ON RUN-TIME SYSTEMS FOR PARALLEL SUPERCOMPUTERS

BY

CHIA-LING LEE

B.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ABSTRACT

This thesis considers the implementation of an auto-scheduling compiler. It first reviews the run-time library support on multiprocessor supercomputer systems, including the microtasking run-time library of Cedar. The implementation of an auto-scheduling compiler based on the Cedar run-time library and the Parafrase-2 restructurer is then outlined. Improvements to the model used for auto-scheduling are discussed in the final chapter.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Constantine Polychronopoulos for his patience and invaluable suggestions and guidance. My fellow graduate students who worked on the Parafrase-2 project, Milind Girkar, Mohammad-Reza Haghighat, Bruce Leung, and Dale Schouten, have kindly given me useful hints and advice. My friends Tian-Pin Ng, Mui-Joo Wee, Ching-Leng Kee, Poh-Lin Lim, Tiow-Seng Tan, Joo-Joo Chia, Jee-Loong Look, Chin-Chau Low, Sharon Chew, Gih-Guang Hung, Yeow-Hin Lai and Dennis Lee, have made my stay in the United States a pleasant and unforgettable experience. Finally, I would like to thank my family who put up with my absence and the National Computer Board of Singapore for sponsoring my undergraduate education here in the United States.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Multiprocessor organization is the predominant architecture of modern supercomputers and mini-supercomputers. Notable models of multiprocessor supercomputers[1] include the Cray X-MP, Cray 2, Convex C-240, Cedar and the NEC SX-2 [1] - [6]. Typically, each processor in a multiprocessor system has its own functional pipelines and control unit. All processors may share a single large memory in addition to having their own local memories. On the other hand, processors may have only local memory and rely on an interconnection network for interprocessor communication. The former is referred to as a *shared-memory* system, while the latter as a *distributed memory* or *message-passing* system. Cray X-MP, Cray 2, C-240, SX-2 and Cedar are all shared-memory multiprocessors, while the Intel iPSC, iPSC/2 and the Cosmic Cube are distributed memory systems [7], [8]. This work concentrates on shared-memory systems, although some of the concepts are directly applicable to distributed memory systems.

[1]In this thesis, the terms multiprocessor systems and multiprocessors are used interchangeably.

The next chapter presents some of the issues involved in using a multiprocessor system. As parallel programming is a complex process, vendors often provide software support in the form of run-time libraries and compilers which restructure a serial programs to generate code that can be executed efficiently on a multiprocessor. Chapter 2 discusses the issues of parallel programming. Chapter 3 gives a critical review of the multitasking and microtasking supports available in commercial supercomputers and mini-supercomputers. Chapter 4 examines the microtasking support available on Cedar, a hierarchical shared-memory supercomputer which is being implemented at CSRD. Chapter 5 describes the components of an auto-scheduling compiler and chapter 6 describes how the microtasking support of Cedar may be used to implement an auto-scheduling compiler. Chapter 7 outlines some of the implementation aspects of an auto-scheduling compiler and various implementation tradeoffs.

Chapter 2

ISSUES IN MULTIPROCESSING

This chapter covers some of the issues that are involved in using a multiprocessor system. We assume that the user is familiar with data dependence analysis [9] - [13]. The issues include multitasking paradigms, tradeoffs with respect to the granularity of parallel tasks, load balancing in multiprocessors, and run-time overhead required to synchronize independent processes.

2.1 Multitasking Paradigms

In this section, we provide a brief overview of how multiprocessing is accomplished in various multiprocessor machines. In order to utilize the resources of a multiprocessor system efficiently and achieve high speedup, a serial program must be partitioned into a number of independent tasks which can be executed simultaneously on a number of processors. An executable user program is called a *job*, and a *task* is a part of a job that can be executed by a process (when the task is serial task) or a set of processes (when the task is a parallel task). The process of partitioning a job into a set of tasks that can be executed by a number of cooperating processes is called *multitasking.* A process is an entity which can be scheduled and created by the operating system, and it can acquire processors and memory maps from the operating system [14][1].

