nested parallel constructs to be supported via a variable length queue, more parallelism of varying grains may be exploited. Although this may require more overhead in run-time support code, the tradeoff may well be worth it, since this overhead is quite low with the current technique. This approach also requires a modification of the language model of parallelism. Care must be taken, however, to add parallel programming constructs that do not severely hamper the performance of the run-time system. This is discussed further in the following chapter.

5.9.2 Cache Consistency

Hardware consistency techniques perform better than software techniques. However, better software techniques are possible, but they require additional hardware support and/or operating system support. For this reason, software approaches seem better suited to statically-scheduled single-user parallel processing systems. On the other hand, only the private hardware consistency technique is scalable in the number of processors. Because we observed poor performance with this technique, other scalable techniques based on shared hardware consistency should be considered if more than a few dozen processors are to be used. One such possibility is a broadcast-based approach, which does not require a presence bit for each cache [4], or a partial map scheme where only some presence bits are kept in the central directory and the others are distributed throughout memory [78, 129] or supported in software [35]. Additionally, depending on its hardware complexity, a policy based on write-broadcast instead of write-invalidate for maintaining consistent data may be more appropriate for the fine-grained sharing present in parallel loops.

Our results show shared hardware consistency to be the best technique for maintaining cache consistency. It outperformed the other techniques we tested, including private hardware consistency. This is attributable to the type of parallelism present in the test programs. A fine-grained parallel program is more likely to share data than those parallelized at the program level. Also, our tests accounted for the effects of sharing cache blocks even when individual words are not shared (i.e., false sharing). This level of sharing caused too many invalidations in the private hardware consistency case.

The software schemes we tested did not fare well at all, with the exception of non-cacheable data for Gauleg and possibly the second version of LFK21. The overhead of write merging for consistency seems to be too great for microtasking. Although this second approach had reasonable performance for some versions of Gauleg, it was still slower than non-cacheable data. Write merging might perform better if larger grain microtasks are present, but, if the result area is also larger, the overhead of merging these areas is also larger. As for non-cacheable data, the performance is directly dependent on the number of references to this data. In some cases, the effect can be extremely bad. If this technique were to be used in a production machine, hardware techniques to speed up these accesses without blocking the cache should be employed.

There are also problems and tradeoffs to consider when using software schemes for consistency. While the techniques we tested did not perform well in many cases, it is possible that techniques using cache management instructions embedded within the parallel loop bodies may perform better. However, these techniques require sophisticated compilers, as well as protection from interrupts or context switching by the operating system. The compiler has to manage the mapping between memory words and cache blocks, to ensure that false sharing does not introduce inconsistency. If the software also assumes the presence of, or absence of, cache blocks based on prefetching or cache management instructions, the operating system must prevent context switching and microtask migration from violating these assumptions. For these reasons, we believe that software consistency schemes are best suited for statically-scheduled single-user compute-intensive parallel processing systems.

In general, parallel programs should be optimized for a minimum number of memory references. As the speedups for Gauleg show, the fewer the memory references, the better the speedup. On the other hand, version 1 of LFK21 almost always provided more MFLOPS with less speedup than version 2 (when compared to itself) in the tests reported here. However, we have observed the opposite result in tests with different iteration counts (see Table 5.3) or when using non-cacheable data for consistency. A tradeoff must be made by the compiler depending on the frequency of memory accesses, any spatial locality in these references, the cache consistency technique employed, and the scheduling algorithm used.

5.9.3 Run-Time Scheduling and Synchronization

Our user-mode run-time system is very efficient. The single processor results demonstrated that run-time support overhead for parallelism was not expensive (see Tables 5.6 and 5.7 for $P = 1$). Although run-time scheduling and synchronization was the bottleneck in some cases, these instances occurred only when the total system overhead was quite low. This is due in large part to the fast semaphore registers and their synchronization primitives. We believe this performance warrants the addition of techniques to support more types of parallelism.

The best scheduling algorithm is dependent on the characteristics of the code. Codes with few memory references and/or variable sized loop iterations should use self-scheduling. Codes with fixed sized iterations and many memory accesses should use chunk or guided self-scheduling. Consideration must also be given to the type of cache consistency used as well as the performance bottleneck of the system. However, while it is possible that the system bottleneck will prevent a change in performance due to the scheduling algorithm, it may be impossible to determine the bottleneck at compile time.

We should also note though, that if one type of scheduling algorithm is to be applied blindly throughout the program, it should be self-scheduling. As our tests with the TRFD program pointed out, with many large microtasks of varying granularities, the benefits of load balancing with self-scheduling will most likely outweigh the minimal benefit of the increase in cache hit rates obtained with chunk or guided self-scheduling.

5.9.4 Analysis

Now that we have quantified the impact of the three major performance barriers on parallel loop program performance, we can begin to look for alternative programming models that overcome the identified performance problems. Table 5.10 specifies the three performance barriers, a desired change in the properties of the programming model to minimize their impact, and a method for achieving this change in properties. These three columns are labeled "Barrier", "Change", and "Method" respectively.

In order to overcome performance loss due to cache consistency overhead, references to data used only locally must be increased as a percentage of all data references. A straightforward way to accomplish this is to increase the grain size of parallelism, which

Barrier	Change	Method
Cache Consistency	Increase locality	Increase grain size
Static Load Imbalance	Increase parallelism	Nested parallelism
Scheduling and Synchronization	Reduce scheduling	Increase grain size

Table 5.10. Methods to overcome impact of performance barriers.

provides for longer instruction streams which compute a larger portion of work on a local data set with fewer accesses to shared data.

The problems associated with static load imbalance can be overcome by increasing the amount of parallelism present in the program. This can be done by allowing nesting in parallel constructs, so that multiple, potentially heterogeneous, execution streams may execute simultaneously. This change need not also reduce the grain size, since there are opportunities for parallelism at a higher level than the inner loops we have concentrated on thus far. This will be demonstrated in Chap. 6.

The amount of overhead due to scheduling and synchronization can also be limited, by reducing the number of scheduling operations. However, we do not want to achieve this goal by constructing microtasks of multiple iterations, since self-scheduling has demonstrated better overall performance with its superior dynamic load balancing. Instead, the number of scheduling operations that occur per instruction can be reduced by simply increasing the average grain size. A new model of parallelism that increases parallelism and grain size is discussed in Chap. 6.

In addition quantifying the major performance barriers and outlining an approach to minimizing them, we can also summarize our major overall conclusions:

- Hardware cache consistency techniques perform better than software techniques.

- Cache consistency and static load imbalance are the major sources of overhead.

- Self-scheduling is the best technique for dispatching microtasks.

While the first two conclusions are straightforward, the last one warrants further discussion. If we assume larger average grain sizes and increased parallelism as suggested above, then self-scheduling will provide the best load balancing and the problem of data locality will be minimized. However, if small-grain microtasks are to be scheduled, a modified self-scheduling technique is possible, where one loop iteration is scheduled at a time, but its value is dependent on the identification of the scheduling processor, as opposed to simple first-come, first-served linear order. Such a modification can ensure that self-scheduling will always provide load balancing good enough to overcome any small advantage the other techniques may have in preserving data reference locality.

5.10 Summary

In this chapter we have evaluated the performance of different scheduling policies and cache consistency techniques in conjunction with loop-based parallel programs. We have also quantified the impact of the three sources of overhead that impede parallel program performance which were introduced in Chap. 4. The discussion above also outlined an approach for minimizing the impact of the three major performance barriers. In the following chapter we will introduce a programming model which implements this approach, by increasing both the potential for parallelism and the grain size. This approach uses a language model that permits procedural parallelism. In the next chapter, after developing a suitable model of parallelism, an enhancement to our run-time support is discussed, and the performance of the resulting system is presented.

CHAPTER 6

PROCEDURE PARALLELISM

6.1 Introduction

In the last chapter, we quantified the impact of the three major barriers which impede the performance of parallel programs using a generally accepted language model for parallel loops. In this chapter we overcome the performance problems discussed in Chap. 5 with procedural parallelism. In the case of procedures, there are many more options and proposals for a language model, as mentioned in Chap. 3. In order to build on our run-time system design discussed in the last chapter, and to meet several design criteria (discussed below), we propose a new language model for procedure parallelism. Following this proposal, we describe the run-time support implementation as well as some performance results from our multiprocessor simulation testbed.

6.1.1 Design Goals

The procedure-based approach outlined in this chapter is intended to offer a middle ground between parallelizing compilers and concurrent languages, while eliminating any ad hoc techniques for run-time support from the language. A major goal in our design, however, is an efficient implementation. An important consequence of this goal is that the language constructs for parallelism must be designed with an eye toward their run-time support requirements. It is critical that parallel language constructs requiring excessive run-time system overhead be omitted from our language proposal. Furthermore, we require the run-time support for these new constructs to be efficient enough to support loop-level parallelism similar to that described in Chap. 5. This requirement enables us to benefit from an existing and generally accepted language model of parallelism while incorporating a new model capable of greater generality. Finally, our language design must also provide nested parallel constructs and a larger grain size in order to overcome the major performance barriers as discussed in Chap. 5.

In addition to overcoming the performance barrier of cache consistency, increasing grain size also reduces the impact of scheduling overhead. This overhead/granularity tradeoff is analyzed by compilers that parallelize at the loop level. Because fine grain loop iterations must be combined in some cases to compensate for overhead, other sources of parallelism within a program are being investigated, so that more processors can be effectively used to increase speedup. Research involving interprocedural data dependency analysis in sequential programs demonstrates a desire to extract more parallelism of a larger granularity [90]. Our approach to parallel programming also addresses this need for larger grain parallelism.

With these points in mind, we can state our goals in developing a parallel language model and run-time system. Our approach focuses on providing language level mechanisms for expressing parallelism. We will use the following words to denote the design goals regarding our parallel programming system:

1. **Efficiency.** The overhead required to support parallel execution threads should be as low as possible. As a result, only a minor penalty should be incurred when the parallel program is executed on a single processor.

2. **Flexibility.** The parallel language constructs should allow expression of varying degrees of granularity from loop-level to procedural-level, which reduces the percentage of execution time spent on overhead by optimizing for specific cases.

3. **Expressiveness.** The language model should allow the straightforward expression of a wide variety of parallel algorithms. An important aspect of this requirement is that a *dynamic* model of parallelism must be supported (this is explained below).

4. **Structure.** In addition to flexibility and expressiveness, the parallel language model should promote the development of well-structured programs. This goal requires that critical sections are centralized and well-identified, so that deadlock among multiple parallel threads is prevented.

By attaining these goals, we will have a parallel programming system that is efficient, easy to use, and widely applicable. The efficiency goal ensures that the system will be useful in both parallel and multistream execution modes. High performance of the run-time system will reduce the minimum grain size of parallelism necessary so that system overhead does not dominate execution time. The flexibility goal will allow the programmer to specify parallel code that meets the practical minimum grain size requirement, so that multiple processors may be profitably exploited. Based on our approach to the efficiency goal, we expect the practical minimum grain size to be small, e.g., on the order of 10,000 executed assembly language instructions (as demonstrated in Sec. 6.4 below), so that large amounts of parallelism may be exploited at run-time.

The expressiveness goal ensures that a wide variety of applications may be coded in the parallel language. The parallel model is intended to support general purpose parallel programming, which includes scientific computation-intensive applications as well as non-numeric applications typified by sorting, searching, and branch and bound applications. The model is not intended to provide mechanisms that are useful for programming embedded or distributed systems, or even simulating such systems. It is also not intended for managing physical resources, and so is not useful for programming operating systems. These types of applications are better suited to *concurrent* languages, e.g., Ada, Occam, Concurrent C, Linda, etc., which allow persistent communicating processes at the expense of run-time system overhead, i.e., multitasking as described in Chap. 2.

On the other hand, the model for parallelism we propose is intended to be more expressive than parallel loops or sections that are based on sequential languages, especially those constructs that may not be nested. To attain this goal, we incorporate a dynamic model of parallelism. Even though parallelizing compilers may use dynamic scheduling techniques to allocate loop iterates to processors (as described in Chap. 5), the structure of the program parallelism is typically static, where the main program creates a number of microtasks to execute the iterates, and then resumes control to set up the next parallel loop. In our dynamic model of parallelism, any thread of execution can dynamically create additional threads using an explicit **create** statement. This requires a run-time system capability to queue and schedule multiple heterogeneous threads or *procedure-processes*. Using this model, the bottleneck of the main program executing serially to set up each new homogeneous parallel section is eliminated. Instead, new instances of parallel code are set up and added to the run queue when they are created, and in parallel with other procedure-processes.

The structure goal is intended to encourage an easy to understand, straightforward style of parallel programming, that does not hamper efficiency, flexibility, or expressiveness. If implemented, the benefit of this requirement is that the model used for expressing parallelism prevents deadlock and "distributed critical sections". Deadlock results when a parallel program unit is suspended waiting for an event that will never occur, e.g., the unlocking of a semaphore. Previous proposals for imperative parallel languages have provided constructs which enable deadlock situations to be easily specified by a programmer. One example is a general purpose semaphore or synchronization statement, e.g., a barrier, that can be used anywhere in a parallel program. In this case, a barrier could be specified where each processor must wait for all others to reach the barrier, but one processor may have branched over the barrier. In other languages where barriers are not provided, it is still possible to encounter deadlock through improper use of primitives for synchronous communication. In any case, it is a difficult task for a programmer to manage all the possibilities of parallel execution and ensure that deadlock will not occur in his or her program.

Distributed critical sections occur when different programming units manipulate shared resources, each within its own code section. Even though these accesses to shared resources may occur in mutual exclusion, the program is still difficult to understand and debug. These problems with semaphores were identified in [74] and [27], where proposals were made to unify synchronization and communication to aid in centralizing a given critical section. This idea was refined with the *extended rendezvous* concept of Ada, where a critical section can be embedded within a rendezvous statement which is used for synchronization and communication between tasks [77]. We propose a parallel programming model that centralizes access to data that is logically shared by multiple program units as well, but our constructs rely on parameter passing with process creation and termination instead of rendezvous. Our model also prevents deadlock in the scheduling and execution of parallel program units.

6.2 Language Model

6.2.1 Overview and Background

Our approach to the development of a parallel programming system begins with the design of a suitable language and its run-time system. In order to focus on the parallel nature of the language, we extend an existing sequential language, namely C. We chose C since it has simple constructs and is easily implemented. Also, from our point of view, C has a desirable property in that functions cannot be nested statically. This is consistent with our model of parallel program units, that are also restricted from being statically nested. Our proposed language constructs can also be added to FORTRAN with only slight modification, as we have already done in order to perform several experiments.

The extensions we propose for parallel programming include the addition of a parallel procedure model to C. Each of these parallel procedures, called *procedure-processes* or, to adopt the terminology of [97], simply *paraprocs*, share parameters with the process that invoked it. Both process synchronization and communication are handled through the **create** and **merge** primitives. Procedure-processes may create other paraprocs, so that a logical tree of procedure-processes can be created that fluctuates dynamically.

This parallel procedure model bears some resemblance to others that have been proposed previously, as introduced in Chap. 3. The system we propose here is intended to be simpler and therefore more efficiently implemented than any of these other proposals. While several ideas from the other systems have been incorporated, much of the complexity has been stripped away. The mechanisms we propose allow procedure-processes to be created in groups, and restrict synchronization to be performed through process creation and termination only. Our model also allows for parallelism to be specified without the possibility of deadlock.

The run-time system for this parallel language is designed to provide maximum performance. In fact, the design of the parallel model of the language has been influenced considerably by run-time system performance considerations. The origin of our run-time system is the self-scheduling technique described in the previous chapter. By building on this basic run-time system, we can support a dynamic model of parallelism where procedure-processes are *created* in addition to being scheduled at run-time. Because parallel work can be created by any active process, the simple self-scheduling technique must be extended to operate with a variable length run queue of available work. Also, due to the semantics of process creation and termination, it is possible for procedure processes to become suspended while waiting for child processes to complete. This requires a blocking and resumption capability in the run-time system, in order to support the run-until-block semantics.

6.2.2 Parallel Procedure Model

The model we propose for specifying program parallelism is based on adding a process model to the C language. This process model has semantics very similar to that of the C function or FORTRAN subroutine. These procedure processes are quite different from the processes or tasks of concurrent languages in that there are no communication ports, channels, or rendezvous type communication calls. Instead, all explicit communication is through parameters passed to the procedure-process at creation. Parameters are passed to the created paraprocs using reference semantics, so any manipulation of their value is evident to the parent paraproc. There is also a shared global memory space, but it is not protected from simultaneous access by multiple paraprocs. There are no explicit synchronization primitives provided in the language model except for the **create** and **merge** primitives for process creation and termination. Furthermore, there is no data dependency checking between procedure-processes. The programmer ensures that this is not necessary by explicitly stating parallelism with procedure-processes. The rationale for our decisions regarding these language constructs is given in Sec. 6.3, where the run-time system implementation is described.

6.2.2.1 Declaration and Scope

The advantages we see in choosing C as a base language for adding parallel constructs are its rules for function and variable declaration, scope, and visibility. Function declarations cannot be nested in C, and we choose to apply this restriction to procedure-processes as well. Paraprocs are declared in a manner almost identical to C functions (see Sec. 6.2.2.3 below for an example). Using semantics similar to C keeps the model both simple and consistent with the C language.

Scope and visibility rules for variables are the same as they are for C [83]. Global variables are declared at the beginning of the main program or are visible in other files using the `extern` declaration. Variables declared within a procedure-process are not visible outside the procedure-process. Variables may be visible to a subset of processes all declared in a single file using the `static` declaration. Accesses to these variables are unprotected as is the case for global variables. Paraprocs themselves may also be declared as `static`, which restricts their visibility to the file where they are declared and prevents paraprocs declared in other files from creating instances of them.

Paraprocs must be reentrant, so that multiple instantiations of them may all execute simultaneously. All variables local to a paraproc are allocated on the stack, so all of them are deallocated when the process completes, just as in the case of a procedure/function return[1]. Static variables are not allowed in processes, they must instead be declared as described above. Besides its own variables, a paraproc can only access parameters, global variables, and visible static variables. Data that is to be shared by subsets of processes is controlled by the programmer by using parameters or static variables. If the programmer requires that access to shared static or global data must be synchronized, the **create** and **merge** primitives must be used.

6.2.2.2 Creation and Termination

Paraprocs are created in groups of 1 or more in a single **create** statement. The paraproc name and parameter list are specified in the **create** statement, and a variable is specified as the destination of a procedure-process group value that is returned. If a parameter is a scalar or structure, it is passed to each paraproc created. If a parameter is an array, the value passed is indexed from the array base using the created child paraproc's instantiation number. This number is unique to each paraproc in the group, and is a value between 1 and N where N is the number of paraprocs in the group. Each procedure-process has a predefined variable **me**, which is the value of its instantiation number.

After paraprocs are created, they execute in parallel with their creator and siblings. Since any procedure-process may create additional paraprocs, the program can be thought of as a "tree" of paraprocs executing in parallel. A paraproc terminates when it executes

[1] As an optimization, it may be possible to allocate locals in registers. This is discussed in Chap. 7.

a **complete** statement. All paraprocs must **merge** with their child paraprocs before they can execute a **complete** statement. The **merge** statement specifies the procedure-process group value returned by the **create** statement. When all of the child paraprocs in the group terminate, the parent paraproc may continue past the merge point. This is effectively a barrier synchronization, but its scope is limited to the parent paraproc and its children rather than the entire program. A paraproc may create several different procedure-process groups during its lifetime, execute with them in parallel, and merge with them in any order it wishes.

The main thread of execution of the program is the "master" process, and it is the only schedulable execution thread when the program begins. The program terminates when the master thread completes, and this can only happen after it merges with all of its children.

6.2.2.3 Example Syntax

The following program fragments show the proposed syntax for the parallel procedure model. The code is a parallel quicksort program adapted from a sequential version given in [5]. The code assumes two predefined functions, `findpivot` and `partition`, which are used to find the pivot value and partition the global array `A` to be partially sorted around the pivot. These sequential routines are be found in [5], and similar FORTRAN versions are reproduced in Appendix C. Because we require that all C functions be reentrant, `findpivot` and `partition` can both be called by many paraprocs simultaneously. This algorithm computes the correct result, because there is no data dependence between paraprocs. Any paraproc sorting the array is working only with its own subrange. This subrange is only shared with its parent paraproc, and the parent stops accessing the array before it creates any children. This is an example of using the **create** statement to synchronize access to shared data.