In most systems, *parent-child* or *master-slave* relationships are maintained between processes of the same job: a parent process creates (or *forks*) child processes. One way to have P concurrently executing processes is by having a parent process create the first child process, which in turn becomes the parent process for the second child process, and so on until $P-1$ child processes are created [15]. Another way is for the parent process to create all $P-1$ child processes. Each of the P processes has a unique *process identifier (pid)*, which could range from 0 to $P-1$. The process identifier can be used to distribute work among the processes. For example, in the case of a parallel loop with NP iterations, process *pid* may execute iterations $pid * (N-1)+1$ to $pid * N$.

When P independent processes are concurrently executing the same job, we say that there P *threads of execution* are created for that job. Typically, the parent process passes to its child a subroutine name and an argument list. Once the child process is ready (after making itself known to the operating system and acquiring stack memory), it starts to work on the given subroutine. Usually, after the parent process executes its share of the parallel work, it waits for all the spawned processes to complete execution. When all of

[1] A processor is a physical chip that is capable of performing computation, while a process is a software entity that is manipulated by the operating system. A process contains information such as virtual memory maps for the process, whether it is being executed by a processor, and if so, the contents for the registers in the processor.

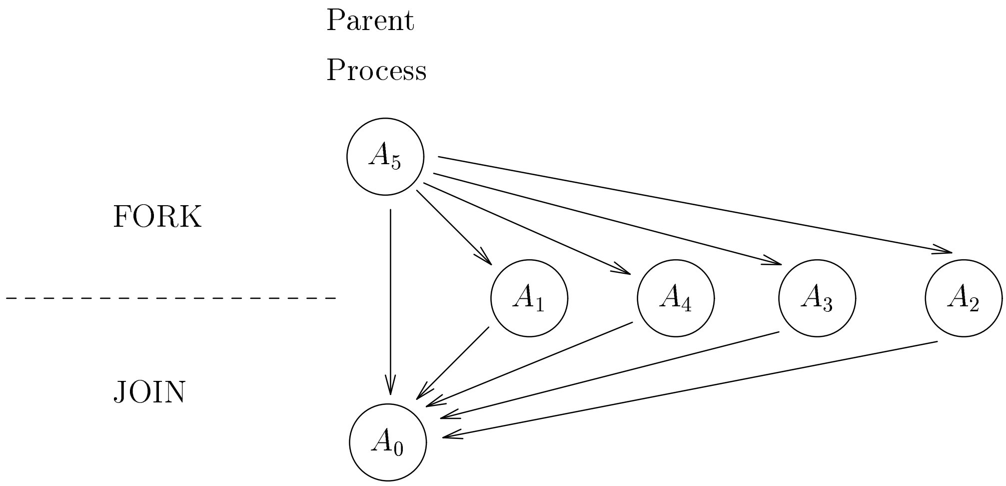

Figure 2.1: A simple fork-join example.

the computation has been completed, the child processes die. The parent process then executes some “clean-up” code, such as saving the computed result into a file and then exiting. This combination of various threads of execution back into one is called a *join*.

For our purpose, we assume that a job is broken down into a *task graph* [16] by a compiler. A task graph is an acyclic directed graph where nodes denote tasks, and arcs denote *task precedence relationships*: an arc from a node A to a node B in the task graph indicates that task B may only be executed after task A has been executed to completion. Tasks represent work that can be assigned to processes.

Figure 2.1 shows an example of the task graph which consists of a parent process that creates four other processes. Each task works on subroutine A with different loop indices. When the subroutine completes, the child processes cease to exist and the parent process is free to proceed.

```
if (fork() == 0) {
        TASK A
        }
else {
        TASK B
}
```

Figure 2.2: Tasks A and B are executed by the child and parent processes respectively.

2.1.1 Run-time library

Process creation and inter-process communication needed in multitasking are accomplished by calls to the operating system. For example, in Unix, the `fork` system call [17] creates a child process. `fork` returns a zero value for the child process and a non-zero value to the parent process. Using the value returned by `fork` as a switch, the parent and child processes are able to execute different tasks. The code segment in Figure 2.2 directs the child process to execute task A and the parent process task B.