This algorithm demonstrates the ability of parallel procedures to specify a grain size that is large in comparison to most parallel loop bodies. The calls to the serial routines `findpivot` and `partition` provide a large number of instructions to be executed within a single parallel procedure. Because of the semantics of our language constructs, these function calls in the body of the paraproc do not inhibit parallelization on procedure-process boundaries.

In the example, the new reserved words needed for the parallel extensions are shown in boldface. The first code fragment demonstrates the syntax used for declaring paraprocs (Fig. 6.1). The keyword **process** denotes that the code is not a C function and instead is to be executed as a parallel thread when created by a corresponding **create** statement. The rest of the code is the same as it would be if the procedure-process was instead a function, except for the **complete** statement, which is used to terminate the paraproc.

The quicksort procedure-process also includes **create** and **merge** statements. The **create** statement is used to create a group of paraprocs using some visible procedure-process declaration. In this case, the visible procedure-process is quicksort itself. The value returned by **create** is a procedure-process group identifier, and is used later as a parameter to the **merge** primitive. The number 2 in brackets in the **create** statement specifies the number of paraprocs of the same type to be created. This value can be replaced by any integer expression. The paraproc name and actual parameters are then specified. In this example, the first created paraproc has access to `iarg[0]` and `jarg[0]`, while the second has access to `iarg[1]` and `jarg[1]`. These array values are assigned from local parameters and variables before the **create** statement. These values correspond to the partitioning of the subrange of the array `A`.

In general, each parameter is passed (with reference semantics) to each created procedure-process, except for any parameters that are arrays with a subscript specified as a $*$. In this case, each paraproc is passed a unique entry of the array based on its unique instantiation identifier. The parameters in the paraproc to be created are declared as $*$'ed arrays also, so that the compiler may generate the proper addressing sequence in instances where it has yet to encounter the actual create statement. The array values are taken sequentially,

```
typedef struct node {       /* type declaration for records to be sorted */
    char     name[NAMESIZE];
    int      key;
}   RECORD;

RECORD A[N];                /* N is some predefined constant, A is array of records */
.  .  .                     /* other global are declared here */
process quicksort(i,j)
int i(*), j(*);                 /* declare type of parameters*/
{
    int pivot, pivotindex, k;       /* local variable declaration */
    iarg[2], jarg[2];           /* variable declaration for child paraproc parameters */
    pid sorters;                /* variable declaration for paraproc group handle */

    pivotindex = findpivot(i,j);        /* source code for findpivot is in App. C */
    if(pivotindex != 0)
        {
        pivot = A[pivotindex].key;
        k = partition(i,j,pivot);       /* source code for partition is in App. C */
        iarg[0] = i; jarg[0] = k - 1;
        iarg[1] = k; jarg[1] = j;
        sorters = create [2] quicksort(iarg[*], jarg[*]);
        /* creates 2 paraprocs, each gets one value of unique index in iarg and jarg */

        merge(sorters);         /* wait for the 2 sorter paraprocs to complete */
        }

    complete;                   /* paraproc terminates */
}
```

Figure 6.1. Paraproc declaration.

starting with the value at subscript 0, since arrays are zero-based in C. If the array is multi-dimensional, with $*$'s in multiple dimensions, the indices are assigned with the right-most index varying first (this is consistent with row-major ordering). These rules are significant if the number of procedure-processes created is less than the total number of values in an array parameter. If the number of paraprocs is greater than the number of values corresponding to $*$'ed dimensions of an array parameter, an erroneous condition occurs. It is possible to generate code to check this condition at run time if it cannot be determined at compile time, e.g., the number of paraprocs to be created is not known at compile time. The issue of checking for erroneous conditions is discussed in Chap. 7.

The next line of quicksort shows the syntax used for merging with child paraprocs. This statement acts as a barrier synchronization by suspending the parent until all of the children referenced by the procedure-process group handle are completed. Since the semantics of the parameter passing are by reference, any values returned by the child paraprocs are available in the actual parameters after the **merge** statement. However, in this example, `iarg` and `jarg` are unaffected. The child paraproc's updates of `A`, though, are recorded at the merge point.

Figure 6.2 can be thought of as a continuation of Fig. 6.1, where, after declaration of the quicksort procedure-process, the main program appears. The main program creates 1 quicksort paraproc to sort the entire array. Because the creator does not suspend after the **create**, other statements may be executed before the **merge**, including more **create** statements for other procedure-processes. For example, several sorts on different keys may all proceed in parallel if the array `A` is used as a read-only variable. This would require some minor modifications to quicksort to make it more general.

```
main()
{
    pid sorter;                        /* declare paraproc handle of predefined type pid */
    .  .  .                            /* code to initialize A goes here */
    sorter = create [1] quicksort(0, N - 1);
    .  .  .                            /* execution continues here after create */
    merge(sorters);                    /* wait for sorter paraproc to complete */
}
```

Figure 6.2. Paraproc creation in main program.

6.2.2.4 Structure

The structure of our parallel language encourages a programming style where procedure-processes are used as computational tasks while the parent paraproc coordinates the data and results. The **merge** primitive provides a barrier type synchronization mechanism. Because this technique is somewhat limited, some algorithms may need to create and merge with procedure-process groups many times in order to synchronize access to shared data. Because of the efficient design of our run-time system (described in Sec. 6.3), we believe this situation is acceptable. Furthermore, as we will show in the next section, our basic parallel extensions prevent a deadlock situation from occurring.

Alternatives to our **create** and **merge** primitives that allow processes to synchronize arbitrarily, e.g., semaphores, allow a coding style which is confusing, contains distributed critical sections, and permits a deadlock situation to occur. Deadlock can also occur if rendezvous style communication is used. Adding rendezvous capabilities also increases process weight by adding communication queues to the process state, which makes run-time system context switching more expensive. However, as stated above, some algorithms are not suitable for our proposed programming model, and they will have to utilize other languages that incorporate alternative synchronization techniques.

6.3 Run-Time Support

The run-time system is the key component in the implementation of a parallel programming system. It provides the interface between the hardware, operating system, and the model of parallelism at the language level. In order to describe the run-time system, we must make some assumptions about the hardware environment. Our proposal is intended for a system of one or more homogeneous processors, each having direct access to a logically global shared memory space. This memory space may be contained in a central "main memory" unit or spread across several memory units. The memory space may be cached into multiple caches or local memories that may share individual data items. In this case, the caches are assumed to be kept consistent. Alternatively, the memory space may be spread across multiple local memories and possibly a central memory, with no two memory units both possessing a single data item. In this case, all processing elements must be able to directly access each local memory, e.g., [113], or the operating system must create the "illusion" of shared memory [24].

The target architecture must also supply a non-interruptible read-modify-write instruction for synchronization. This instruction should be executable by the run-time support in user mode so that overhead is kept to a minimum. The ZS series simulator testbed described in Chap. 4 satisfies our requirements for a generic shared-memory multiprocessor.

6.3.1 Overview

The basic structure of the run-time system is based on microtasking. Each processor allocated for the execution of the program begins by executing the run-time system scheduling kernel which runs in user mode (the role of the operating system is discussed in Chap. 7). The kernel code continuously attempts to obtain work for the processor from a global queue. When a program begins, the main unit begins execution and the run queue is empty. Only when additional procedure-processes are created does work enter the queue. All work present in the run queue is ready for scheduling, there is no need to synchronize between execution of queue entries. When work is obtained from the run queue, it is processed until the run-time system is reentered, either to obtain more work, add work to the queue, or to perform synchronization. When the run-time system is reentered, it is possible that some updating of global run-time system data structures will be performed as a result of the work just completed or the synchronization request. When all work is completed each processor will be busy looping in the kernel attempting to acquire work. When the last thread of execution terminates (the main program unit), an operating system call is made so that those processors may be reclaimed and used for another job.

6.3.2 Scheduling

Our approach for scheduling processors can be described as a self-scheduling style. As stated in Chap. 5, this term has been used to describe the technique where multiple processors each obtain a unique iteration of a parallel loop they are to execute [120, 106]. The case we describe is similar in that each processor acquires an index and other basic information from the queue that determines which instantiation of which parallel procedure it is to execute. The queue structure enhances the analogy, since one queue entry is made for each *group* of procedure-processes created.

The queue is a linked list of work entry data structures, or *frames*. These frames are similar to the frames used in the Spoc run-time system [95]. However, the frames we use are simpler, and are not variable length execution frames for procedures. Instead, they contain a fixed amount of basic information that is needed to begin a procedure-process. A frame consists of the parallel procedure's starting address, the number of members in the procedure-process group, a pointer to a memory space where parameter pointers reside, and a pointer to the next frame in the work queue. The slot that holds the number of instantiations to be executed also doubles as a synchronization counter. Frames are allocated from an area in memory designated to be the frame pool. Because frames are of a fixed size, their allocation and reclamation can be performed very quickly without any interaction with the operating system.

When a processor schedules a paraproc, an indivisible *fetch&decrement*[2] operation is performed on a global register or well-known memory location that contains the number of paraprocs yet to be created for the procedure-process group represented by the frame at the head of the run queue. This global value is initialized by reading the count of procedure-processes to be created (which is also the synchronization counter) from the corresponding frame when it is moved to the head of the run queue. The global value is read before the fetch&decrement operation to assure that it is greater than zero. If it is not, the run queue is empty, and the value is reread in a tight loop until it is greater than zero, indicating that the fetch&decrement can proceed[3]. If the number returned by the decrement is greater

[2] If the hardware does not support fetch&op, the operation is performed non-atomically using the provided synchronization primitive to ensure mutual exclusion.

[3] The processor may also check its stack in the loop to see if it has a blocked paraproc that

than zero, the processor begins execution of the procedure-process indicated by the current frame with the unique instantiation index returned from the fetch&decrement operation (this index is the value of the **me** variable described above). The parameter pointer is used to obtain access to the parameters passed to the procedure-process. The synchronization counter in the frame is decremented when the paraproc executes the **complete** statement.

If the value returned from the fetch&decrement operation is not greater than zero, one of two operations takes place. If the value is negative, the scheduling kernel is reentered to reread the value in a tight loop until it is positive[4]. When the value becomes greater then zero, the fetch&decrement is performed again. If the value returned from the fetch&decrement is equal to zero, the last procedure-process of the current frame has been scheduled and the global run queue pointer to this frame must be updated. The processor that assumes this task must wait until all other processors that scheduled one of the current paraprocs has read the starting address and frame pointer before these run-time system globals can be changed. These values are then updated from the next frame in the queue and the queue head pointer is set to reference this next frame of work. If no new work is available, the processor must wait for a new set of paraprocs to be created by entering the tight loop mentioned above. It may also be possible for the run-time system to call the operating system to relinquish the processor instead of waiting for more work. This can only be done under certain conditions, and is discussed in Chap. 7.

6.3.3 Synchronization

Synchronization among multiple procedure-processes is expressed at the language level using the **create** and **merge** primitives. There are no other synchronization primitives provided, but mutual exclusion is observed by the run-time system when needed to perform

may be resumed. This is discussed further below.

4 In the Astronautics ZS series, the processor may idle until the global register becomes positive. This avoids busy waiting. Also, the processor may check its stack to see if it has a blocked paraproc that may be resumed. This is discussed further below.

its services. Because **create** may specify parameters, communication between parent and child is possible, and the **merge** primitive is used to synchronize access to this data. Communication between sibling paraprocs must be coordinated by the parent, and is done by passing the same parameters to more than one child paraproc. Further possibilities for communication are discussed below, in Sec. 6.3.4.

Synchronization in the run-time system is performed by directly manipulating the synchronization hardware or using the assembly-level synchronization instructions. Mutual exclusion is necessary in the run-time system to protect the integrity of run-time system data structures that are shared by all processors running in the kernel. The synchronization performed in the run-time system allows the high-level synchronization statements of the language to be supported, thus eliminating the need for programmers to use low-level synchronization routines such as semaphores.

The barrier style synchronization of the **merge** statement is supported by decrementing the synchronization counter in a frame on behalf of a completing procedure-process. The merging parent checks this value to see if it is zero. If it is not, that paraproc must block, and a new one is scheduled from the run queue with control being transferred in a manner similar to a procedure call. When that procedure-process completes, the synchronization counter for the blocked parent paraproc is checked, and if it is still not zero, another paraproc is scheduled on that processor. If it is zero, control is returned to the parent paraproc, in a manner similar to a procedure return.

Because of the simple structure of synchronization in the language, it is not possible for: 1) a blocked parent paraproc to be waiting for the completion of a child paraproc; *and* 2) all other processors are trying to schedule more work; *and* 3) no new paraprocs are available to be scheduled. This deadlock condition cannot occur because conditions 2) and 3) together imply that any children created by the parent in condition 1) must have completed, and the parent may resume execution. The only way that child paraprocs may block and in turn block their parent is if they themselves have created more paraprocs[5]. This, however, violates conditions 2) and/or 3) for deadlock. There is no other way that paraprocs can block waiting for other events, since there are no other synchronization primitives.

The only possibility for deadlock within the run-time system is when the frame pool becomes exhausted. If the pool cannot be enlarged dynamically, then the program must abort. However, a simple operating system call should trivially accomplish this task. Any other chance for deadlock to occur comes from the actions of the operating system. This issue is discussed in Chap. 7.

6.3.4 Communication

As stated above, communication between paraprocs is primarily through parameters that are passed at paraproc creation. Data can also be shared through the use of global variables. Programmers use **create** and **merge** to synchronize access to shared data as stated above. Parameters allow parent paraprocs to share different variables with different child paraprocs. For example, the syntax described above demonstrated how to split up the elements of an array over a group of child paraprocs.

Parameters are passed with reference semantics to improve efficiency. Since we assume a shared memory space, reference parameters are more efficient than making unnecessary copies of variables. This approach also relieves the programmer from passing pointers to objects that will be modified, as must be done when parameters are passed by value. Passing pointers with value parameters would also complicate the syntax proposed in Sec. 6.2.2 above.

In addition to reducing the number of variable copies, another aspect of the run-time system that provides for increased efficiency is the fixed size of frames. This is done to make frame allocation and initialization very fast. It requires, though, that a set of parameters

[5] Any created paraproc is ready for execution due to the dynamic nature of the run queue. That is, there is NO work placed in the run queue unless it is ready for execution.

be passed via a single address. This address refers to an area where the addresses of all the parameters reside[6]. In order to implement this approach, there must be a quick way to allocate local memory for a procedure-process.

6.3.5 Stack Management

In order to provide this local memory, we assume that each processor has a chunk of sequential memory locations that it can efficiently manage as a local stack. This stack is used in the classic way for procedure (C function) calls and operating system calls. For parallel procedure-processes, it is used for parameter passing with a pointer to the parameter space placed in the corresponding frame. Also, the stack is used for saving contexts of merging paraprocs and accessing synchronization counters at scheduling points. Thus, the stack is shared concurrently among all paraprocs scheduled on a particular processor that have yet to complete. Only one paraproc at a time is active on any processor, so the sharing of the stack is very similar to the sharing that goes on between the caller and callee of a function.

When a procedure-process begins execution, it allocates space for its local variables using the stack in the usual way. The address in the frame that refers to the parameter space is available to the paraproc, and several strategies are possible for using this address to refer to the parameters. These techniques may involve use of the stack. When a new group of paraprocs is to be formed, the stack is used to allocate a parameter space area and an address referring to this area is placed in the new frame. After paraproc creation, the creator continues execution, and the stack may continue to grow in size. When a paraproc terminates, it returns the stack top pointer to reflect the top of the stack when the paraproc began execution. Figure 6.3 shows a logical organization of stacks and frames in the run queue.

6 Possibilities for optimizing parameter passing are discussed in Chap. 7.

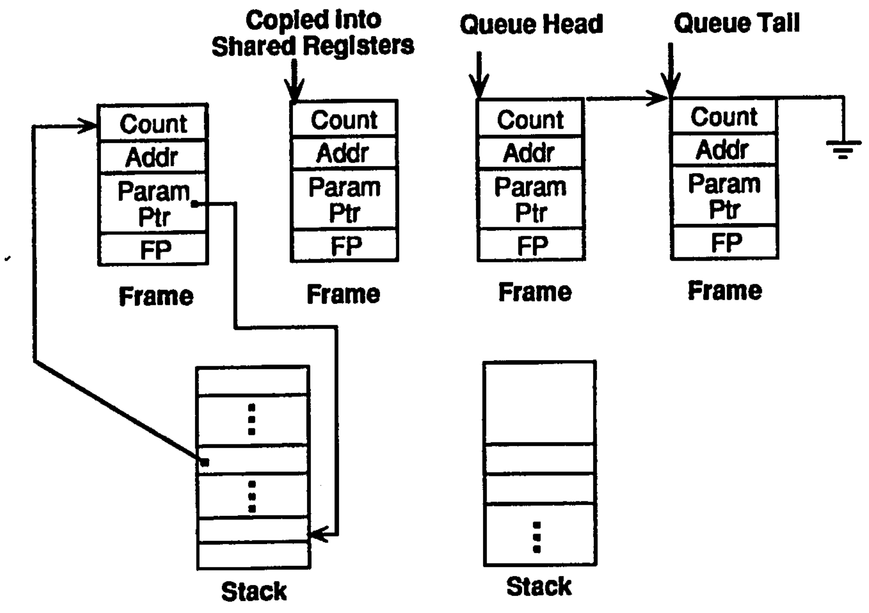

Figure 6.3. Processor stacks and frames in run queue.

If a paraproc executes a **merge** statement and the child paraprocs have not all yet terminated, the parent paraproc must block. Any registers that must be saved by the parent paraproc are pushed on the stack in addition to a return address, as is similar to the case of a procedure call. Also, the last item to be pushed on the stack is the address of the synchronization counter that the blocking paraproc is waiting on. The processor then jumps to the run-time system's self-scheduling code to obtain more work.

This approach relieves paraprocs from saving registers on the stack before they may begin execution. Since a blocking paraproc saves its own context[7] on the stack, it need only save the registers that it will use after the merge point. In the programs we have implemented with this technique, the amount of registers to be saved at a merge point is usually less than 4, as opposed to saving all general purpose registers upon entry. Furthermore, this technique enables a paraproc context to be transferred to another processor (or virtual processor) via a stack pointer. Although this type of migration has not yet been implemented, its usefulness and exact implementation are discussed in Chap. 7.

7 Essentially its stack space and general purpose registers, not special purpose registers or other information managed at the operating system level

When a paraproc completes, it returns the stack to its original location as described above. A check is then made to see if the top of the stack is a valid pointer to a synchronization counter that is now equal to zero. If it does, the counter reference is popped and a procedure-style return is executed so that the blocked parent may resume. If the synchronization value is non-zero, a jump to the run-time system's self-scheduling code is executed instead.

Because the stack is used as a link to parameters passed to child paraprocs, the parent paraproc is required to merge with its children before completing. This is to prevent the stack top from being returned to a point that deallocates the parameter space. We believe this a not a serious restriction, since it is likely for parent paraprocs to require synchronization with their children so that shared data can then be manipulated safely. This restriction also permits the use of fixed length frames, since the stack is used for the variable length portion of the logical execution frame.

While our system for suspending and resuming a merging procedure-process may seem quite unconventional, we believe that this technique saves us considerable execution overhead. An alternative scheme would be to save the entire state of the suspending process in a process control block (PCB), and link all such blocks in a queue of blocked processes. This approach would increase the overhead of scheduling, since a check of the suspended process queue would then be necessary. We believe the procedure call and return technique to be more efficient, as calling and returning sequences for procedure calls are well understood, and can be implemented with only a few instructions. This approach also allows local stack areas to be used for process stacks, eliminating the need to reload stack pointers from and allocate stack areas in PCB's.

6.3.6 Microtask Graph

Figure 6.4 shows a revised version of the microtask graph introduced in Chap. 5. In this version, additional ellipses are present to denote additional run-time system activities. Also, a dashed line is included to denote a fence in a microtask where execution may suspend due to a merge point. The microtask groups, or *tasks* in the terminology of [106], are denoted in the graph as rectangles with rounded corners. These tasks are parallel procedure groups, and a microtask is an allocation of one parallel procedure from the group. Although the main program thread is depicted as a paraproc in the figure, we must point out that this task contains only one microtask. Tasks that do not include a potential block point could, in principle, also be `doall` loops. The prospect of merging these two programming models is discussed in Chap. 7.

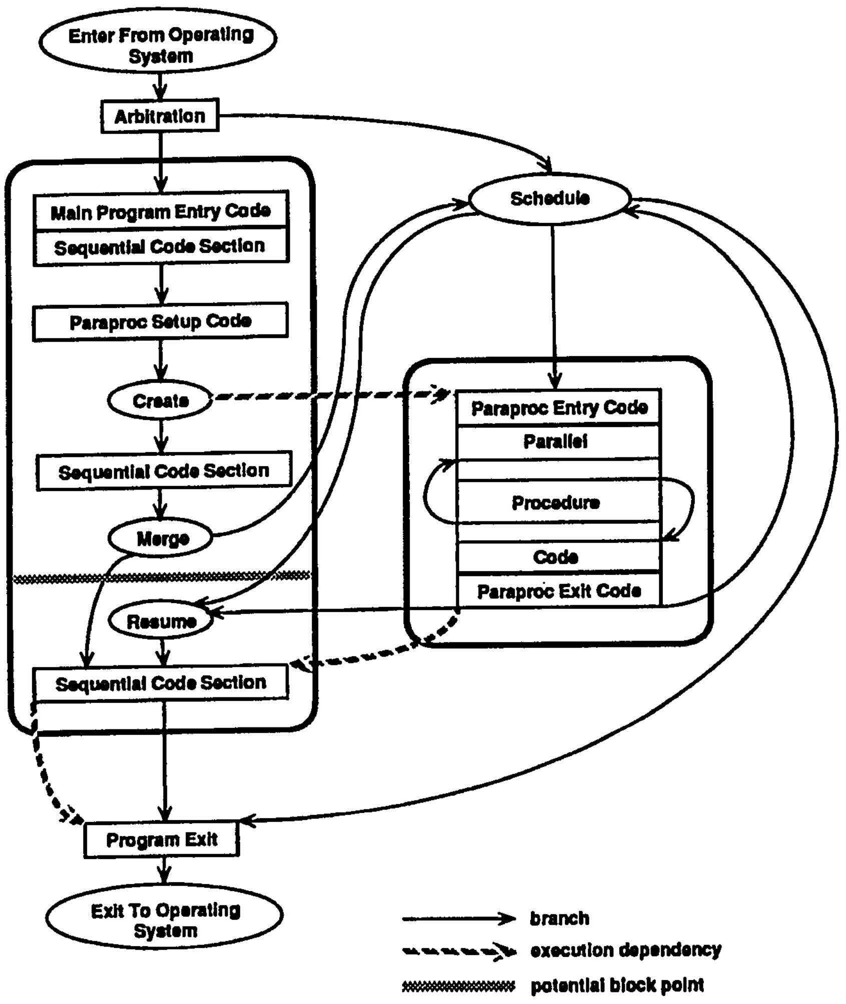

Figure 6.4. Microtask graph for parallel procedure program.

The program depicted in Fig. 6.4 begins execution in the same manner as before. Upon an initial arbitration, one processor begins execution of the main program while the others are dispatched to the scheduling code. The main program sets up the parameters for the first group of paraprocs and then enters the run-time system to "create" them. After creation, the main program resumes execution, and the scheduling kernel is free to start allocating paraprocs to processors. When the mainline reaches the merge point, it must block and branch to the scheduler if its child paraprocs have not yet completed, as indicated by the execution dependency. If the children have completed, the main program microtask may continue execution and then exit to the operating system.

Although not shown in Fig. 6.4, the group of paraprocs shown on the right could themselves create another group of paraprocs. In this case, the creating group would contain a merge point, which would represent a potential block point for every paraproc in the group. Also, since all created paraprocs share the main program microtask as a common ancestor, they each contain a conditional branch to the main program's resume point. The condition is a check of the synchronization counter pointed to at the top of the stack on one of the processors as described above. In fact, with multiple groups of paraprocs created and executed simultaneously in a single program, it is possible for a completing paraproc to branch to a blocked paraproc embedded in its stack, even if the two are unrelated. Finally, we should also point out that the microtask graph represents code fragments and their relationships. It does not indicate the number of instances of any particular group of microtasks, so that multiple levels of instantiations are represented only by one microtask group per level, even though at run-time there may be several groups at each level.

6.3.7 ZS Implementation

The implementation of parallel procedure run-time support for the ZS architecture was straightforward. The frames allocated and initialized as part of paraproc creation were assembled into a linked list as the run queue. The head frame of the run queue was placed in the semaphore registers, so that scheduling could be performed quickly without the need for chasing pointers. Only one processor was needed to update the list and copy the head frame into the semaphore registers once all paraprocs of the current frame had been scheduled. The assembly code for frame allocation and initialization, scheduling and run queue management, and paraproc blocking and resumption can be found in Appendix A.

6.4 Experimental Results

Our experiments with the parallel procedure run-time system described above involved the experimental testbed described in Chap. 4. We tested two programs using the run-time system. The first was a new version of the TRFD program described in Chap. 5. The second was a parallel quicksort program based in part on the code sections used in the language discussion above.

6.4.1 Source Programs

6.4.1.1 TRFD

The TRFD program used in these tests is very similar to the one used in Chap. 5. However, there are two major differences in the parallel procedure version. First, each parallel loop is treated as if it were a parallel procedure. This transformation is one that could be performed by a compiler to make parallel loop code suitable for our parallel procedure run-time system. However, some minor code changes are necessary at the assembly language level in order to use the paraproc run-time system. Instead of passing values through semaphore registers to parallel code loop bodies as done in the code of Chap. 5, these values are passed as parameters using stack space and the frame structure described above. Also, each paraproc body is passed the frame pointer of the parent code, so that read only stack variables may be shared. This last change may be thought of as an enhancement that enables a reduction in parameter space for paraproc calls when values are used in a read-only fashion.

The second major change for the paraproc version of the code is the inclusion of nested parallelism. This approach allowed parallelizing the outermost loop of the program and the first nested loop in the final parallel loop of the OLDA subroutine. As mentioned in Chap. 5, the TRFD program overlays the memory space for several arrays in one large common block. Because the outermost loop of the program repeats the entire calculation for different values of n, the memory space in this common block had to be expanded to avoid unnecessary data dependencies between the calculations associated with different values of n. This required an expansion from 16.8M bytes of the parallel loop version to 36M bytes with the procedure version that can simultaneously accommodate values of n equal to 10, 15, 20, 25, 30 35, and 40. However, in our simulations with $n = 10$ and $n = 15$, only 4M bytes were needed.

The nested parallel loops in the OLDA subroutine did not require any further memory expansion. It did, however, require some pre-calculation of integer values before entrance into loop bodies. In the sequential version of the code, these integer values are accumulated in the bodies of the loops. The pre-calculations of the parallel procedure version were also required for the parallel loop version of the code described in Chap. 5.

The addition of the nested parallel loop in the OLDA subroutine increased the number of microtasks for this last loop from n to $n * (n + 1)/2$. The parallelization of the outermost loop of the entire program did not increase parallelism for the test where $n = 10$. However, when $n = 15$, the program also repeats the calculation for $n = 10$. In this case, the entire calculations involving all parallel loops may execute simultaneously for both $n = 10$ and $n = 15$. The code for this version of TRFD is included in Appendix C.

6.4.1.2 Quicksort

The parallel procedure code for the quicksort program is based on the recursive sequential code found in [5] and used above to demonstrate the syntax and semantics of our language extensions. For our experiments, though, the array of records to be sorted was made up of floating point numbers which were also used as keys. The program was also parallelized at the assembly language level after compiling sequential portions written in FORTRAN. The FORTRAN version of the parallel procedure quicksort program is given in Appendix C.

6.4.2 Performance

6.4.2.1 TRFD

Figures 6.5 and 6.6 show the speedups for the parallel procedure version of TRFD. These graphs should be compared to Figs. 5.12 and 5.13 found in Chap. 5. Recall from Chap. 5 that each bar in these graphs is made up of overlays of bars for each cache consistency technique. That is, for example, the area shaded for assumed consistency shows the additional performance gain over shared hardware consistency. Also, there is only one bar per processor in Figs. 6.5 and 6.6 because only simple self-scheduling is used to dispatch microtasks.

For $n = 10$, we see that, for more than 4 processors, the increased parallelism associated with the paraproc version provides more speedup (20% to 25%) for both assumed and shared hardware consistency. Recall though, that the increased parallelism for $n = 10$ is due solely to the nesting in the last parallel loop in the OLDA subroutine. The enables the creation of 55 microtasks for the inner parallel loop as opposed to 10 outer loop microtasks with the original version. However, despite the increased parallelism and the potentially increased overhead associated with it, the performance of the program on 4 or less processors is still nearly linear even for shared hardware consistency. This suggests that the overhead for the paraproc run-time system is close to that of the parallel loop run-time system.

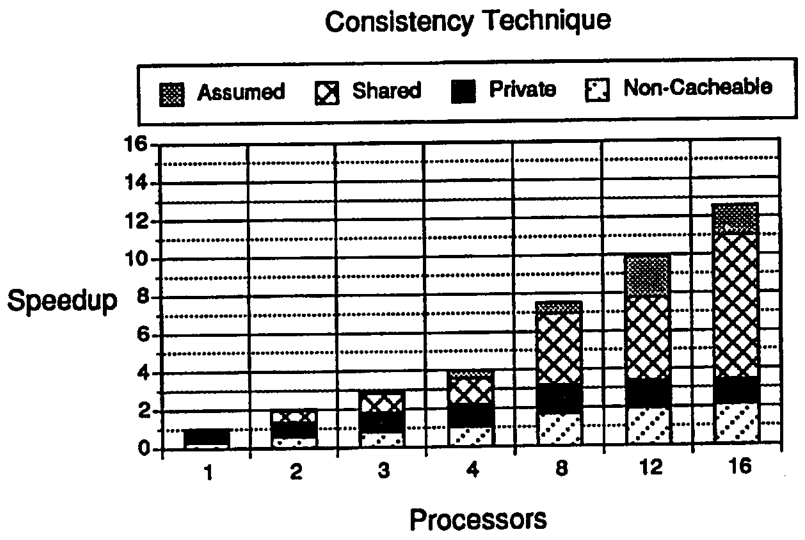

Figure 6.5. TRFD speedups for $n = 10$.

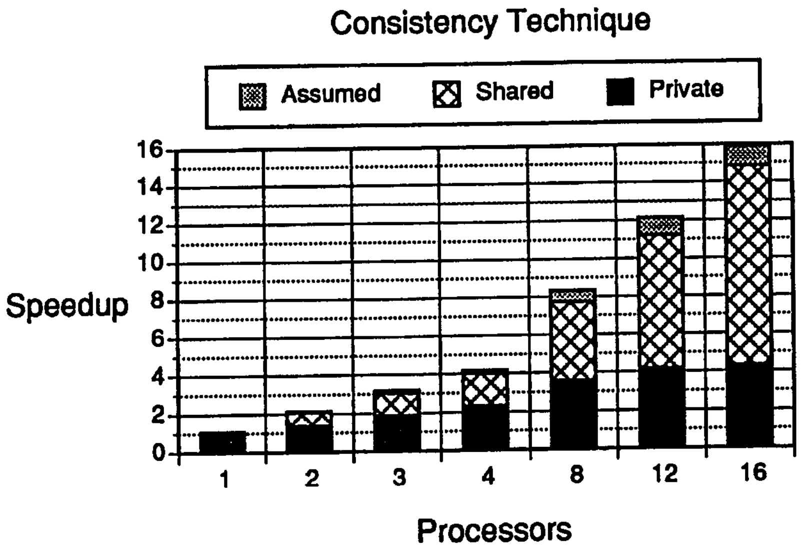

Figure 6.6. TRFD speedups for $n = 15$.

For $n = 15$, the speedups are up to 25% better for both assumed and shared hardware consistency. However, in this case, the speedup is linear for assumed consistency and nearly linear for shared hardware consistency even for 16 processors. In addition to the larger problem size which includes more microtasks of a larger grain, this program also includes more parallelism due to the parallelizing of the outermost loop. Figure 6.7 illustrates the additional parallelism for the paraproc version of TRFD over the loop version for $n = 15$.

As with $n = 10$, the increased parallelism and support from the run-time system does not reduce speedup for smaller numbers of processors which cannot, in general, make use of the additional parallelism.

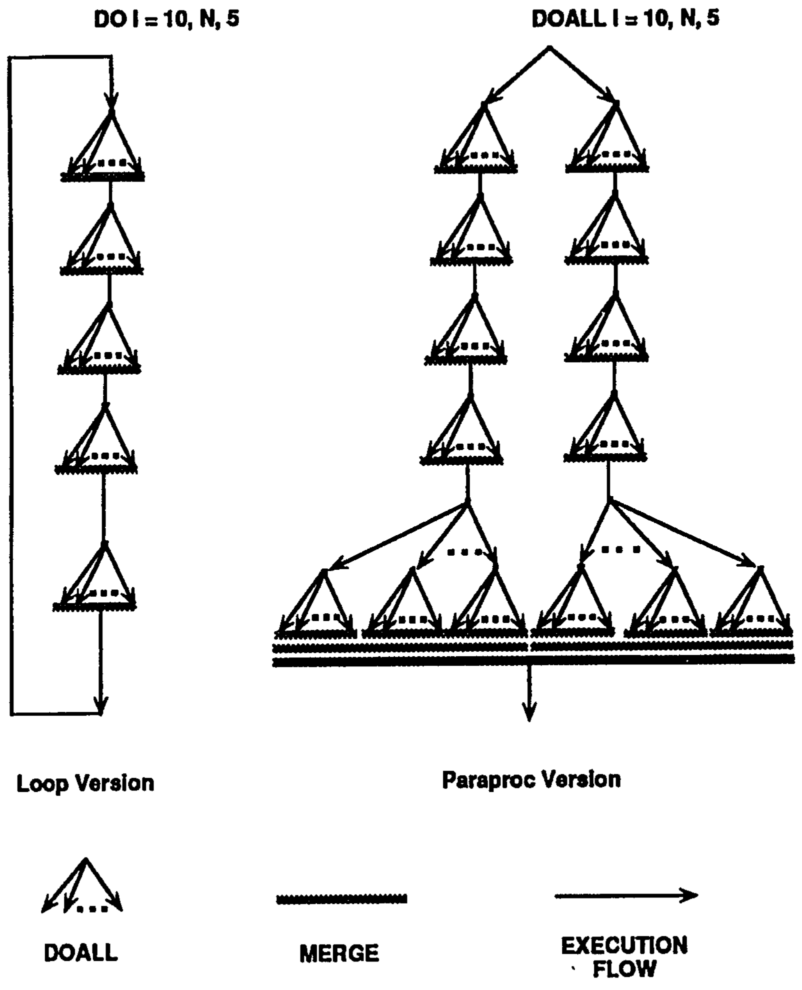

Figure 6.7. Parallelism for TRFD, $n = 15$.

Although the speedups are better overall for the paraproc version of the code, the difference between assumed consistency and shared hardware consistency is greater. The cache consistency overhead is greater in the paraproc case due to its tendency to reduce data reference locality. The speedup for private hardware consistency and non-cacheable data is much less for both loops and paraprocs. This is due to the increased overhead associated with these consistency techniques.

	TRFD, $n = 10$			TRFD, $n = 15$	
P	Shared	Private	Non-Cacheable	Shared	Private
1	0.00%	0.00%	251.95%	0.09%	0.00%
2	4.15%	49.41%	260.88%	2.73%	54.81%
3	4.91%	61.68%	274.27%	4.40%	69.14%
4	10.44%	77.76%	285.86%	4.49%	81.37%
8	7.97%	135.72%	347.75%	7.50%	131.20%
12	27.54%	191.40%	404.00%	8.01%	191.44%
16	14.15%	267.81%	492.76%	6.89%	268.68%

Table 6.1. Cache consistency overhead for paraproc TRFD.

Scheduling	Assumed	Shared	Private
$n = 10$	99.3%	98.2%	74.2%
$n = 15$	99.6%	98.9%	73.8%

Table 6.2. Data cache hit rates for paraproc TRFD, 16 processors.

Tables 6.1 and 6.2 show the cache consistency overhead and the cache hit rates respectively for TRFD using paraprocs. These tables can be compared to Tables 5.8 and 5.9 in Chap. 5. For the paraproc version of the code, the cache consistency overhead is greater. It is also somewhat volatile for the smaller version of the program where $n = 10$. This increase in cache consistency overhead is confirmed by the reduced hit rates for the paraproc version of TRFD. This is further evidence that the increased level of parallelism combined with simple self-scheduling results in a smaller amount of data reference locality per processor. The probability that a processor will schedule consecutive microtasks associated with the same run queue entry is decreased with the paraproc run-time system.

However, in spite of the reduced hit rates and increased cache consistency overhead, the speedups for the paraproc version of TRFD are greater. This clearly shows that the advantage of increased parallelism with its positive effect on load balancing far outweighs the disadvantages of reduced data reference locality. While this may not be true for programs with very small grain microtasks, we believe this example is representative of codes that can benefit from fine-grained parallelization. The microtasks created in the TRFD program are on the order of 10,000 executed assembly language instructions. Other programs with very small grained microtasks, e.g., less than 1000 executed assembly language instructions, have further problems associated with run-time support overhead in addition to load balancing and cache consistency overhead.

6.4.2.2 Quicksort

Figs. 6.8 and 6.9 show the speedups for the parallel quicksort program for 6400 and 50,000 floating point numbers respectively. Similar results were also observed for 1000 numbers, with speedups slightly higher than the 6400 number case. While the speedups for greater numbers of processors is somewhat disappointing for this program, it is important to note that the speedups are at least respectable for 2, 3, and 4 processors.

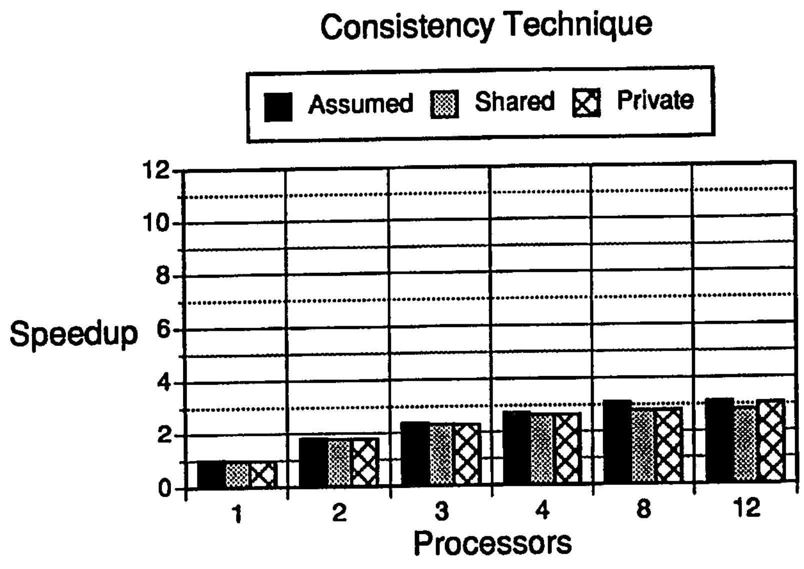

Figure 6.8. Quicksort speedups for n = 6400.

The overall limit to speedup for this program is due to the nature of its parallelization. The `FINDPIVOT` and `PARTITION` routines consume a large percentage of the execution time for each paraproc. Also, the running time for these routines is dependent on the size of

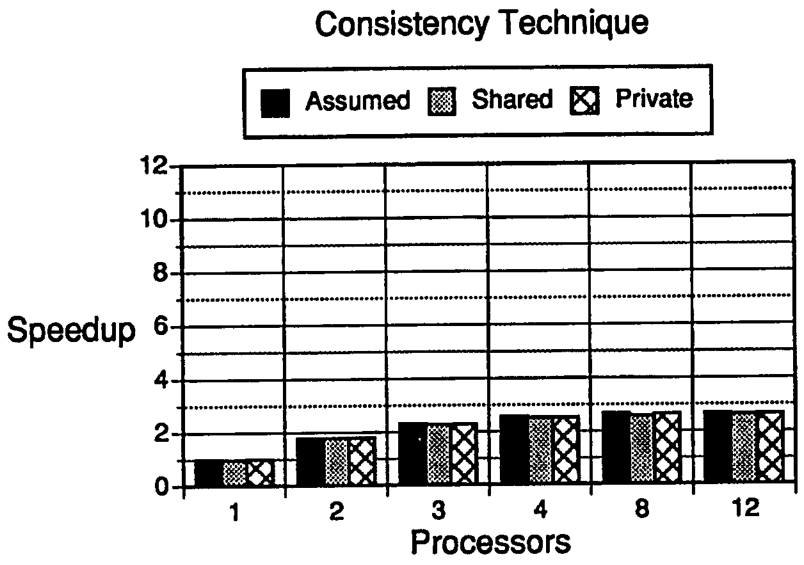

Figure 6.9. Quicksort speedups for $n = 50,000$.

the interval they are working on. At the beginning of the program, before the first recursion, there is only one paraproc executing and the interval to be processed is the entire array A. After the first recursion, there are only 2 paraprocs executing, each with an interval roughly one-half the size of A. Clearly, there is a large portion of execution time spent with less than 4 schedulable paraprocs. Also, as parallelism increases, grain size decreases.