Process creation in the context of parallel execution on a multiprocessor is considerably more complicated. The user must take care of the sharing of data between processes, the initialization of the stack space of the child processes, process synchronization, and management of task assignments to processes. To free the user of these intricacies, a *run-time library* is provided. Library routines provide multitasking primitives at a higher, more abstract level and free the user to concentrate more on programming the parallel algorithm. Examples of library routines for process creation include `TSKSTART` of Cray [15] and `ctskstart` of Cedar [18], both of which are used for process creation and will be discussed in the following chapters.

In some systems, several levels of the run-time library are provided. For example, Cray provides the macrotasking and microtasking run-time libraries [19], [15]. The former contains the basic primitives for multitasking such as process creation and management, and synchronization packages, and the latter provides light-weight routines used for the parallelization of DO loops and concurrent subroutines. These will be described briefly in the next chapter.

2.1.2 Barrier synchronization

Although simplified by the provision of the run-time library, process creation and interprocess communication are expensive operations. In a task graph that forks and joins several times, the overhead of many fork-join operations could render multitasking ineffective. As an alternative, instead of allowing the child processes to die at each join, the child processes may busy wait until the next phase of parallel execution.

Barriers are ideally suited for this purpose [20] - [24]. A barrier B is essentially a hardware or software "reporting station." B is a shared counter that keeps track of the number of processes that have "reported," and it must be initialized to be the number of processes involved in the barrier.

The operation of a barrier can be viewed as a subroutine call to `REPORT` which accepts one parameter, B, the barrier. Initially, $B = P$. Then any call to `REPORT` does not return until B has counted up to P. The Pth call to `REPORT` allows all previous $P - 1$ calls to

```
⋮
PARALLEL CODE involving P processes
if (my_id ≠ parent_process_id)
         Report(B) ---------------------- (1)
else {
         SERIAL CODE
         REPORT(B) ---------------------- (2)
}
PARALLEL CODE involving P processes
⋮
```

Figure 2.3: Simple example of barrier synchronization.

`REPORT` to return as well. This is also known as the *clearing* of the barrier, as processes are free to proceed past this point.

Figure 2.3 gives an example of how a join operation is followed by a fork, with serial code being executed in between. After executing the parallel code, each process its own identifier to see if it is the parent process; if so, it proceed to execute the serial code, while all other processes report to B at (1). Note that barrier B will be counting up to $P-1$. As the parent process will report only after the serial code has been executed, all the child processes are blocked at (1). Once the parent completes execution of the serial code, it reports at (2), in effect clearing the barrier, and allowing all P processes to execute the next segment of parallel code.

The above is a very simplified description of how a barrier works. A deficiency in the above example is how the barrier may be initialized to count to P, which is rarely known at compile-time, and even then is usually subject to change at run-time due to process suspensions and preemptions by the operating system. In addition, during phases when the serial code segments are being executed, the system is heavily underutilized. Thus the tradeoff for using barriers is reduced run-time overhead (because child processes are re-used) for a lower system utilization. In Chapter 3, we discuss how an auto-scheduling compiler, by virtue of its implementation, eliminates the need for barrier synchronization of the above kind.

2.2 Levels of Parallelism

Early restructuring compilers focused on transforming serial programs for on vector processors. However, the present generation of restructing compilers such as Parafrase-2 and PTRAN [9] [10] has an additional goal: to extract as much of the parallelism inherent in the code as possible and generate efficient parallel code. This process of partitioning user programs into several threads of execution is often referred to as *multitasking*. The multitasking process can take place either through redesigning the program at the algorithmic level or through the use of an intelligent restructuring compiler. While a redesign of the program by hand usually yields the most efficient program (in term of the *speedup*[2] obtained), it requires much time, effort, and expertize.