In addition to the reasonable speedups for 2, 3, and 4 processors, there is further evidence that the run-time support code is not the limiting factor in speedup. In an attempt to compare our paraproc quicksort program to "the best sequential algorithm" in order to compute speedup, we tested a sequential FORTRAN version taken from the Numerical Recipes [111]. This program contained loops and GO TO statements together with a stack-like data structure to implement recursion at the FORTRAN source level. This code structure inhibited loop-level parallelization due to data dependencies. Also, in an attempt to avoid a slowdown with quicksort in the case where the data set is already sorted [5], a random number generator was incorporated to aid in partitioning the intervals.

In our tests with this program, it ran between 38% and 76% slower than our paraproc version on 1 processor. This indicates that the "retrofit" of recursion at the source level combined with the random number generator required more overhead than our paraproc run-time system and the procedure calls to FINDPIVOT and PARTITION.

6.5 Summary and Conclusions

In Chap. 5 we identified the need to exploit more parallelism and increase grain size in order to alleviate the problems with load balancing and cache consistency overhead. Our goal in this chapter was to provide language extensions to specify this parallelism in terms of procedures, and to implement efficient run-time support for both these language extensions and parallel loops. We required an efficient implementation of the run-time support so that a solution to the problems with load balancing and cache consistency overhead did not create new problems with run-time scheduling and synchronization.

Our results from experimentation with the paraproc run-time system with the TRFD program suggest that we have met our performance goals. Although cache hit rates decreased slightly, program speedup increased significantly in some cases. Perhaps more importantly, in no instance did performance decrease as a result of the more robust run-time support code. This confirms the efficiency of our run-time support implementation.

We also demonstrated speedup potential for recursive algorithms using our language extensions and run-time system. However, the experiments with the parallel quicksort program also exposed the limits of speedup due to the nature of the parallel algorithm itself. Although enhancements to the run-time system and its operating system interface may overcome this difficulty (discussed below), it is also possible to improve speedup efficiency by taking advantage of the language features to specify more parallelism, including heterogeneous parallelism. For example, the quicksort program could be modified to sort using pointers to records with multiple fields and keys. Then, several sorts on different keys could proceed in parallel. Also, since paraproc creation does not block the parent until a merge point, other paraprocs could be created to perform other operations on the same shared data. For example, a set of student records could be processed to determine a curve for test scores in parallel with sorts based on student names and identification numbers.

It is also worth noting that these paraproc programs specify synchronization implicitly through the language constructs. There is no need for explicit programmer specified synchronization through system calls. This provides the deadlock prevention property of the language extensions, while still allowing a wide variety of general-purpose parallel algorithms to be expressed. This language structure meets the design criteria stated in Sec. 6.1

In all of our tests, there are no operating system calls, I/O statements, or OS-level context switching. As a result, virtual processors are in effect physical processors, and the program's allocation does not vary at run-time. This is not unreasonable for these tests, since all program running times are less than 1 second, a reasonable amount of time observed between interrupts for an OS-level process on a shared-memory multiprocessor[8]. However, in order to make our system feasible in a realistic setting, it must handle OS interaction and be able to vary processor allocation dynamically. For example, the parallel quicksort program is one which could benefit from acquiring additional processors only after several recursions.

The following chapter addresses the issues of merging our run-time system with a multiprocessor operating system. As always, the goal is to provide additional capabilities without unnecessarily increasing overhead.

[8] Although an interrupt may occur as often as once every millisecond to check for schedulable OS-level processes, these interrupts may only be felt by one processor, possibly idle and not allocated for use by a parallel program. Hence, an observed time between interrupts of 1 second is not unreasonable.

CHAPTER 7

SYSTEM ISSUES

7.1 Introduction

In the previous chapters we have addressed problems with parallel language models and their associated run-time support requirements. In this chapter we address the issue of incorporating our run-time system design into a viable system similar to one that could be produced commercially. In order to overcome the restrictions of our simulation testbed and develop a run-time system compatible with existing multiprocessor operating systems, minor modifications to the run-time system described in Chap. 6 are provided. We use this run-time system as a baseline because of its ability to efficiently support both loop- and procedural-level parallelism.

7.2 Language Model

As mentioned above, the run-time system described in Chap. 6 can efficiently support both parallel loops and parallel procedures. This leads us to a language model that contains the constructs introduced in Chap. 6 to support parallel procedures, as well as a `doall` construct to support parallel loops. This enables the programmer to specify parallel loops, but still allows the compiler to detect them. The compiler may also coalesce nested loops to increase grain size and reduce overhead [134]. By adding parallel loop constructs to the language, we can allow this type of parallelism to be expressed naturally, without forcing the programmer to use procedure-processes. Furthermore, by adding a `doacross` looping feature, inter-iteration synchronization may be introduced and handled by the code generator. If we assume that the compiler generates the correct sequence of instructions for synchronization, our argument for deadlock prevention is still valid (provided that the microtasks are scheduled in the order specified by the `doacross`, and that processors do not suspend at these synchronization points). Parallel loop constructs have been proposed and incorporated into several FORTRAN [73, 87] and C [97] dialects. One of those proposals could be adapted to coexist with our language extensions for parallel procedure-processes.

7.3 Code Generation and Optimization

7.3.1 Code Generation

The compiler for a paraproc-based language will require a few modifications in convention and code generation for parallelism. For one, all procedure code must be reentrant, so that multiple sequential procedure calls can be executed by multiple paraprocs simultaneously. Second, the code generated for paraprocs must save and restore registers as part of a **merge** operation, but not as part of the standard procedure entry code. Thus, sequential procedures must save all used registers on entry, but paraprocs do not. The keyword **process** is included in paraproc declaration to aid the compiler in determining which register save policy to employ at different points in the program.

The compiler may also be forced to mix parameter semantics between calls to sequential procedures and creation of paraprocs. Paraprocs use reference semantics not only to relieve the programmer of explicitly passing pointers, but also to facilitate data sharing between parent and child through the use of shared memory. The implementation is free, however, to use value parameters in cases where the value is not modified by the child. This may be desirable in the case where a number in a register is to be passed. A mixing of parameter passing semantics is not uncommon in high-level languages, particularly when sophisticated optimization is involved.

The code generated for **create** statements must also handle the condition of array slices (denoted with an $*$) passed as parameters. This code is rather straightforward to generate, in part because of the syntax, which tells the compiler when it is necessary at both the **create** point and at the entry point of the paraproc. However, this feature requires that the number of paraprocs created be less than or equal to the number of array values. In some cases, the compiler can detect a violation of this condition and issue an error message. In other cases, a run-time test at the point of the **create** statement can detect an erroneous indexing condition. Such a test would certainly increase overhead, but it could be omitted once the debugging phase is over.

Outside of the requirements listed above, the recognition of, and code generation for, each paraproc keyword is rather straightforward. The syntax has been designed to help in this effort. For example, the data required for the **create** statement can be easily parsed and converted into a call to the run-time system routine `aframe`. The **merge** and **complete** statements simply generate standard instruction sequences. The only variable is the number of registers to be saved by the **merge** statement. The **complete** statement is almost unnecessary, but is included to make the compiler's job easier. However, the compiler could detect a missing **complete**, or a case where a **complete** occurs before a necessary **merge**. Assembly code examples for the code generated to implement these statements can be found in Appendix A.

7.3.2 Optimization

Our system for procedure-processes proposed in Chap. 6 also provides several opportunities for optimization. The techniques for parameter passing and local stack variable allocation are prime targets for optimization. Because the run-time system assumes no data dependence between procedure-processes, parameter values may be copied into registers, as long as they are written back to their original locations in memory before the next **create** or **merge** statement (hereafter referred to as a *synchronization point*). Local variable allocation may occur in registers and avoid the stack entirely if they are not to be passed as parameters. On the other hand, stack variables that are to be passed as read only parameters to child paraprocs can be accessed via the parent's frame pointer, which is copied into a new frame at paraproc creation.

These optimizations can also be used in generating code for parallel loops. In this case, it may be possible for the generated code and run-time support to revert back to the baseline self-scheduling system. If the compiler can determine that no other procedure-processes may be active, frame allocation may be bypassed, and the global registers or well-known memory locations may be initialized directly. If other paraprocs may be present, frame allocation and initialization are required, but parameter passing is not needed. The shared variables are referred to directly as globals, and registers may be used to hold their values between synchronization points. These techniques can also be applied to paraprocs that do not create any children, since they can be executed inline.

7.4 Operating System Interface

The role of the operating system (OS) in our parallel processing system is simple and straightforward. We still require the OS to support I/O requests, shared virtual memory, and library routines, e.g., timing support. The concepts of time and process priorities are omitted from our proposed language, because they complicate the implementation. On the other hand, we believe that a predefined dynamic memory allocation routine (similar to the Unix routine `malloc`) should be supported by the run-time system, in order to avoid unnecessary OS interference. This routine must account for the fact that multiple paraprocs may be calling it simultaneously. This is easily implemented by the run-time system, with its ability to directly execute instructions for synchronization. Separate memory areas for the frame pool and dynamic allocation should be acquired from the OS at program startup.

7.4.1 Virtual Processors

As mentioned in Chap. 2, the method we propose for OS scheduling of the parallel program is the *virtual processor* approach. The program is allocated some number of OS processes or threads, which it uses as virtual processors. The virtual processors begin execution of the run-time system as a user program. The OS manages the scheduling of virtual processors on physical processors. Clearly, a policy of allocating a number of virtual processors equal to the number of available physical processors should reduce OS level context switching overhead. The virtual processor approach to an OS interface has been used successfully by several compilers, e.g., the Ada compilers for the Sequent Symmetry and Encore Multimax, and parallel language environments [23]. In addition to scheduling virtual processors, the OS must also ensure that these OS processes can share their virtual memory space. This system enables the run-time system to run in user mode and manage the program level parallelism, even though it is transparent to the programmer.

In order to ensure that deadlock will not occur in the program, we require the OS scheduler to prevent starvation of any virtual processor. However, if a parent paraproc is not blocked at a merge point on a given virtual processor, the corresponding OS process may be relinquished by the run-time system at a scheduling point associated with a **complete** statement, decreasing the program's allocation of virtual processors by 1. This is desirable in the presence of increasing system load, where the number of OS processes due to other jobs may increase to be far greater than the number of physical processors.

With some modification to the run-time system, it is also possible for the program to relinquish a virtual processor at a paraproc **merge** point. When a paraproc blocks at a merge point, it saves its registers and a pointer to its synchronization counter on the stack, and branches to the scheduling kernel. Alternatively, a system call could occur to relinquish the virtual processor. However, all virtual memory allocated to the program must be retained, because the stack space essentially contains the context of the blocked paraproc. These *non-resident* stack pointers can be saved in a stack pointer queue. Then, in the future when other virtual processors are looking for work, they can switch stack pointers and resume the blocked paraproc. If a stack pointer switch condition is checked for when the run queue is empty and no ancestor on the current stack is available for resumption, the conditions for preventing deadlock are not violated. Any blocked paraproc in a non-resident stack will eventually be resumed, enabling the resumption of any ancestor paraprocs waiting for it to complete.

7.4.2 I/O

The other major requirement of the operating system is the support for I/O operations. As mentioned in Chap. 2, problems can occur when multiple microtasks in a single program are attempting I/O operations using the same file descriptor information. In general, we believe that the program should be responsible for synchronizing I/O requests between microtasks. This is most easily implemented by coordinating I/O operations in parent paraprocs, after child paraprocs complete. The low overhead synchronization present in the paraproc system makes this a reasonable approach. However, in the case of heterogeneous parallelism, multiple I/O requests may be executed simultaneously by different parent paraprocs. In this situation, it is likely that different file descriptors will be used for what are apparently unrelated I/O activities. However, some I/O support for synchronizing I/O requests may be necessary as stated in Chap. 2.

Another issue related to parallel program I/O activity is blocking and scheduling of virtual processors. If a virtual processor is blocked during I/O activity, and then resumed following it, an OS level context switch takes place which temporarily blocks the microtask that virtual processor was executing. If the block time is significant, it can hamper the progress of the rest of the program if it is waiting for completion of that microtask. However, the context switch after I/O completion will resume the microtask at the point of the system call, and the program can proceed without experiencing deadlock.

Alternatively, some systems, e.g., Mach, have introduced an asynchronous system call which allows the virtual processor to proceed while the I/O activity is executing. The status of the call is checked at a later time, to determine if the microtask may be resumed. This approach can be incorporated into our run-time system by treating the I/O request as a creation of a child paraproc. The microtask simply blocks at a merge point using the stack in the usual way. The code generation for the system call and block/merge sequence will be modified slightly to make the pointer to the synchronization counter act as a check on the completion of the system call. This may require some operating system support that indicates system call completion by zeroing out a word at a given address. This approach can be used with any system call which the operating system is willing to execute asynchronously.

7.4.3 Scheduling

Relevant to the discussion on I/O and asynchronous system calls is the issue of OS-level scheduling and its impact on the parallel program. As mentioned in Chap. 2, some proposals have been made that advocate gang scheduling or coscheduling, where all virtual processors for a given parallel program are scheduled and preempted *en mass*. This approach is probably necessary when all virtual processors are executing microtasks involved in a `doacross` loop, since the inter-iteration synchronization does not result in a run-time system level context switch. In this case, coscheduling avoids a situation where all but one microtask is executing a busy wait, wasting processor cycles. However, in general we believe that the support for nested heterogeneous parallelism in our system will avoid this pathological situation. Because the run-time system can adapt to a dynamically varying allocation of virtual processors while still avoiding deadlock, the coscheduling requirement is not necessary. If a situation arises where most virtual processors cannot schedule work, they may be deallocated, thus freeing up resources for other jobs. Other virtual processors may be acquired later in the program, and they will assume the stack pointers queued by the earlier virtual processors. Of course, a policy of coscheduling by the operating system will not harm the execution of a parallel program using our run-time system.

Perhaps a more important issue related to OS scheduling is the preemption of a virtual processor by the operating system for the purpose of running a different virtual processor from a different program. The blocking of the virtual processor may result in the blocking of a microtask or run-time system action related to synchronization. As long as the virtual processor is resumed eventually, deadlock will not result for the parallel program. Also, as mentioned above, the capability of the run-time system to free up resources when work cannot proceed will avoid the wasting of resources or operating system thrashing.

However, in this case, performance of the parallel program may suffer. A better approach in this situation would be for the operating system to "nudge" the program to voluntarily relinquish a physical processor. With this approach, a virtual processor voluntarily blocks when its microtask reaches a **merge** point. As described above, the stack pointer is saved, and the virtual processor is released. An optimization may retain the virtual processor for use later instead of deallocating it. This approach to context switching and resource allocation will require some operating system support as well as minor run-time system changes. Of course, there will be some situations where "nudging" is not practical, and the consequences of the first approach must be endured.

7.5 Summary

In this chapter, the integration of our paraproc run-time system with a language model, compiler, and multiprocessor operating system has been presented. The language model supports parallel loops, both implicit and explicit, as well as parallel procedures. The compiler requirements for such a language are straightforward. In most cases, the OS interface can be implemented using existing operating systems for shared-memory multiprocessors. Additional capabilities can be implemented with minor modification to both the run-time system and the operating system. We believe that these modifications can be easily implemented.

CHAPTER 8

SUMMARY AND CONCLUSIONS

In this thesis, we have presented a design and implementation of run-time support code for parallel programs targeted to a shared-memory multiprocessor. The run-time system presented supports both parallel loop and parallel procedure language models. For the parallel procedure language model, we have introduced language constructs that enable the expression of this form of parallelism without the possibility of deadlock, while not unnecessarily complicating the run-time system. We have also demonstrated the efficiency of our run-time support code through experimentation using our simulator testbed. These tests also aided in making final design choices for the run-time system, and identified performance bottlenecks. Finally, we have presented an approach to integrating our run-time support into a complete system comprised of a language, compiler, run-time system, and operating system. For the most part, the complete system can be implemented using software technology which is available today.

As a result of this work, we have observed a number of performance characteristics which impact design choices for compiler writers and run-time support implementors. Our research has led to the following conclusions:

- With our user-mode, language-specific run-time system, the major performance barriers are cache consistency overhead and load balancing, not run-time scheduling and synchronization.
- The best scheduling algorithm depends on the grain size and its variability, as well as memory referencing frequency and pattern.

- The improved load balancing of self-scheduling often offsets the higher cache hit rates of chunk or guided self-scheduling.
- Overall, hardware cache consistency techniques provide for much better performance than software techniques.

Our main contribution with this work is the first quantitative study of run-time support structures for shared-memory parallel programs. We have identified three major performance barriers which impede the speedup of parallel programs and quantified their impact on the performance of a multiprocessor system. We have also quantified the tradeoffs between parallel language semantics and implementation requirements, and used the results from the study of performance barriers to direct a new design for a parallel language model.

8.1 Alternative Architectures

In order to confirm our contribution to research in this area, the results of our quantitative study must be applicable to other shared-memory multiprocessor architectures that are similar to the Astronautics ZS series. At the architectural level, most existing machines in this class differ only in the processor/memory interconnect, hardware synchronization mechanism, or the presence/absence of shared registers. For example, the Sequent Symmetry multiprocessor, a shared-memory architecture which supports up to 30 processors, has a shared-bus interconnect to memory, a synchronization mechanism providing the functionality of test-and-set using cache-based locks, and no shared register set. Despite these differences to the ZS series, we can still apply our general conclusions to this particular architecture.

The shared-bus interconnect of the Symmetry provides cache consistency using a write-invalidate snooping protocol. This cache consistency scheme is equivalent to shared hardware consistency, in that it permits all caches to possess a single cache block simultaneously. The cache miss rates observed in our tests with shared hardware consistency are similar to those reported in [128], and are low enough so that the shared bus is not saturated. Thus, our parallel language model and run-time system is suitable for execution on machines with a shared-bus interconnect to memory.

The hardware synchronization mechanism of the Symmetry does not support fetch&op, but instead provides a test-and-set mechanism using cache-based locks. This mechanism is designed for lightly contended locks used in conjunction with short critical sections [71]. The test-and-set operation combined with very short critical sections is sufficient to satisfy the synchronization needs of our run-time system. However, despite the presence of heterogeneous parallelism, accesses to the head of the run queue may be heavily contended at times. This in turn causes lock variables to move repeatedly from cache to cache, consuming bus bandwidth and degrading performance. This situation can be overcome using one of the software enhancements to the locking protocol proposed in [71], which prevents thrashing by following a software protocol that delays access to lock variables until they are likely to be available. Thus, the absence of a fetch&op synchronization mechanism does not invalidate our results.

Finally, the Symmetry does not provide a shared register set like those found in the ZS-1. While we used these registers to hold run queue information and semaphore values, the same effect could be accomplished using well-known memory locations. The accesses to this data are mostly reads, so the values could exist in multiple caches simultaneously. All writes take place in short critical sections which can be controlled using the software enhancements to the locking protocol as mentioned above. This approach eliminates unnecessary migration of a memory word from cache to cache by serializing its access. This serialization is similar to that provided in hardware with a fetch&op synchronization primitive. The combination of software-enhanced cache-based locks and well-known memory locations allows our run-time system to be implemented efficiently in a system without fetch&op synchronization or shared registers.

8.2 Future Work

In addition to determining the structure of an efficient run-time system for a class of shared-memory multiprocessors, this work has raised several issues which should be addressed by future research:

- More parallel algorithms need to be coded using the new language constructs to further demonstrate the usefulness of the model.
- Tests with longer programs in a multiuser environment are necessary to further validate the conclusions. This testing will require a full implementation on a suitable multiprocessor operating system.
- For very fine-grained microtasks, a modified self-scheduling algorithm which increases data locality should be developed. This modified algorithm would still schedule one iteration at a time, but would select the iteration based on the processor identifier.
- Tests on different systems with greater numbers of processors need to be done to determine the scalability of the run-time system.

In conclusion, we can say that our test results warrant the implementation of our system on a shared-memory multiprocessor which is to be used as a dynamically scheduled time-shared server among many parallel programs. However, in addition to the points listed above, further experimentation may be useful to determine the system performance in the case of `doacross` loops, alternative cache consistency schemes, and the modifications proposed to improve operating system integration. The test data reported in this dissertation do confirm, though, that our approach to run-time system design and implementation is one which meets our goals for flexibility and efficiency.

APPENDICES

APPENDIX A

RUN-TIME SUPPORT CODE

A.1 Parallel Loop Run-Time Support Code

Figure A.1 shows the self-scheduling kernel code used both for self-scheduling and chunk scheduling. In this code, `A22` and `A16` are general purpose registers. The `SGEC S1` statement blocks the processor if the first semaphore register is not greater than or equal to zero. When the semaphore register satisfies this requirement, the processor is unblocked and may continue. The `SADDC` statement is a fetch&add operation on the first semaphore register. As a side effect of `SADDC`, the fetched value is stored into the semaphore copy register, special control register `SC`. This fetch is done before the add, which is effectively a decrement, since `A22` was initialized to `-1`. This fetched value is then copied into `A22`, and a test is done to see if it is less than zero. If so, a branch to `sched:` is executed, and the processor must re-attempt to acquire work. This may require blocking at the `SGEC` statement until more work is created and the first semaphore register is made to be greater than zero. If the fetched value copied into `A22` is greater than or equal to zero, an address is copied from the second semaphore register into `A16` via a `CPSAC` instruction. A branch to this address is then executed. The parallel loop setup code initializes the second semaphore register to contain the start address of the body of the parallel loop.

Figure A.2 shows an example of parallel loop setup code using self-scheduling and the kernel code in Fig. A.1. The `SLTC` statement waits for the value in the third semaphore

```
        .globl sched
sched:
        DUPAC     A22,   0xfff                #A22 = -1
        SGEC      S1                          #wait for sem reg S1 >= 0
        SADDC     S1,    A22                  #SC = S1, S1=S1+A22
        RCRC      A22,   SC                   #A22 = SC
        TLTAB     A0,    A22,   A0            #test A22<0
        JMPT      sched                       #if A22<0, go to sched
        CPSAC     A16,   S2                   #A16 = S2
        BRI       A16                         #branch to address in A16
```

Figure A.1. Self-scheduling kernel code.

register to become negative, indicating that all microtasks associated with the previous parallel loop have completed their scheduling operations and are done reading the semaphore registers. The CPASC statements then load values required by the loop body into several semaphore registers. The last three of these loads involve the argument pointer AP, the stack pointer SP, and the frame pointer FP. In this example, A17 is equal to the index value, initially the highest value in the range of indices for the DO loop. A16 is initialized to one less this value, and this number is copied into the semaphore registers to indicate the number of microtasks yet to be created. The third semaphore register, S3, is used to implement the barrier described above. The fourth semaphore register, S4, is used to implement a barrier indicating that all microtasks created in conjunction with this loop have completed and (if necessary) have written their results to memory. The first semaphore register, S1, is used by the scheduling kernel shown in Fig. A.1. The starting address for the parallel loop body is copied via A3 into a semaphore register. The jump to first: represents the prescheduling of the first microtask by the processor. This is possible because the general purpose registers read by the parallel loop are already initialized.

The starting address for the other processors which acquire microtasks through the self-scheduling kernel is the label begin:, and the code at this point initializes general purpose registers to be equal to values copied from the semaphore registers. Note that A17 is initialized to the value in A22, the value returned by the scheduling kernel. However, if this value is zero, a branch to last: occurs, which is the starting address for the code

```
          SLTC      S3                      #wait for S3 to become <0
          CPASC     S8,    A22              #copy general purpose (gp) -
          CPASC     S9,    A13              #registers to semaphore (s) -
          CPASC     S10,   A14              #registers
          CPASC     S11,   A15
          CPASC     S12,   A12
          CPASC     S28,   AP
          CPASC     S29,   SP
          CPASC     S30,   FP
          DUPA      A17,   A16              #A17=A16
          SUBAC     A16,   A16,   0x1       #A16=A16-1
          DUPAD     A3,    begin            #A3= address of begin
          CPASC     S3,    A16              #S3=A16
          CPASC     S2,    A3               #S2=A3
          CPASC     S4,    A16              #S4=A16
          CPASC     S1,    A16              #S5=A16
          JMP       first
begin:
          TEQAB     A0,    A22,   A0        #test if A22==0 (A0=0 always)
          CPSAC     AP,    S28              #copy argument (AP), stack (SP), -
          CPSAC     SP,    S29              #and frame (FP) pointers from -
          CPSAC     FP,    S30              #S28, S29, and S30
          JMPT      last                    #if A22==0, go to last
          DUPA      A17,   A22              #A17=A22
          CPSAC     A22,   S8               #copy s regs to gp regs
          CPSAC     A13,   S9
          CPSAC     A14,   S10
          CPSAC     A15,   S11
          CPSAC     A12,   S12
first:
          DUPAC     A1,    0xfff            #A1=-1
          SADDC     S3,    A1               #SC=S3, S3=S3+A1
          RCRC      A1,    C2              #A1=C2, read proc id
          DUPA      FP,    SP                      #FP=SP
          ANDAD     A1,    A1,    0x0000001f       #mask off high bits
          MULAC     A1,    A1,    0x100            #0x100=two cache blocks
          SUBA      FP,    FP,    A1               #FP=FP-A1
          SUBAC     A1,    A17,   0x1              #A1=A17-1
          ASRC      A1,    A1,    0xffe            #A1=A1*4
          ADDA      A15,   A15,   A1               #A15=A15+A1
```

Figure A.2. Self-scheduling setup code.

following the parallel loop.

The code beginning at the label `first:` is the last set of instructions to be executed by all microtasks before entry into the parallel loop body. The third semaphore register, `S3`, is first decremented to indicate the completion of copying values from the semaphore registers. The next several statements modify the `FP` pointer so that each microtask has a local memory area for storing locals. Any available general purpose register can be used in stead of the frame pointer if the frame pointer is required for other references global to all microtasks. This local area is acquired off the stack top, indexed according to a unique hardware identifier found in special control register C2. Each local area is the size of two cache blocks to avoid unnecessary false sharing between microtasks. The last operation by the code fragment is the initialization of general purpose register A15 using the base value it already contains and the microtask index value contained in A17.

Figure A.3 shows the first part of the chunk scheduling setup code for the same parallel loop. This code also uses the scheduling kernel shown in A.1, but is a replacement for the code given in A.2. This code is similar to the setup for self-scheduling, except a calculation of chunk size is performed and the number of microtasks to be created is roughly equal to the number of processors. The number of processors is read from semaphore register seven, S7, which has been previously initialized. If the number of iterations (stored in A16) is less than the number of processors, the chunk calculation is bypassed, and a chunk size of one is used. The code starting at `iokayb:` computes the chunk size by dividing the number of iterations by the number of processors in use. If the division is not exact, the number of microtasks to be created is increased by one plus the number of chunks left in the remainder less one. The code starting at `iskipc:` initializes the semaphore registers as before, except the chunk size and the total number of iterations are also stored.

Figure A.4 shows the second part of the chunk scheduling setup code. The code at `begin:` is almost the same as for self-scheduling, as is the code after `iskipd:`. The only difference is the calculation of beginning and ending iteration values at `first:` based on the chunk size A18 and the microtask index value A17. The code following computes the new FP and initializes A15 as before.

Figure A.5 show the scheduling kernel for guided self-scheduling. Due to the nature of the calculation of the number of iterations per microtask, this scheduling kernel is somewhat longer than that shown in Fig. A.1. The start of the kernel is the same, but the value in

```
        SLTC    S3                      #wait for S3<0
        CPASC   S8,   A22               #copy gp regs to s regs
        CPASC   S9,   A13
        CPASC   S10,  A14
        CPASC   S11,  A15
        CPASC   S12,  A12
        CPASC   S28,  AP                #copy AP, SP, FP to s regs
        CPASC   S29,  SP
        CPASC   S30,  FP
        CPSAC   A1,   S7                #read num procs into A1
        DUPAD   A3,   begin             #A3=begin address
        TLTAB   A0,   A16,  A1          #check if num iterations <
                                        #num procs
        JMPF    iokayb                  #if A16>=A1, go to iokayb
        DUPA    A1,   A16               #A1=A16, set num microtasks
        DUPAC   A2,   0x1               #A2=1, set chunk size
        JMP     iskipc                  #go to iskipc
iokayb:
        DIVA    A2,   A16,  A1          #compute chunk size
        REMA    A4,   A16,  A1          #check remainder
        TEQAB   A0,   A4,   A0          #test A4==0
        JMPT    iskipc                  #if A4==0, go to iskipc
        ADDAC   A1,   A1,   0x1         #increase microtask count
        SUBAC   A4,   A4,   0x1         #by more than one if
        DIVA    A4,   A4,   A2          #remainder is bigger
        ADDA    A1,   A1,   A4          #than chunk size
iskipc:
        ADDAC   A5,   A1,   0xfff       #A5=A1-1
        CPASC   S6,   A2                #store chunk size
        CPASC   S5,   A16               #store iteration ceiling
        CPASC   S3,   A5                #one less microtask count
        CPASC   S2,   A3                #store begin address
        CPASC   S1,   A5                #store microtask count
        SADDC   S4,   A1                #init barrier counter
        DUPA    A18,  A2                #init A17 and A18 for
        DUPA    A17,  A1                #preschedule
        JMP     first
```

Figure A.3. Chunk scheduling setup code, part 1.

the first semaphore register S1 is only used to guard entrance into the kernel, and is not used to provide an index value. Instead, the fourth semaphore register S4 is read to see if any iterations for the current parallel loop remain. If no iterations remain, the first semaphore register is zeroed to allow another processor to enter the kernel, and a branch to the address contained in the fifth semaphore register S5 occurs. At this address, a barrier is implemented, and one processor is selected to set up the next parallel loop.

```
begin:
        TEQAB    A0,   A22,  A0           #test for microtask num==0
        CPSAC    AP,   S28                #copy pointers from s regs
        CPSAC    SP,   S29
        CPSAC    FP,   S30
        JMPT     last                     #if micro num==0, go to last
        DUPA     A17,  A22                #A17=A22
        CPSAC    A22,  S8                 #copy s regs to gp regs
        CPSAC    A13,  S9
        CPSAC    A14,  S10
        CPSAC    A15,  S11
        CPSAC    A12,  S12
        CPSAC    A18,  S6
        CPSAC    A16,  S5
first:
        DUPAC    A1,   0xfff               #A1=-1
        SADDC    S3,   A1                  #done reading s regs
        MULA     A11,  A17,  A18           #compute iteration limit
        SUBAC    A10,  A17,  0x1           #and fist iteration
        TLTAB    A0,   A16,  A11           #for microtask
        JMPF     iskipd                    #if greater than ceiling,
        DUPA     A11,  A16                 #reduce upper value
iskipd:
        RCRC     A1,   C2                     #read proc id
        DUPA     FP,   SP                     #code to allocate temp
        ANDAD    A1,   A1,   0x0000001f       #variable space for
        MULAC    A1,   A1,   0x100            #each proc for microtask
        SUBA     FP,   FP,   A1
        ASRC     A1,   A10,  0xffe            #A1=A10*4
        ADDA     A15,  A15,  A1               #A15=A15+A1
```

Figure A.4. Chunk scheduling setup code, part 2.

If iterations do remain to be scheduled, a branch to `alloc:` occurs. At this point, a number of iterations to be scheduled is computed based on the ceiling of the number of iterations yet to be scheduled divided by the number of processors in use. After the calculation, the number of iterations remaining must be updated, and stored in the fourth semaphore register `S4`. If the value allocated is greater than 127, the adjustment to the count of remaining iterations must be done in pieces, since the `SADDC` instruction can only do byte arithmetic. After this calculation, the kernel is unlocked by zeroing the first semaphore register `S1`. Final adjustments are then made to `A15` and `A16` to determine the first and last iteration values respectively for this microtask.

Figure A.6 shows the setup code used in conjunction with the guided self-scheduling

```
        .globl gsched
gsched:
        DUPAC   A12,   0xfff           #A12 = -1
        SGEC    S1                     #wait for S1 >= 0
        SADDC   S1,    A12             #SC=S1, S1=S1+A12
        RCRC    A13,   SC              #A13 = SC
        TLTAB   A0,    A13,   A0       #if A13<0
        JMPT    sched                  #then go to sched
        CPSAC   A13,   S4              #A13=S4
        TEQAB   A0,    A13,   A0       #test A13==0
        CPSAC   A1,    S7              #read num procs into A1
        JMPF    alloc                  #if not A13==0, go to alloc
        CPSAC   A12,   S5              #A12=S5
        CPASC   S1,    A0              #S1=0
        BRI     A12                    #branch to address in A12
alloc:
        CPSAC   A14,   S6              #read iteration limit
        SUBA    A15,   A14,   A13      #compute first iteration
        DIVA    A16,   A13,   A1       #compute ceiling of
        REMA    A11,   A13,   A1       #remaining iterations
        TEQAB   A0,    A11,   A0       #over number of processors
        CPSAC   A12,   S2
        JMPT    norem
        ADDAC   A16,   A16,   0x1      #adjust to get ceiling
norem:
        DUPA    A11,   A16             #if necessary
again:
        TLEABC  A0,    A11,   0x7f     #test if greater than 127
        JMPT    okidok
        DUPAC   A17,   0xf81           #if so, compute new remaining
        SADDC   S4,    A17             #count in pieces
        ADDA    A11,   A11,   A17
        JMP     again
okidok:
        SUBA    A11,   A0,    A11
        SADDC   S4,    A11             #update remaining iter count
        CPASC   S1,    A0              #unlock kernel
        ADDA    A16,   A16,   A15      #compute start and end
        ADDAC   A15,   A15,   0x1      #iteration values
        BRI     A12
```

Figure A.5. Guided self-scheduling kernel code.

kernel code given in Fig. A.5. While the kernel code for guided self-scheduling is more complex than the other techniques, its setup code is simpler. However, due to the calculation involved with guided self-scheduling, the first microtask is not prescheduled with this setup code. The semaphore registers are initialized as before, and the values required by the kernel are also stored. The code at `begin:` reads the semaphore registers, initializes the

```
        CPASC     S8,    A22                       #copy gp regs to s regs
        CPASC     S9,    A13
        CPASC     S10,   A14
        CPASC     S11,   A15
        CPASC     S12,   A12
        CPASC     S28,   AP
        CPASC     S29,   SP
        CPASC     S30,   FP
        DUPA      A17,   A16                       #A17=A16
        DUPAD     A3,    begin                     #A3=begin address
        DUPAD     A4,    last                      #A4=last address
        CPSAC     A5,    S7                        #A5=S7, num of procs
        CPASC     S2,    A3                        #store begin addr
        CPASC     S5,    A4                        #and end addr
        SUBAC     A5,    A5,    0x1                #A5=A5-1
        CPASC     S6,    A17                       #store iteration total
        CPASC     S4,    A17
        CPASC     S3,    A5                        #init barrier register
        CPASC     S1,    A0                        #unlock kernel
        JMP       gsched                           #go schedule
begin:
        CPSAC     AP,    S28                       #load AP, FP, SP from
        CPSAC     SP,    S29                       #s regs
        CPSAC     FP,    S30
        DUPA      A17,   A15                       #get begin iteration
        DUPA      A11,   A16                       #get end iteration
        CPSAC     A22,   S8                        #copy s regs to gp regs
        CPSAC     A13,   S9
        CPSAC     A14,   S10
        CPSAC     A15,   S11
        CPSAC     A12,   S12
        RCRC      A1,    C2                        #A1=proc id
        DUPA      FP,    SP                          #code to allocate temp
        ANDAD     A1,    A1,    0x0000001f           #variable space
        MULAC     A1,    A1,    0x100
        SUBA      FP,    FP,    A1
        SUBAC     A1,    A17,   0x1                  #A1=A17-1
        ASRC      A1,    A1,    0xffe                #A1=A1*4
        ADDA      A15,   A15,   A1                   #A15=A15+A1
```

Figure A.6. Guided self-scheduling setup code.

iteration registers, and then computes the new frame pointer and value for A15 as before.

A.2 Parallel Procedure Run-Time Support Code

Figure A.7 shows the code to initialize the frame pool for the parallel procedure run-time system. The code is called as a regular sequential procedure from the main program during initialization. It is passed the starting address of the frame pool as well as its size in frames. Each frame is 20 bytes or 5 halfwords. One halfword is sufficient to hold and integer value or address. The pool initialization simply zeros out the first two words of each frame to indicate that they have not been allocated. The semaphore registers initialized in this routine are reserved for run-time system globals.

```
        #iframe(addr, num)
        .globl  _iframe_
_iframe_:
        SUBAC   FP,   SP,   0x8         #standard proc entry code
        SUBAC   SP,   SP,   0x8
        DUPA    ASQ,  A2                #save A2
        STHAC   A0,   FP,   0
        LDHAC   A0,   AP,   0x8         #load begin addr of pool
        LDHAC   A0,   AP,   0x10        #get pool size in frames
        DUPA    A1,   ALQ               #put begin addr in A1
        LDHA    A0,   ALQ,  A0          #another load to get pool size
        DUPA    A2,   ALQ               #put size in A2
        ADDAC   A1,   A1,   0x4         #align pool start to halfword
        ANDAD   A1,   A1,   0xfffffffc  #boundary
        CPASC   S8,   A1                #init pool pointer
        CPASC   S5,   A1                #init tail pointer
        MULAC   A2,   A2,   0x14        #framesx20, size of frame
        ADDA    A2,   A1,   A2          #A2=addr of first word
                                        #after pool
        DUPA    ASQ,  A0                #put 0 in store queue
        STHAC   A0,   A1,   0           #put 0 in the head of 1st frame
lop :                                   #loop to put 0 in 1st 2 words
        DUPA    ASQ,  A0                #of each frame
        DUPA    ASQ,  A0
        TLTAB   A0,   A1,   A2         #test if > end pointer
        STHAC   A0,   A1,   0x4
        STHAC   A1,   A1,   0x14       #A1=A1+20, put 0 at this addr
        JMPT    lop
        CPASC   S9,  A2                #init end of pool ptr
        DUPA    A1,   A2               #set return value
        LDHAC   A0,   FP,   0xc        #standard proc exit code
        LDHAC   A0,   FP,   0
        LDHAC   A0,   FP,   0x8
        LDHAC   A0,   FP,   0x10
        ADDAC   SP,   FP,   0x18
        DUPA    AP,   ALQ
        DUPA    A2,   ALQ
        DUPA    FP,   ALQ
        BRI     ALQ
```

Figure A.7. Frame pool initialization code.

Figures A.8, A.9, and A.10 show the code for adding a frame to the run queue. After entering the subroutine and saving registers A2, A3, and A4 on the stack, the code starting at rep: acquires a lock from the fourth semaphore register S4. Once the lock has been acquired, the values for queue tail pointer A1, the frame pool end pointer A4, and the frame pool head pointer A2 are loaded from the corresponding semaphore registers reserved to hold these values.

```
        #pid = aframe(num, routine, pptr, fp)
        .globl  _aframe_
_aframe_:
        SUBAC       FP,    SP,    0x10       #standard proc entry code
        SUBAC       SP,    SP,    0x10
        DUPA        ASQ,   A2                #save A2, A3, A4
        CPAX        XSQ,   A3,    A4         #on the stack
        STHAC       A0,    FP,    0
        STXC        A0,    FP,    0x8
        DUPAC       A1,    0xfff             #A1=-1
rep:
        SGEC        S4                       #wait for S4>0
        SADDC       S4,    A1                #SC=S4, S4=S4+A1
        RCRC        A2,    SC                #A2=SC
        TLTAB       A0,    A2,    A0         #test A2<0
        JMPT        rep                      #if so, go to rep
        CPSAC       A1,    S5                #copy tail ptr to A1
        CPSAC       A4,    S9                #copy end of pool ptr to A4
        CPSAC       A2,    S8                #copy pool ptr to A2
```

Figure A.8. Add frame code, part 1.