Five levels of parallelism can be identified by a compiler [15]:

1. **Instruction or Statement Level.** At the instruction level, some sequences of instructions can be executed in parallel. If the processors have multiple instruction

[2]The speedup of a program is the ratio of the execution time of the program on one processor, T_1, to the execution time of the program on P processors, T_P, i.e., $\frac{T_1}{T_P}$.

execution units or multiple functional units, a compiler can pack several instructions together. This is the technique used in Very Long Instruction Word (VLIW) machines such as the Multiflow Trace-300 computer [25], [26]. On the other hand, the compiler can rely on the hardware to satisfy data dependence constraints at run-time; this is the approach taken by dataflow computers [4].

2. **Loop Level.** Loops that have no *loop-carried dependencies* [11][3] can be transformed into DOALL loops [18][4] and therefore be executed in parallel. DOACROSS loops [27][5] can also be parallelized if the compiler is capable of inserting appropriate synchronization instructions. Nested DOALLs can be transformed such that the outer loop executes in parallel while the inner loop is vectorized for multiprocessors with pipelines [28].

3. **Subroutine Level.** Careful analysis of the call graph and interprocedural analysis help to identify subroutines that may execute in parallel [29], [9]. The importance of interprocedural analysis can be illustrated in the following examples:

```
       DO 100 I = 1 TO 100
100           CALL F(I)
       CALL MAX(A)
       CALL FFT(B)
```

[3]In this thesis, we have assumed that the reader is sufficiently familiar with dependence analysis [9].
[4]A DOALL loop is a DO loop where all the iterations of the loop can be executed in any order.
[5]A DOACROSS loop is a DO loop where iteration $i + 1$ can be executed following a delay of d after iteration i, has started.

```
DO 10 I = 1 TO 100
        A[I]     = C[I] + D[I + 5]
        A[I + 5] = C[I - 2] - D[I]
10 CONTINUE
DO 20 I = 1 TO 100
        B[I + 4] = E[I * 2]
        B[I]     = B[I - 5]
20 CONTINUE
DO 30 I = 1 TO 100
        C[I + 4] = A[I] * B[I]
30 CONTINUE
```

Figure 2.4: Task level parallelism.

If calls to the subroutine F are found to have no loop carried data dependencies, then each iteration of the loop may be executed in parallel. In addition, calls to the subroutines `MAX` and `FFT` can be executed in parallel — perhaps using the COBEGIN-COEND construct [14] — as they operate on different data.

4. **Task Level.** The compiler can generate code so that disjoint pieces of code execute in parallel. For example, loop 10 and loop 20 of Figure 2.4 operate on different data sets. The loops can therefore be executed on two different processors, so long as loop 30 is executed only after both loops 10 and 20 have completed execution.

5. **Job Level.** Job level parallelism is usually exploited by the operating system, assuming that users do not share writable data. Jobs are executed on different physical processors (or even different computer systems) or time-multiplexed on the same set of processors. In distributed database systems where data can be modified by several users at once, the database management system provides specific routines

for users to access the database [30]. These routines help ensure the integrity of the database.

An important concept in parallel programming is the granularity of parallelism. We can roughly classify the different types of parallelism into three classes: *fine grain, medium grain* and *coarse grain.* Fine grain parallelism results from the parallel execution of different statements of the program, or different operations of the same statement on different processors or functional units. Medium grain parallelism exploits parallelism in tasks of the size of a *basic block* or small loops. Coarse grain parallelism is the result of the parallel execution of subroutines, or large parallel loops.

2.3 Run-Time Overhead

A multitasked job relies on the operating system for various functions, such as process creation, scheduling, and synchronization. Included in the overhead of process creation is the delay for the manipulation of the data structures which keeps track of what processes exist and what state they are in, the setting up of virtual memory maps and the acquisition of physical memory, and the initialization of various hardware registers, such as the stack pointer and base registers [14], [31]. This overhead must be justified by the time savings from breaking up the work into smaller parts to be executed in parallel. The greater the overhead, the greater the savings from multitasking must be in order for any speedup to be gained. Hence, parallelism of small granularity can only be exploited when the overhead of concurrentization is sufficiently small.