The code starting at back: in Fig. A.9 attempts to find an unallocated frame for addition to the run-queue. The first frame checked is the first one spatially following the current tail pointer. As the checking progresses with each spatially following frame, a check is made to see if the end of the pool is reached using A4. If so, a flag in A3 is updated and the next frame to be tested will be at the head of the frame pool. If the pool is passed through twice and no frame is found, a jump to die: occurs and the program halts. Alternatively, the code at die: could instead take action to expand the frame pool, probably via an operating system call.

The code after skip: in Fig. A.9 shows the test to determine if a frame is available. If the first two halfwords in a frame are zero, the frame is unallocated and may be used for the current group of paraprocs to be created. The code that follows loads the parameters passed to add frame via the argument pointer and stores them into the newly allocated frame. `A2` is also loaded with the run-queue tail pointer, and a test is done to see if it is equal to the newly allocated frame. This condition occurs if the run queue is currently empty. If the run queue is not empty, the frame at the tail of the run queue is set to point to the newly allocated frame, effectively adding this frame to the end of the queue.

The code in Fig. A.10 waits for the memory writes to the frame to be performed, and then updates the global tail pointer in the fifth semaphore register `S5`. The global run queue head pointer is then read from the second semaphore register `S2`, and tested to see if it is null. If not, the code jumps to `done:` where the lock protecting add frame is released and procedure return is executed. If the queue head pointer is null, the frame information is loaded directly into the semaphore registers for scheduling. However, the processor must first wait for all previously loaded microtasks to be scheduled, thus the barrier using `SLTC` on the third semaphore register.

Figure A.11 shows the calling sequence for add frame which essentially creates a group of paraprocs. The paraproc parameter addresses are first pushed on the stack, since parameters to paraprocs are passed by reference. Next, the parameters to add frame itself are pushed. These include a stack address that points to the paraproc parameter addresses pushed as described above. They also include the starting address of the code, the current frame pointer, and the number of paraprocs to be created in conjunction with this frame.

Figure A.12 shows the code generated to implement the completion of a paraproc. As stated in the comments, a paraproc jumps to this code fragment after restoring its stack to the way it found it and placing its own frame address in `A1`. The sixth semaphore register `S6` is used as a lock to protect the access of any synchronization counter. Using only one lock for all synchronization counter accesses did not prove to be a performance bottleneck in our tests, but additional locks and/or more sophisticated locking schemes could alternatively be

```
back:
        ADDAC     A1,    A1,    0x4       #start with next frame
        DUPA      A3,    A0               #initialize flag
agn:
        TLTAB     A0,    A1,    A4        #check for end of pool
        JMPT      skip
        TEQAB     A0,    A3,    A0        #check for twice thru loop
        SUBAC     A1,    A2,    0x10      #reset A1 to head of pool
        JMPF      die
        DUPAC     A3,    0x1              #set flag after once thru
skip:
        LDHAC     A1,    A1,    0x14      #load value
        TEQAB     A0,    ALQ,   A0        #and check for 0
        JMPF      agn                     #not zero, try again
        SUBAC     A1,    A1,    0x4       #check other frame slot for 0
        LDHA      A0,    A1,    A0
        TEQAB     A0,    ALQ,   A0
        JMPF      back                    #not zero, try again

        LDHAC     A0,    AP,    0x8       #load proc params, num
        LDHAC     A0,    AP,    0x10      #routine
        LDHAC     A0,    AP,    0x18      #pptr
        LDHAC     A0,    AP,    0x20      #fp
        CPSAC     A2,    S5               #A2=S5, tail pointer
        LDHA      A0,    ALQ,   A0        #another load to get value num
        DUPA      ASQ,   ALQ              #store other pts in new frame
        DUPA      ASQ,   ALQ
        DUPA      ASQ,   ALQ
        TEQAB     A0,    A2,    A1        #test if new frame==tail ptr
        STHAC     A0,    A1,    0x4       #initialize frame
        DUPA      ASQ,   ALQ
        STHAC     A0,    A1,    0x8
        DUPA      ASQ,   A0
        STHAC     A0,    A1,    0x10
        STHAC     A0,    A1,    0
        STHAC     A0,    A1,    0xc
        JMPT      dont                    #dont add frame to tail
                                          #if queue empty
        DUPA      ASQ,   A1               #add current frame to
                                          #end of queue
        STHAC     A0,    A2,    0xc
```

Figure A.9. Add frame code, part 2.

employed. After acquiring the lock, the paraproc decrements the synchronization counter associated with its own paraproc group. After memory is updated, the lock is released.

The next step is to check the stack top to see if an ancestor paraproc is blocked and has its context embedded in the stack. This is so if the value on the stack top is non-zero. This

```
dont:
        QUIET                            #wait for stores!
        CPASC     S5,    A1              #set queue tail to new frame
        CPSAC     A3,    S2              #A3=S2, queue head pointer
        TEQABC    A0,    A3,    0xfff    #test for null head ptr
        JMPF      done                   # if not null, then done
        LDHA      A0,    A1,    A0       #otherwise, put this frame
        LDHAC     A0,    A1,    0x4      #at the head of the queue
        SLTC      S3                     #wait for last frame to
                                         #be consumed
        DUPA      A2,    ALQ
        CPASC     S3,    A2              #load semaphore registers
        CPASC     S0,    ALQ             #with work to be scheduled
        CPASC     S2,    A1
        CPASC     S1,    A2
done:
        QUIET                            #wait for updates to complete
        CPASC     S4,    A0              #release lock
        LDHAC     A0,    FP,    0x14     #standard proc exit code
        LDHAC     A0,    FP,    0xc
        DUPA      AP,    ALQ
        LDHAC     A0,    FP,    0x8
        DUPA      A2,    ALQ
        LDHAC     A0,    FP,    0x0
        DUPA      A3,    ALQ
        LDHAC     A0,    FP,    0x10
        DUPA      A4,    ALQ
        LDHAC     A0,    FP,    0x18
        DUPA      FP,    ALQ
        ADDAC     SP,    SP,    0x20
        BRI       ALQ

        .globl    die
die:
        HALT
```

Figure A.10. Add frame code, part 3.

value is a pointer to the synchronization counter that the ancestor paraproc is waiting on. If the counter is zero, the parent is resumed by branching to the address stored at the next halfword on the stack. If there is no ancestor in this stack, or its synchronization counter is non-zero, a branch to the scheduling kernel is executed.

Figure A.13 shows the code generated by a parent paraproc for the purposes of merging with its children. The associated synchronization counter is checked first, and, if it is zero, all children have completed and execution may resume at `post:`. At this point, the frame

```
        ADDAC     ASQ,   FP,    0xfb4       #push parameter address
        ADDAC     ASQ,   FP,    0xf7c       #push parameter address
        ADDAC     ASQ,   FP,    0xf98       #push parameter address
        STHAC     SP,    SP,    0xff8
        STHAC     SP,    SP,    0xff8
        STHAC     SP,    SP,    0xff8
        DUPA      ASQ,   FP                 #push current frame ptr
        DUPA      ASQ,   SP                 #push parameter space addr
        STHAC     SP,    SP,    0xff8
        ORAD      ASQ,   A0,    _crunch_    #push start address
        STHAC     SP,    SP,    0xff8
        ADDAC     ASQ,   FP,    0xfd4       #push ptr to number of
        STHAC     SP,    SP,    0xff8       #paraprocs to be created
        ORAD      A1,    A0,    dreta
        STHAC     SP,    SP,    0xff8
        CPAXC     XSQ,   A1,    0x4
        STXC      SP,    SP,    0xff8
        CPAX      XSQ,   FP,    AP
        DUPA      AP,    SP
        STXD      SP,    SP,    0xffffff8
        JMP       _aframe_
dreta:
```

Figure A.11. Call to add frame, creating paraproc group.

used for the descendant paraprocs is deallocated by zeroing its starting address slot.

If the initial check of the children's paraproc synchronization counter fails, the parent must block by saving any registers it needs later on the stack and jumping to the scheduling kernel to obtain more work. In this example, only the current frame pointer and argument pointer are saved on the stack. This was typically the case in our codes, since registers are not often reused between procedure calls or non-nested loops. A pointer to the children's frame and the resumption address of the blocking parent are the last two items placed on the stack. The code at the resumption address simply reloads the registers and continues execution.

Figure A.14 shows the first part of the paraproc self-scheduling kernel. A processor actually enters this code at `sched:`, where it tests the first semaphore register `S1` to see if it is non-negative thus indicating that work to be scheduled is present in the semaphore registers. If the value is non-zero, it is the paraproc index value, and the processor has successfully acquired a microtask for execution. In this case, a branch the the address stored

```
        .globl  complete
complete:
        #one way jump to here with "leveled" stack,
        #which points to ancestor's synch pointer at stack top
        #frame address is passed in A1
lock:
        DUPAC       A2,    0xfff
        SGEC        S6                      #wait for S6>0, acquire lock
        SADDC       S6,    A2               #fetch-and-add
        RCRC        A2,    SC
        TLTAB       A0,    A2,    A0
        JMPT        lock
        LDHA        A0,    A1,    A0        #load value at synch ptr
        SUBAC       ASQ,   ALQ,   0x1       #subtract 1 to indicate
                                            #completion
        STHA        A0,    A1,    A0        #store result
        QUIET                               #wait for it to complete
        CPASC       S6,    A0               #release lock
        QUIET
        LDHA        A0,    SP,    A0        #load synch ptr on stack top
        DUPA        A1,    ALQ
        TEQAB       A0,    A1,    A0        #is it a valid ptr?
        JMPT        sched                   #no ancestor
                                            #otherwise look for parent

        .globl  look
look:
        LDHA        A0,    A1,    A0        #load value at synch ptr
        TEQAB       A0,    ALQ,   A0        #test if ready to resume (==0)
        JMPF        sched                   #not yet ready
        LDHAC       SP,    SP,    0x8
        BRI         ALQ                     #resume ancestor
```

Figure A.12. Paraproc completion code.

in the zeroth semaphore register is executed. If the value obtained from the first semaphore register is equal to zero, a branch to `next:` is executed to reload the semaphore registers with the values from the frame at the head of the run queue.

If the value obtained from the first semaphore register is negative, a branch to `bsched:` is executed so that the stack may be checked for a blocked ancestor. If one is present, a branch is executed to `look:`, which is defined in the complete sequence shown in Fig. A.12. After checking the state of the ancestor's synchronization counter, the processor may return to `sched:`. This essentially creates a "loose" loop where the semaphore registers and the ancestor's synchronization counter are alternately checked until one source provides

```
merge:
        # merge code - generated inline with parent
        # assume child frame address is in A1

        LDHA        A0,     A1,     A0              #load synch counter
        TEQAB       A0,     ALQ,    A0              #if 0, merge is complete
        JMPT        post
        DUPA        ASQ,    FP                      #else, save needed regs,
        DUPA        ASQ,    AP                      #a return addr, and the
        STHAC       SP,     SP,     0xff8           #synch ptr on the stack
        DUPAD       ASQ,    dresume
        STHAC       SP,     SP,     0xff8
        DUPA        ASQ,    A1
        STHAC       SP,     SP,     0xff8
        STHAC       SP,     SP,     0xff8
        JMP         sched                           #get some more work
dresume:
        LDHAC       SP,     SP,     0x8             #resume after
        LDHAC       SP,     SP,     0x8             #waiting for merge
        DUPA        AP,     ALQ                     #reload saved regs
        DUPA        FP,     ALQ
post:
        DUPA        ASQ,    A0                      #deallocate frame
        STHAC       A0,     A1,     0x4
```

Figure A.13. Code to merge with descendants.

schedulable work. If an ancestor is not present on the stack, the processor blocks on the first semaphore register, waiting for another to load work into the semaphore registers.

The code starting at `next:` either reloads the semaphore registers with work found at the head of the run queue or waits for work to be created if the run queue is empty. The global run queue head pointer is loaded from the second semaphore register `S2` to see if it is null. If so, the processor must wait. Otherwise, the next frame pointed to by the run queue head is also checked to see if more work is ready to be scheduled. If this pointer is null, the processor must wait. If not, the data is loaded from the queue into the semaphore registers, and the queue is updated. The fourth semaphore register `S4` is used as a lock to protect the integrity of the run queue, as previously shown in Fig. A.8. Once the lock is acquired, the contents of the frame pointed to by the head of the queue are checked again to see if it was updated between the first check and acquisition of the lock. If so, the semaphore registers may be loaded as stated earlier.

```
bsched:
        LDHA      A0,   SP,   A0      #check for ancestor
        DUPA      A1,   ALQ
        TEQAB     A0,   A1,   A0
        JMPF      look                #check ancestor status
        SGEC      S1

        .globl  sched
sched:
        DUPAC     A1,   0xfff
        SADDC     S1,   A1
        RCRC      A1,   SC            #acquire index value
        TLTAB     A0,   A1,   A0      #if negative, check
        JMPT      bsched              #for ancestor
        TEQAB     A0,   A1,   A0      #if zero, update queue
        JMPT      next
        CPSAC     A2,   S2            #if >0, get work
        CPSAC     A3,   0
        BRI       A3
next:
        DUPAC     A3,   0xfff         #decrement semaphore register
        SADDC     S3,   A3            #protection barrier
        CPSAC     A1,   S2            #get queue head pointer
        TEQABC    A0,   A1,   0xfff
        JMPT      bwait               #wait if no more work
        LDHAC     A0,   A1,   0xc
        DUPA      A2,   ALQ
        TEQAB     A0,   A2,   A0
        JMPF      okay                #load next frame
```

Figure A.14. Paraproc scheduling kernel, part 1.

If the processor is forced to wait for more work to be created, the run queue head pointer is explicitly set to null, and the lock protecting the run queue is released. At this point, another check for a blocked ancestor is made, enabling a flow in the loose loop described above. If no ancestor is present, the processor waits for the run queue head pointer to be updated. This will be done when another processor enters the add frame routine. Once the queue head is initialized with a new frame pointer, the processor may continue. In this last case, the processor in add frame will load the frame contents directory into the semaphore registers.

```
acquire:
         DUPAC      A2,    0xfff
         SGEC       S4                  #acquire queue lock
         SADDC      S4,    A2
         RCRC       A2,    SC
         TLTAB      A0,    A2,    A0
         JMPT       acquire
         LDHAC      A0,    A1,    0xc #check if next frame has work
         DUPA       A2,    ALQ
         TEQAB      A0,    A2,    A0
         JMPF       bokay
         SLTC       S3
         DUPAC      A2,    0xfff
         CPASC      S2,    A2           #reset queue head pointer
         CPASC      S4,    A0           #unlock queue
bwait:
         LDHA       A0,    SP,    A0   #check for ancestor
         DUPA       A1,    ALQ
         TEQAB      A0,    A1,    A0
         JMPT       wait
         JMP        look                #check ancestor status
wait:
         QUIET
         SGEC       S2                  #wait for more work
         JMP        sched
bokay:
         CPASC      S4,    A0           #unlock queue
okay:
         LDHA       A0,    A2,    A0
         LDHAC      A0,    A2,    0x4 #load frame data into
         DUPA       A1,    ALQ          #semaphore registers
         DUPA       A3,    ALQ
         SUBAC      A4,    A1,    0x1
         SLTC       S3
         CPASC      S2,    A2
         CPASC      S3,    A1           #will schedule first one, and
         CPASC      S0,    A3           #decrement S3 when we get there
         CPASC      S1,    A4
         BRI        A3
```

Figure A.15. Paraproc scheduling kernel, part 2.

APPENDIX B

ADDITIONAL PERFORMANCE DATA

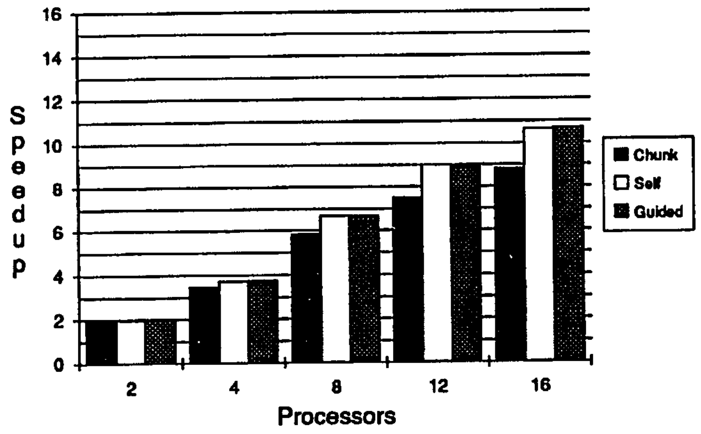

Figure B.1. Speedups for LFK21, version 1, n=50, assumed consistency.

Figures B.1, B.2, B.3, B.4, B.5, and B.6 show speedups for several different cache consistency configurations for both versions of LFK21 with n=50. The trends for n=25 are similar. Details can be found in [50].

Figures B.7, B.8, and B.9 show speedups for Gauleg using each type of cache consistency mechanism. In each case, self-scheduling provides the most speedup. Speedup also increases with problem size.

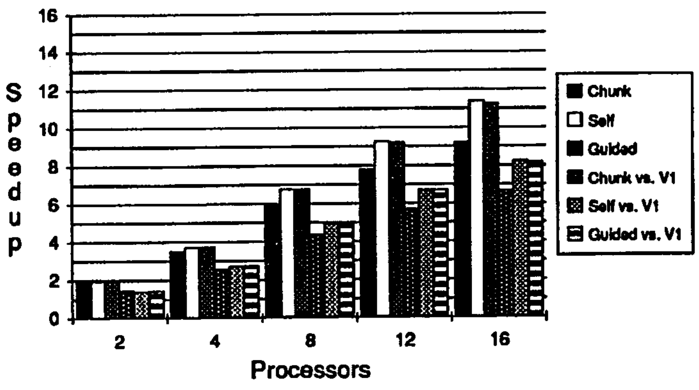

Figure B.2. Speedups for LFK21, version 2, n=50, assumed consistency.

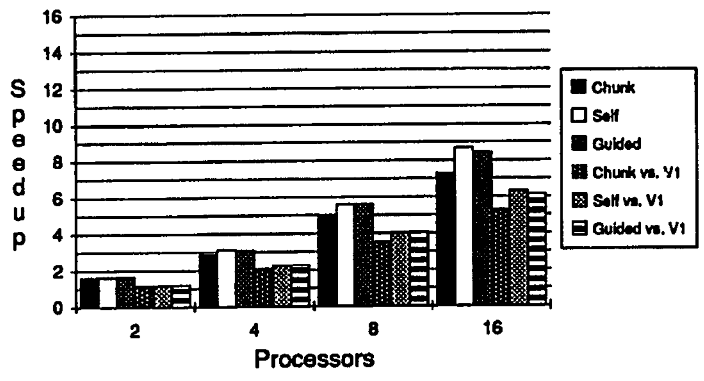

Figure B.3. Speedups for LFK21, version 2, n=50, non-cacheable result.

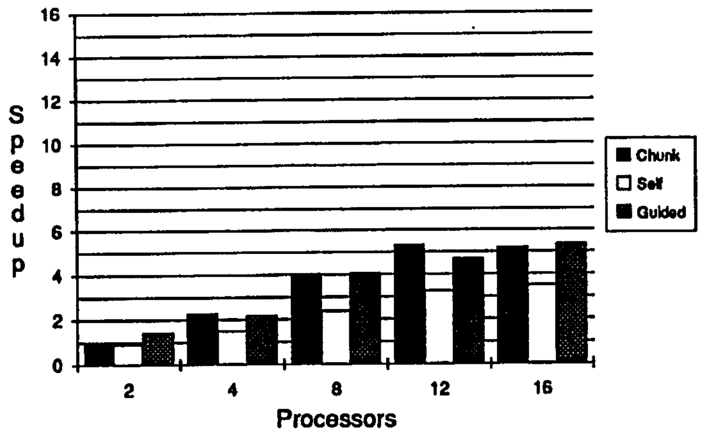

Figure B.4. Speedups for LFK21, version 1, n=50, private hardware consistency.

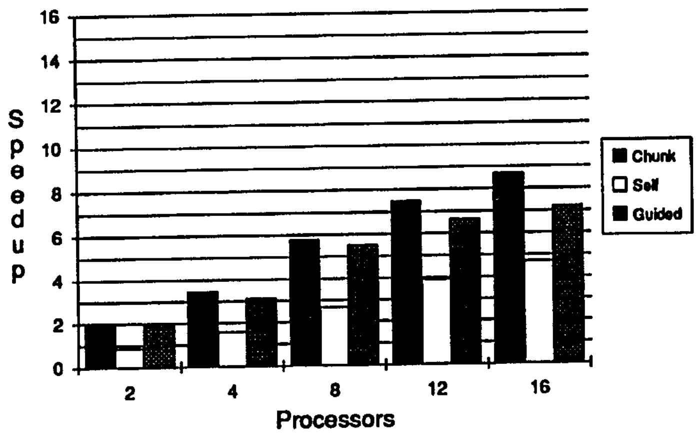

Figure B.5. Speedups for LFK21, version 1, n=50, shared hardware consistency.

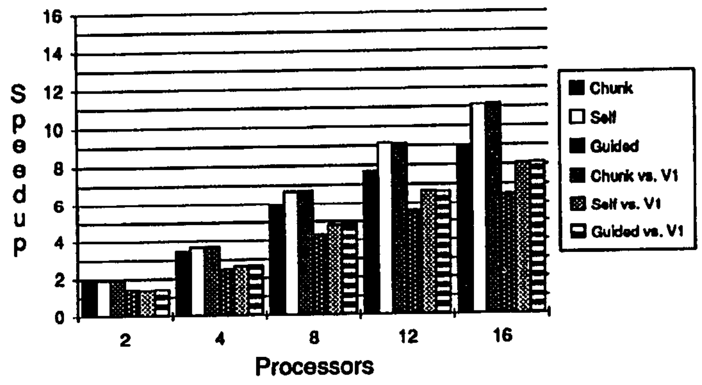

Figure B.6. Speedups for LFK21, version 2, n=50, shared hardware consistency.

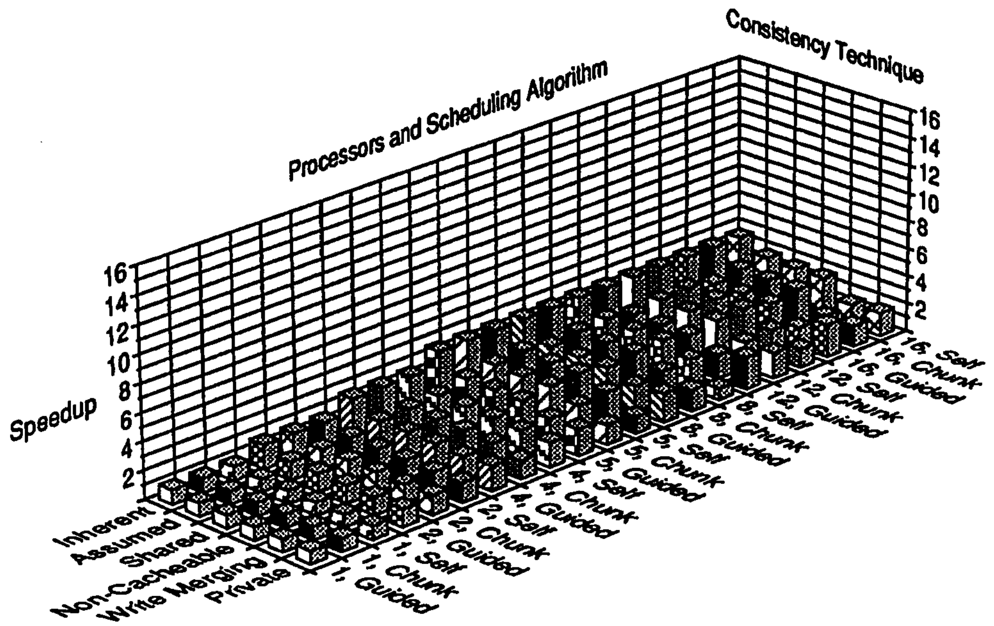

Figure B.7. Speedups for all versions of Gauleg, 5 iterations.

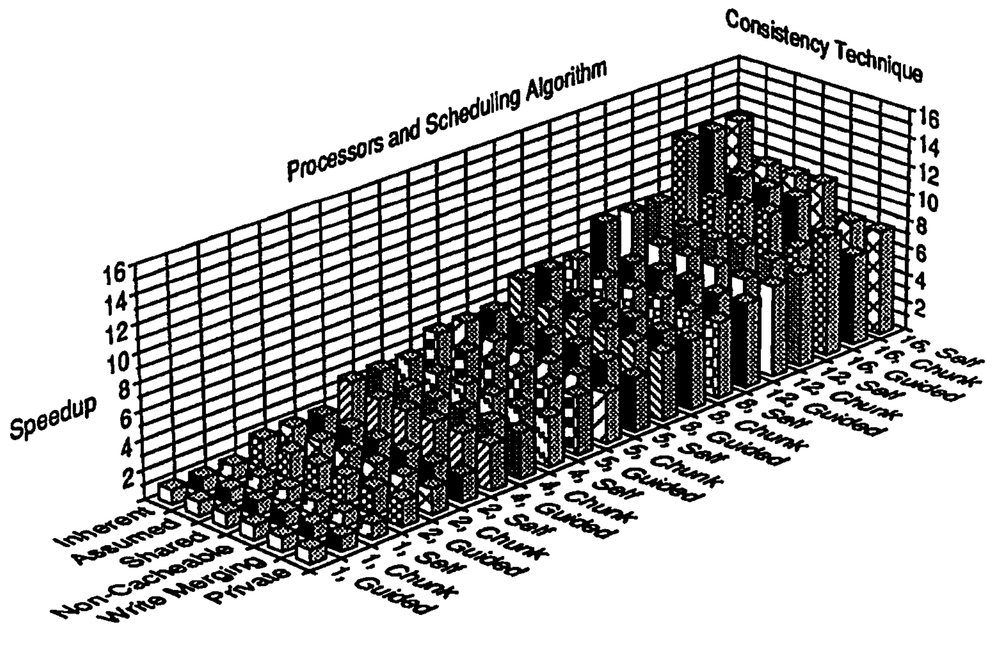

Figure B.8. Speedups for all versions of Gauleg, 25 iterations.

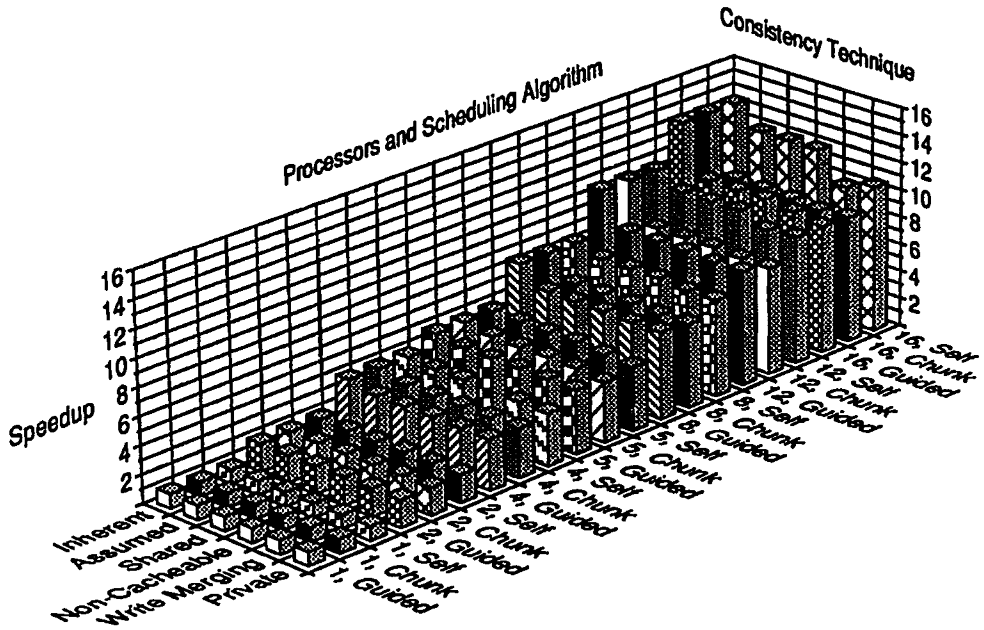

Figure B.9. Speedups for all versions of Gauleg, 50 iterations.

APPENDIX C

PARALLEL LANGUAGE SOURCE CODE

Figures C.1, C.2, C.3, C.4, C.5, and C.6 show the code for the parallel loop version of TRFD. In all cases of `doall` loops, the arrays referenced are shared amongst all microtasks participating in the loop's execution. The integer values are private to each iteration, and thus to each microtask. The data dependencies of a sequential program could, in general, produce a situation where integer values must be shared by multiple iterations. This could require restrictions on the compiler as far as placing these values in registers, and may also require `doacross` semantics with synchronization. With each of our example `doall` loops, such sharing of individual words was unnecessary.

The main program unit for the parallel loop version of TRFD is shown in Fig. C.1.

Figure C.2 shows the TRFD code for the `INTGRL` subroutine. This code contains two `doall` loops that may both execute simultaneously.

Figures C.3, C.4, C.5, and C.6 show the TRFD code for the `OLDA` subroutine. This code contains three `doall` loops. Each loop must complete entirely before any loop that follows it may begin. This is necessary to protect data dependencies.

Figures C.7, C.8, and C.9 show the modified code for the paraproc version of the TRFD program. The code for the `INTGRL` and `OLDA` subroutines is almost the same at the source level, except that the inner `DO 300` loop in `OLDA` is changed to a `doall` loop. Figure C.7 shows the main program, Fig. C.8 shows the `DRIVER` subroutine, and Fig. C.9 shows the `CRUNCH` paraproc. Although this modified source code indicates the changes made to use the paraproc system, in principle, it is possible for a compiler to automatically generate code for use with the paraproc run-time system based on a parallel loop program that is either hand-parallelized or automatically restructured.

Figures C.10, C.11, C.12, and C.13 show the code for the paraproc parallel quicksort program. Figure C.10 shows the driving code of the main program for only 100 array elements to be sorted. Our tests with larger array sizes used many more `DATA` statements to initialize the array. The quicksort paraproc itself is called `QSORT` and is shown in Fig. C.11. The parameter `M` is used to limit the depth of recursion. Uniprocessor tests showed little difference in performance based on `M`, but 7 did give the fastest running time. Figure C.12 shows the `FINDPIVOT` subroutine, and Fig. C.13 shows the `PARTITION` subroutine.

```
      program TRFD

      IMPLICIT REAL*8 (A-H,O-Z)
C...TRANSLATED BY FPP 2.26B16 12/12/89  09:21:51
C...MODIFIED BY RMC, I/O AND MEGAFLOP CALCULATION REMOVED
      COMMON/IJPAIR/IA(255)
      COMMON/MEMORY/X(2100000)
      COMMON/TRFPAR/NUM,NORB
      DATA MAX /2100000/
C
C     ----- SET MO'S AND AO INTEGRALS -----
c     Repeat calculation for different values of NUM
c
      DO 30 NUM=10,40,5
C
c     Loop to set up indicies into MEMORY common block for use
c     as compute storage in INTGRL and OLDA
      DO 10 I=1,NUM
   10 IA(I)=(I*(I-1))/2
      NORB=NUM
C
c     Compute indicies and loop limits for subroutine calls.
      NP=NUM
      NPQ=(NUM*(NUM+1))/2
      NRS=(NUM*(NUM+1))/2
      I00=1
      I10=I00+NUM*NUM
      I20=I10+NUM*NUM
      I30=I20+NORB*NUM*NRS
      I40=I30+NPQ*NRS
      LAST=I40
      NEED=LAST-1
      IF(NEED.GT.MAX) WRITE(IW,9998) NUM,NEED,MAX
 9998 FORMAT(' NOT ENOUGH CORE. NUM,NEED,MAX = ',I4,2I10)
      IF(NEED.GT.MAX) STOP
C
C     ----- CALCULATE -V- AND (IJ//KL) -----
c     INTGRL subroutine contains two doall loops.
      CALL INTGRL(NP,NPQ,X(I00),X(I30))
C
C     ----- TRANSFORM INTEGRALS -----
c     OLDA is only called once in following loop.  It contains
c     3 or 4 doall loops, depending on supported nest level.
      NTRF=1
      DO 20 ITRF=1,NTRF
      CALL OLDA(X(I00),X(I30),X(I30),X(I30),X(I30),
     1                 X(I20),X(I20),X(I10),X(I10),
     2                 NORB,NORB,NUM,NUM)
   20 CONTINUE
   30 CONTINUE
      STOP
      END
```

Figure C.1. TRFD main program code.

```
      SUBROUTINE INTGRL(NP,NPQ,V,X)
      IMPLICIT REAL*8 (A-H,O-Z)
C...TRANSLATED BY FPP 2.26B16 12/12/89  09:21:51
      REAL*8 A,B
      COMMON/IJPAIR/IA(255)
      DIMENSION V(NP,1),X(NPQ,1)
C
c     First doall loop, initializes a dummy vector.
c
      doall 20 J=1,NP
      DUM1=DFLOAT(J)
      DO 10 I=1,NP
      DUM2=DFLOAT(I)
      V(I,J)=1.0D+00-1.0D+00/(DUM1+DUM2)
   10 CONTINUE
   20 CONTINUE
C
c     Second doall loop, initializes X, part of MEMORY block,
c     for later use in OLDA.
c
      doall 140 I=1,NP
      DO 130 J=1,I
      IJ=IA(I)+J
      A=1.0D+00/(I+J)
      DO 120 K=1,I
      MAXL=K
      IF(K.EQ.I) MAXL=J
      DO 110 L=1,MAXL
      B = 1.0D+00 / (K + L)
      VAL = A + B
      IF (I .EQ. J) VAL = VAL * 0.5D+00
      IF (K .EQ. L) VAL = VAL * 0.5D+00
      X(IA(I)+J,IA(K)+L) = VAL
      X(IA(K)+L,IA(I)+J) = VAL
  110 CONTINUE
  120 CONTINUE
  130 CONTINUE
  140 CONTINUE
C
      RETURN
      END
```

Figure C.2. TRFD subroutine `INTGRL` code.

```
      SUBROUTINE OLDA(V,XRSPQ,XRSIJ,XIJRS,XIJKL,
     1 XRSIQ,XIJKS,XIJ,XKL,NORB,MORB,NUM,NDIM)
C
C     ----- THIS IS THE ORIGINAL -TRFCOR- FROM -HONDO- -----
C
      IMPLICIT REAL*8 (A-H,O-Z)
C...TRANSLATED BY FPP 2.26B16 12/12/89  09:21:51
      DIMENSION V(NDIM,1)
      DIMENSION XRSPQ(1),XRSIJ(1),XIJRS(1),XIJKL(1)
      DIMENSION XRSIQ(MORB,NUM,1),XIJKS(MORB,NUM,1)
      DIMENSION XIJ(1),XKL(1)
      DATA ZERO /0.0D+00/
C
C     ----- TRANSFORM -P- AND -Q- -----
C
      NRS=(NUM*(NUM+1))/2
      NP=NUM
      NQ=NUM
      MRSIJ0=0
      MRSPQ=0
C
c     First doall loop, begins calculation, values to be reused later.
c
      doall 100 MRS=1,NRS
      MRSPQ = NRS*(MRS-1)
C
      DO 15 M2Q = 1, NUM
      DO 15 MQ = 1, MORB
         XRSIQ(MQ,M2Q,MRS) = ZERO
   15 CONTINUE
      DO 40 MP=1,NP
      DO 30 MQ=1,MP
      MRSPQ=MRSPQ+1
      VAL=XRSPQ(MRSPQ)
      IF(VAL.EQ.ZERO) GO TO 30
C
      DO 20 MI=1,MORB
      XRSIQ(MI,MQ,MRS)=XRSIQ(MI,MQ,MRS)+VAL*V(MP,MI)
      XRSIQ(MI,MP,MRS)=XRSIQ(MI,MP,MRS)+VAL*V(MQ,MI)
   20 CONTINUE
   30 CONTINUE
   40 CONTINUE
```

Figure C.3. TRFD subroutine OLDA code, part 1.

```
      MRSIJ=MRSIJ0
      DO 90 MI=1,MORB
C
      DO 50 MJ=1,MI
      XRSIJ(MRSIJ+MJ)=ZERO
   50 CONTINUE

      DO 70 MQ=1,NQ
      VAL=XRSIQ(MI,MQ,MRS)
      IF(VAL.EQ.ZERO) GO TO 70
C
      DO 60 MJ=1,MI
      XRSIJ(MRSIJ+MJ)=XRSIJ(MRSIJ+MJ)+VAL*V(MQ,MJ)
   60 CONTINUE

   70 CONTINUE
      MRSIJ = MRSIJ + MI
   90 CONTINUE
C
      MRSIJ0=MRSIJ0+NRS
  100 CONTINUE
C
C     ----- TRANSPOSE (R,S//I,J) -----
C
      NIJ=(MORB*(MORB+1))/2
      MRSIJ=0
C
c     2nd doall loop, computes matrix transposition.
c
      doall 120 MRS=1,NRS
      MIJRS=0
      MAX=MRS
      IF(MAX.GT.NIJ) MAX=NIJ
      DO 110 MIJ=1,MAX
      DUM=XRSIJ(MRSIJ+MIJ)
      XIJRS(MRSIJ+MIJ)=XRSIJ(MIJRS+MRS)
      XIJRS(MIJRS+MRS)=DUM
  110 MIJRS=MIJRS+NRS
  120 MRSIJ=MRSIJ+NRS
```

Figure C.4. TRFD subroutine OLDA code, part 2.

```
C
C     ----- TRANSFORM -R- AND -S- -----
C
      NR=NUM
      NS=NUM
      MIJKL=0
      MIJRS=0
      MIJ=0
      MLEFT=NRS-NIJ
c     3rd doall for final calculation.
c
      doall 300 MI=1,MORB

      MIJOT = (MI*(MI-1))/2
      MIJRSOT = MIJ*((NR*(NR+1))/2)
      MIJKLOT = MIJOT*(NIJ+MLEFT)
      MIJKL=MIJKLOT

c     With minor modification, this loop can be made a doall as well.
c     Of course, this requires use of the paraproc run-time system.
c
      DO 300 MJ=1,MI
      MIJ=MIJOT+MJ
C
      DO 210 MS=1,NS
C
      DO 210 MK=MI,MORB
  210 XIJKS(MK,MS,MIJOT+MJ)=ZERO
C
      MIJRS=MIJRSOT+(NRS*(MJ-1))
      DO 240 MR=1,NR
      DO 230 MS=1,MR
      MIJRS=MIJRS+1
      VAL=XIJRS(MIJRS)
      IF(VAL.EQ.ZERO) GO TO 230
C
      DO 220 MK=MI,MORB
      XIJKS(MK,MS,MIJOT+MJ)=XIJKS(MK,MS,MIJOT+MJ)+VAL*V(MR,MK)
      XIJKS(MK,MR,MIJOT+MJ)=XIJKS(MK,MR,MIJOT+MJ)+VAL*V(MS,MK)
  220 CONTINUE
  230 CONTINUE
  240 CONTINUE
```

Figure C.5. TRFD subroutine OLDA code, part 3.

```
C
      MIJKL = MIJKLOT+((MJ-1)*(NIJ+MLEFT))
      LMIN=MJ
      LMAX=MI
      DO 290 MK=MI,MORB
C
      DO 250 ML=LMIN,LMAX
      XIJKL(MIJKL+ML)=ZERO
  250 CONTINUE

      DO 270 MS=1,NS
      VAL=XIJKS(MK,MS,MIJOT+MJ)
      IF(VAL.EQ.ZERO) GO TO 270
C
      DO 260 ML=LMIN,LMAX
      XIJKL(MIJKL+ML)=XIJKL(MIJKL+ML)+VAL*V(MS,ML)
  260 CONTINUE
  270 CONTINUE
      MIJKL = MIJKL + 1 + LMAX - LMIN
      LMIN=1
      LMAX=MK+1
  290 CONTINUE
C
  300 CONTINUE
      RETURN
      END
```

Figure C.6. TRFD subroutine OLDA code, part 4.

```
      program TRFD

      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/IJPAIR/IA(255)
      COMMON/MEMORY/X(4500000)
      COMMON/POOL/IFPOOL(5002)
      DATA MAX /4500000/
c
c     ----- Initialize frame pool ---------
c           should be done automatically
c
      CALL IFRAME(IFPOOL,1000)
C
C     ----- SET MO'S AND AO INTEGRALS -----
C
c     Replaces do loop to repeat entire program for
c     N between 10 and 40
      CALL DRIVER(10,40)

      STOP
      END
```