Consider a system where the overhead is small enough for loop level parallelism to be exploited efficiently, e.g., 0.5 time units. A job partitioned into 1000 independent tasks, each requiring an execution time of 1 time unit, is to be executed on this system. If the job acquires 1000 processors, assuming that they start at the same time, then the job will complete in 1.5 time units. If, however, only 500 processors are available, then each processor will have to execute 2 tasks and incur twice the run-time overhead. In this case, the job will complete in 3 time units. Yet if the job had been partitioned into 500 tasks in the first place, each of 2 time units, the completion time of the job would be 2.5 time units. In this example, the optimal partition of the job is directly dependent on the number of processors the job has acquired at run-time; this information is unfortunately unavailable at compile-time in a multiprogramming environment.

The example, albeit a simple one, illustrates an important point: the level of parallelism that can be efficiently exploited is usually best decided at run-time. The compiler should therefore generate code such that the granularity of disjoint code segments is dependent on run-time conditions, subject to minimum or maximum task size which is computed by the compiler.

2.4 **Load Balance**

Consider a DOALL loop with N iterations, to be executed on P processors. Various policies have been proposed for DOALL loop scheduling. They can be broadly classified into two categories: *static* and *dynamic*. Static policies compute the assignment of iterations to processors at compile-time, while the assignments of processors are decided at run-time for dynamic loop scheduling. We shall discuss three dynamic scheduling policies: chunk scheduling, dynamic scheduling (or self-scheduling), and guided self scheduling (GSS) [32].

Each time a processor obtains a chunk of iterations according to one of the three scheduling algorithms, a "dispatching" is said to have occurred. Chunk scheduling dispatches a maximum of $k = \lceil N/P \rceil$ iterations to each processor. Processor 1 executes iteration 1 to k, processor 2 from $k+1$ to $2k$ $\ldots$, processor P from $(P-1)k+1$ to N. Dynamic scheduling dispatches exactly one iteration each time, but there would be N dispatches as compared to P dispatches for static scheduling. Consequently, dynamic scheduling incurs run-time overhead of the order $O(N)$.

Since loop iterations have, in general, different execution times (e.g., due to branches in the loop body), it is possible for one processor to execute most of the iterations that have the longer execution times. That particular processor would complete execution long after all the other processors have terminated. As the completion time for the entire loop is determined by the time when all the iterations have been executed, it is desirable to "transfer" some of the workload of the late processor to one of the processors that finished earlier. Hence, the tradeoff in this case is the better load balance of dynamic scheduling against the lower run-time overhead of chunk scheduling.

Guided self-scheduling attempts to benefit from the load balancing of dynamic scheduling while minimizing the run-time overhead by assigning $\lceil R/P \rceil$ at each dispatch, where R is the number of remaining iterations. GSS dispatches $x_0 = \lceil N/P \rceil$ in its very first dispatch. The number of iterations dispatched decreases from x_0 and eventually reaches one. In an ideal situation where all processors start at the same time, zero dispatching overhead and constant loop body execution time, each processor ends up having $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ iterations. Guided self-scheduling incurs run-time overhead in the order $O(\log N)$.

2.5 Other Issues

There are many other factors influencing the effectiveness of multitasking environments. An efficient concurrentizing compiler must consider all the factors carefully in deciding how the parallelism in a program is to be utilized. The following contain some of the more important factors, and is by no means an exhaustive list:

1. **Critical Region.** In order to ensure *mutual exclusion*, a processor secures a *lock* before it enters the critical region. Locks are guaranteed, by hardware or software means, to be in the possession of at most one processor at any one time. Other processors that request the lock wait in a queue until the lock is released. If critical regions are too long, each processor holds the lock for a long time, causing more processors to wait. On the other hand, if critical regions are too small, the number of critical regions increases, and therefore more time is spent securing locks.