Figure C.7. TRFD main program code for the paraproc version.

```
      SUBROUTINE DRIVER(ILOW,IHIGH)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON /IJPAIR/ IA(255)
      COMMON /MEMORY/ X(4500000)
      DIMENSION INDICIES(7)
      DIMENSION NUMS(7)
      DIMENSION NPQS(7)
      INTEGER IBASE
C
c     Declaration for paraproc group handle.
c
      PID ICRUNCHERS
c
c     Compute array of indicies as in original program
c
      DO 10 I=1,IHIGH
 10      IA(I) = (I*(I-1))/2

c     Compute parameters for each child proc created.
c
      I=0
      IBASE=1

      DO 30 J=ILOW,IHIGH,5
         I=I+1
         NUMS(I)=J
         NPQS(I)=(J*(J+1))/2
         INDICIES(I)=IBASE
         IBASE = IBASE+1+(2*J*J)+(J*J*(NPQS(I)+J))+(NPQS(I)*NPQS(I))
 30   CONTINUE

c     Create child procs

      ICRUNCHERS = CREATE [I] CRUNCH(NUMS(*),NPQS(*),INDICIES(*))
c
c     Wait for children to complete.

      MERGE(ICRUNCHERS)

      RETURN
      END
```

Figure C.8. TRFD subroutine DRIVER code.

```
      PROCESS CRUNCH(NUM,NPQ,INDEX)
      INTEGER NUM(*),NPQ(*),INDEX(*)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON /IJPAIR/ IA(255)
      COMMON /MEMORY/ X(4500000)
      INTEGER NORB,NP,NRS,I00,I10,I20,I30,I40

c     Same code as previously in the mainline of TRFD.

      NORB = NUM
      NP = NUM
      NRS = NPQ
      I00 = INDEX
      I10 = I00+NUM*NUM
      I20 = I10+NUM*NUM
      I30 = I20+NORB*NUM*NRS
      I40 = I30+NPQ*NRS

      CALL INTGRL(NP,NPQ,X(I00),X(I30))

      CALL OLDA(X(I00),X(I30),X(I30),X(I30),X(I30),
     1                 X(I20),X(I20),X(I10),X(I10),
     2                 NORB,NORB,NUM,NUM)


      COMPLETE
      RETURN
      END
```

Figure C.9. TRFD subroutine CRUNCH code.

```
      PROGRAM QUICKSORT
      DIMENSION A(100)
PID ISORTERS
      DATA (A(I), I= 1,100) / 29.82,71.51,3.30,87.44,
     9 53.42,63.16,
     1 89.10, 25.75, 93.16, 27.72, 71.58, 48.34, 53.11, 18.34,
     2 27.13, 60.31, 83.34, 22.81, 66.84, 52.91, 53.42, 15.22,
     3 8.01, 53.39, 76.12, 79.09, 67.61, 38.39, 24.81, 73.21,
     4 13.42, 52.10, 34.86, 99.83, 38.46, 81.59, 61.75, 79.62,
     5 93.39,  3.21, 99.34, 92.22, 94.29,  7.03,  6.67, 89.35,
     6 83.14,  9.01, 12.68, 62.22, 2.95, 85.02, 95.82, 73.96,
     7 49.29, 77.72, 36.65,  3.48, 48.98, 71.83, 1.41,  9.48,
     8 32.37, 89.95, 28.39, 79.36, 54.05, 46.08, 11.67, 37.78,
     9 77.17, 74.33, 10.13,  4.62, 49.95, 68.40, 19.40, 34.06,
     1 4.11, 98.40, 42.44, 64.14, 89.41, 52.99, 71.79,  3.94,
     2 19.73, 44.91, 71.44, 59.10, 27.54, 15.67, 67.95, 55.61,
     3 26.05, 25.01, 82.09, 89.67, 57.08, 38.27/
C
      ISORTERS = CREATE [1] QSORT(100,A,1,N)
      END
```

Figure C.10. Main program for parallel quicksort.

```
      PROCESS QSORT(N,A,I,J)
c     Procedure process to perform quicksort on array A
c     of N values using subinterval between I and J.
      INTEGER N
      DIMENSION A(N)
      INTEGER I(*),J(*)
      PARAMETER(M=7)
      INTEGER FINDPIVOT,PARTITION
      INTEGER PIVOTINDEX,K
      DIMENSION IARG(2),JARG(2)
      PID SORTERS
c
c     If interval is less than M, then sort in place.
      IF (J-I.LT.M) THEN
         DO 13 IJ=I+1,J
            ATMP=A(IJ)
            DO 11 II=IJ-1,1,-1
               IF(A(II).LE.ATMP) GO TO 12
               A(II+1)=A(II)
11          CONTINUE
            II=0
12          A(II+1)=ATMP
13       CONTINUE
      ELSE
c
c     Find pivot index for interval I to J.
         PIVOTINDEX = FINDPIVOT(N,A,I,J)
         IF (PIVOTINDEX .NE. 0) THEN
            PIVOT = A(PIVOTINDEX)
c
c     Find partition around pivot.
            K = PARTITION(N,A,I,J,PIVOT)
c
c     Prepare parameters for child processes.
            IARG(1) = I
            IARG(2) = K
            JARG(1) = K - 1
            JARG(2) = J
c
c     Create 2 children to sort 2 sub-intervals.
            SORTERS = CREATE [2] QSORT(N,A,IARG(*),JARG(*))
c
c     Wait for children to complete.
            MERGE(SORTERS)
         ENDIF
      ENDIF
      COMPLETE
      RETURN
      END
```

Figure C.11. Quicksort paraproc QSORT code.

```
      INTEGER FUNCTION FINDPIVOT(N,A,I,J)
c
c     Find a pivot value for subinterval I to J in
c     array of size N.

      DIMENSION A(N)
      INTEGER K
c
c     Pivot index is either I or index of first value
c     encountered in A between I and J that is greater
c     than A(I).
c
      FIRSTKEY = A(I)
      DO 15 K=I+1,J
         IF (A(K).GT.FIRSTKEY) THEN
            FINDPIVOT=K
            GOTO 16
         ELSE
            IF (A(K).LT.FIRSTKEY) THEN
               FINDPIVOT=I
               GOTO 16
            ENDIF
         ENDIF
15    CONTINUE
      FINDPIVOT=0
16    RETURN
      END
```

Figure C.12. Quicksort subroutine FINDPIVOT code.

```
      INTEGER FUNCTION PARTITION(N,A,I,J,PIVOT)
c
c     Partition of subinterval between I and J
c     in array A of size N using pivot value PIVOT.
c
      DIMENSION A(N)
      INTEGER L,R

      L=I
      R=J
19    CONTINUE
c
c     Swap values at L and R.
c
         ATMP=A(L)
         A(L)=A(R)
         A(R)=ATMP
20       IF (A(L).LT.PIVOT) THEN
c
c     Increase L while values less than pivot.
c
            L=L+1
            GO TO 20
         ENDIF
21       IF (A(R).GE.PIVOT) THEN
c
c     Decrease R while values less than pivot.
c
            R=R-1
            GO TO 21
         ENDIF
      IF (L.LE.R) GO TO 19

      PARTITION = L
      RETURN
      END
```

Figure C.13. Quicksort subroutine PARTITION code.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Perfect Club Benchmark Suite 0 documentation. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, March 1989.

[2] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX 1986 Summer Technical Conference*, pages 193–210, June 1986.

[3] S. V. Adve and M. Hill. Weak ordering - a new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–11, June 1990.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, June 1988.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addision Wesley, Reading, Massachusetts, 1986.

[7] R. Allen and K. Kennedy. PFC: A program to convert FORTRAN to parallel form. Technical Report MASC-TR 82-6, Department of Mathematical Sciences, Rice University, March 1982.

[8] G. A. Alverson. *Abstractions for Effectively Portable Shared Memory Parallel Programs*. PhD thesis, University of Washington, 1990. Department of Computer Science and Engineering.

[9] American National Standards Institute. FORTRAN *8x, X3J3/S8(X3.9-198x)*, 1986.

[10] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[11] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. Technical Report 90-04-02, The Department of Computer Science and Engineering, University of Washington, October 1990.

[12] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions Computers*, 38(12):1631–1644, December 1989.

[13] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilson, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[14] M. Annaratone and R. Rühl. Performance measurements on a commercial multiprocessor running parallel code. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 307–314, June 1989.

[15] B. Applebe, K. Smith, and C. McDowell. Start/Pat: A parallel-programming toolkit. *IEEE Computer*, pages 29–38, July 1989.

[16] J. Archibald and J. L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 355–362, June 1985.

[17] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[18] Astronautics Corporation of America. *ZS Central Processor – Architecture Reference Manual*. Madison, WI, 1988.

[19] R. G. Babb II, ed. *Programming Parallel Processors*. Addison-Wesley, Reading, Mass., 1988.

[20] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, 1988.

[21] J. G. P. Barnes. *Programming in Ada*. Addison-Wesley, Reading, MA, second edition, 1984.

[22] B. Beck, B. Kasten, and S. Thakkar. VLSI assist for a multiprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 10–20, October 1987.

[23] B. Beck and D. Olien. A parallel-programming process model. *IEEE Software*, pages 63–72, May 1989.

[24] M. Beltrameti, K. Bobey, and J. R. Zorbas. The control mechanism for the Myrias parallel computer system. *ACM Computer Architecture News*, August 1988.

[25] P. Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. In *Proceedings of the Cache and Interconnect Workshop, 16th International Symposium on Computer Architecture*, pages 37–52. Kluwer Academic Publishers, Norwell, MA, 1989.

[26] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, pages 35–43, May 1990.

[27] P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[28] E. D. Brooks III. PCP: A parallel extension of C that is 99% fat free. Technical report, Lawrence Livermore National Laboratory, 1988.

[29] J. C. Browne, M. Azam, and S. Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, pages 10–18, July 1989.

[30] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the 1986 Symposium on Compiler Construction, ACM SIGPLAN Notices*, 21(7):162–175, July 1986.

[31] D. Callahan, K. Cooper, K. Kennedy, and L. Torczan. Interprocedural constant propagation. *Proceedings of the 1986 Symposium on Compiler Construction, ACM SIGPLAN Notices*, 21(6), June 1986.

[32] P. Carnevali, P. Sguazzero, and V. Zecca. Microtasking on IBM multiprocessors. *IBM Journal of Research and Development*, 30(6):574–582, November 1986.

[33] N. Carriero and D. Gelernter. Applications experience with Linda. In *Proceedings of Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS 1988)*, pages 173–187, July 1988.

[34] L. M. Censier and P. Feautrier. A new solution to coherence. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[35] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, pages 49–58, June 1990.

[36] M. Chastain, G. Gostin, J. Mankovich, and S. Wallach. The Convex C240 architecture. In *Proceedings of Supercomputing '88*, pages 321–329, November 1988.

[37] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 239–248, June 1990.

[38] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 299–307, June 1988.

[39] H. Cheong and A. V. Veidenbaum. Software-directed cache management in multiprocessors. In *Proceedings of the Cache and Interconnect Workshop, 16th International Symposium on Computer Architecture*, pages 259–276. Kluwer Academic Publishers, Norwell, MA, 1989.

[40] H. Cheong and A. V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, pages 39–47, June 1990.

[41] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Software controlled caches in the VMP multiprocessor. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 366–374, June 1986.

[42] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze. Toward real-time performance benchmarks for Ada. *Communications of the ACM*, 29(8):760–778, August 1986.

[43] R. M. Clapp and T. Mudge. Ada on a hypercube. In *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, pages 399–408, Pasadena, CA, January 1988.

[44] R. M. Clapp and T. Mudge. Ada on a hypercube. *ACM Ada Letters*, IX(2):118–128, March/April 1989.

[45] R. M. Clapp and T. Mudge. A parallel language for a distributed-memory multiprocessor. In *Proceedings of The Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 515–522, Monterey, CA, March 1989.

[46] R. M. Clapp, T. Mudge, and D. Roy et al. A rationale for the design and implementation of Ada benchmark programs. *ACM Ada Letters, Special Edition on Performnce Issues*, X(3):7–90, Winter 1990.

[47] R. M. Clapp, T. Mudge, and D. C. Winsor. Cache coherence requirements for interprocess rendezvous. *International Journal of Parallel Programming*, 19(1):31–51, February 1990.

[48] R. M. Clapp and T. N. Mudge. Distributed Ada on a loosely coupled multiprocessor. Technical Report RSD-TR-3-88, Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, January 1988.

[49] R. M. Clapp and T. N. Mudge. Parallel language constructs for efficient parallel processing. Technical Report CSE-TR-66-90, Advanced Computer Architecture Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, June 1990.

[50] R. M. Clapp, T. N. Mudge, and J. E. Smith. Performance of parallel loops using alternative cache consistency protocols on a non-bus multiprocessor. In *Proceedings of the Cache and Interconnect Workshop, 16th International Symposium on Computer Architecture*, pages 131–152. Kluwer Academic Publishers, Norwell, MA, 1989.

[51] R. M. Clapp, T. N. Mudge, and R. A. Volz. Distributed run-time support for ada on the ncube hypercube multiprocessor. Technical Report RSD-TR-10-87, Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, August 1987.

[52] R. F. Cmelik, N. H. Gehani, M. Plotnick, and W.D. Roome. Concurrent C project. Technical report, AT&T Bell Laboratories, 1985.

[53] Cray Research Inc. *Cray X-MP Mainframe Reference Manual*. Minneapolis, MN, 1982.

[54] Cray Research Inc. *Cray X-MP Multitasking Manual*. Minneapolis, MN, 1982.

[55] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[56] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 229–238, August 1986.

[57] Department of Defense, OUSD (R&D), Washington, D.C. 20301: Ada Joint Program Office. *Ada Programming Language (ANSI-MIL-STD-1815A).*, January 1983.

[58] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[59] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[60] M. Dubois and S. S. Thakkar. Workshop report: Cache and interconnect architectures in multiprocessors. *ACM Computer Architecture News*, 17(6):105–110, December 1989. comments by J. Goodman.

[61] J. Edler, J. Lipkis, and E. Schonberg. Process management for highly parallel Unix systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, 1988.

[62] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, April 1989.

[63] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, June 1989.

[64] Encore Computer Corporation. *The Multimax Family of Computer Systems.* 257 Cedar Hill Street, Marlboro, MA 01752, 1988.

[65] D. G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, pages 65–77, May 1990.

[66] N. H. Gehani and W. D. Roome. Concurrent C. *Software Practice and Experience*, 16(9):821–844, September 1986.

[67] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[68] D. A. George. EPEX – environment for parallel execution. Technical Report RC 13158 (#58851), IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1987.

[69] A. Gottlieb et al. The NYU ultracomputer—Designing a MIMD, shared memory parallel machine. *IEEE Transactions on Computers*, C-32:175–189, February 1983.

[70] C. M. Grassl and J. L. Schwarzmeier. Performance of applicaiions programs on supercomputers: Results from the Perfect Benchmarks. Technical report, Cray Research, Inc., Mendota Heights, Minnesota, April 1990.

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

[72] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume I, pages 312–321, August 1990.

[73] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, and D. H. Lawrie. Cedar FORTRAN and other vector and parallel FORTRAN dialects. In *Proceedings of Supercomputing '88*, pages 114–121, November 1988.

[74] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[75] D. E. Hudak and S. G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 187–200. ACM Press, June 1990.

[76] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, New York, 1984.

[77] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Breckner, O. Roubine, and B. A. Wichmann. Rationale for the design of the Ada programming language. *ACM SIGPLAN Notices*, 14(6), June 1979.

[78] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable coherent interface. *IEEE Computer*, pages 74–77, June 1990.

[79] R. Jha, G. Eisenhauer, J. M. Kamrad II, and D. Cornhill. An implementation supporting distributed execution of partitioned Ada programs. *ACM Ada Letters*, 9(1), January/February 1989.

[80] M. Kallstrom and S. S. Thakkar. Programming three parallel computers. *IEEE Software*, pages 11–22, January 1988.

[81] A. H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[82] A. H. Karp and R. G. Babb. A comparison of 12 parallel FORTRAN dialects. *IEEE Software*, pages 52–67, September 1988.

[83] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[84] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12), December 1972.

[85] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[86] J. L. Larson. Multitasking on the Cray X-MP-2 multiprocessor. *IEEE Computer*, pages 62–69, July 1984.

[87] B. Leasure et al. PCF FORTRAN: Language definition. Technical Report Version 1, The Parallel Computing Forum, August 1988.

[88] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *The 1987 International Conference on Parallel Processing*, pages 28–31, August 1987.

[89] T. G. Lewis, H. El-Rewini, J. Chu, P. Fortner, and W.-J. Su. Task Grapher: A Tool for Scheduling Parallel Program Tasks. Technical Report TR-PPSE-89-12, Oregon Advanced Computing Institute (OACIS), 1989.

[90] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS 1988)*, pages 85–97, July 1988.

[91] Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[92] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[93] D. May. Occam. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[94] Frank H. McMahon. The Livermore FORTRAN kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA 94550, December 1986.

[95] A. J. Musciano and T. L. Sterling. Efficient dynamic scheduling of medium-grained tasks for general purpose parallel processing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 166–175, August 1988.

[96] T. D. Newton. An implementation of Ada tasking. Technical Report CMU-CS-87-169, Carnegie Mellon University, October 1987.

[97] A. Norton and W. L. Chang. Self-scheduling in the runtime environment. Technical Report RC 12572 (#56256), IBM T. J. Watson Research Center, Yorktown Heights, NY, February 1987.

[98] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, June 1990.

[99] D. M. Olien. Parallel Ada tasking in Balance computers. In *Proceedings of the Second International Conference on Supercomputing*, pages 28–37.

[100] A. Osterhaug. *Guide to Parallel Programming*. Sequent Computer Systems, Inc., 1985.

[101] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 230–242, April 1989.

[102] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, pages 12–24, June 1990.

[103] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[104] F. N. Parr and R. E. Strom. NIL: A high level language for distributed systems programming. *IBM Systems Journal*, 22(1 and 2), April 1983.

[105] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[106] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell, MA, 1988.

[107] C. D. Polychronopoulos. Toward auto-scheduling compilers. *The Journal of Supercomputing*, pages 297–330, 1988.

[108] C. D. Polychronopoulos. Multiprocessing versus multiprogramming. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 223–230, August 1989.

[109] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2 : An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 39–48, August 1989.

[110] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[111] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, England, 1986.

[112] A. P. Reeves. Parallel Pascal: An extended Pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, 1984.

[113] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, 29(12):1202–1212, December 1986.

[114] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1986.

[115] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[116] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.

[117] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):356–364, July 1990.

[118] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, June 1988.

[119] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[120] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time Processing IV, Proceedings of SPIE*, pages 241–248, 1981.

[121] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[122] J. E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[123] L. Snyder. Parallel programming and the Poker programming environment. *IEEE Computer*, pages 27–36, July 1984.

[124] T. L. Sterling, A. J. Musciano, E. Y. Chan, and D. A. Thomae. Effective implementation of a parallel language on a multiprocessor. *IEEE Micro*, pages 46–62, December 1987.

[125] H.-M. Su and P.-C. Yew. On data synchronization for multiprocessors. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 416–423, June 1989.

[126] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *AFIPS Proc. National Computer Conf.*, volume 45, pages 749–753, 1976.

[127] J. A. Test. Muiltprocessor management in the Concentrix operating system. In *Proceedings of the USENIX 1986 Winter Technical Conference*, pages 173–182, January 1986.

[128] S. S. Thakkar. Performance of symmetry multiprocessor system. In *Proceedings of the Cache and Interconnect Workshop, 16th International Symposium on Computer Architecture*, pages 53–82. Kluwer Academic Publishers, Norwell, MA, 1989.

[129] M. Thapar and B. Delagi. Stanford distributed-directory protocol. *IEEE Computer*, pages 78–80, June 1990.

[130] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proceedings of Supercomputing '88*, pages 35–41, November 1988.

[131] R. H. Thomas and W. Crowther. The Uniform System: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[132] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 243–256, April 1989.

[133] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[134] M. Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, pages 34–42, January 1988.

[135] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 330–343, July 1990.