2. **Synchronization.** Synchronization is needed when processes must communicate to ensure the correct execution of a program [4], [14], [20], [21], [33]. For example, the enforcement of a critical region is a form of synchronization, as the processes have to ensure that only one of them enters the critical region at any time. The use of barriers in a program is another form of synchronization. Similarly, if each task in a task graph is executed by a separate process, enforcement of task precedence relationship is also a form of synchronization. Various primitives have been devised for different synchronization purposes, including semaphores, test-and-set, fetch-and-add, increment/decrement, compare-and-swap, and barriers.

3. **Deadlock.** When synchronization instructions are not properly used, concurrent processes may enter in a deadlock [14]. A simple example where deadlock occurs is when process 1 holds lock A while waiting for lock B, and process 2 holds lock B while waiting for lock A; neither process will be able to proceed. The four necessary conditions for deadlock are mutual exclusion, wait for, no preemption, and circular wait [14]. A concurrentizing compiler must take care to generate code that does not deadlock at run-time by ensuring that one of the four conditions will never happen.

Finally, we note that a multitasking compiler is dissimilar to a vectorizing compiler, although they employ the same principles and techniques in many cases. First, because a multiprocessor system usually incorporates functional pipelines for each processor, a concurrentizing compiler would partition tasks such that all the pipelines in the system can be utilized. While a vectorizing compiler seeks vectorizable inner loops, a concurrentizing compiler seeks vectorizable inner loops that are nested within a parallel loop. In addition, a concurrentizing compiler may insert synchronization instructions inside a DOACROSS loop to convert it to a DOALL loop. As such, the code transformation process would be different. Second, vectorization is usually robust to arithmetic round-off errors while multitasking may not be. Third, while the overhead for employing a pipeline is generally fixed for any vector length, the overhead of multitasking increases with the number of parallel threads that are created. Furthermore, the overhead of forking a separate thread is generally higher than that for starting a pipeline, as expensive calls to the operating system are necessary.

The aforementioned factors are all important contributors to the effectiveness of multitasking in multiprocessor systems, and a concurrentizing compiler must weigh the trade-offs involved in all important cases. This thesis concentrates mainly on compiler-assisted task scheduling based on microtasking support, provided in the form of a run-time library. However, a task in our case may consist of a parallel loop, a serial code segment, or a subroutine that can be concurrently executed.

Chapter 3

OVERVIEW OF MULTITASKING SUPPORT

In this chapter, we present an overview of some of the software and hardware support for multitasking in certain multiprocessors, namely, the Cray series, the Alliant FX-8, and the Sequent computer systems.

3.1 Cray Multitasking Support

Cray multitasking support may be broadly classified into three classes: basic synchronization and task manipulation primitives, microtasking support built on the basic primitives, and autotasking compiler which automatically restructures a user program to run on the microtasking library.

Synchronization primitives are provided in the form of *locks* and *events* [15]. As the name implies, locks are meant to be used to enforce mutual exclusion at critical sections in the user program. A lock is acquired by a call to `lockon`. If several processes call `lockon`, only one will return, allowing the corresponding process to enter the critical section. Other processes have to wait in a queue until the lock is released (via a call to `lockrel`).

Events are used to keep processes informed on what part of the computation has been completed. A call to `evclear` initializes an event so that all subsequent calls to `evwait` forces processes to wait until some other task calls `evpost` to clear the event. Subsequent calls to `evwait` will no longer force processes to wait, until another `evclear` is called to re-initialize the event.

Process creation is facilitated by the `tskstart` call with three sets of parameters [15]: the first is a *task control array* consisting of two or three integers storing the name of the process, the second is a subroutine name for the newly created process to work on, and the third is a list of arguments for the subroutine. `tskstart` returns as soon as the child process has been created and inserted into the ready queue of the operating system. The parent process may eventually call the `tskwait` routine, which returns only when the child process is done.

The scheduling of the child process is left entirely to the operating system and therefore a child process may not be executing even when `tskstart` returns in the parent process. For this reason, the list of arguments for the subroutine of the child process cannot be registers, expressions and local variables, as they may be overwritten by the parent process before the child process reads them. On the other hand, the user can synchronize the two and force the parent process to wait until the child process has read all its parameters. Otherwise, subroutine parameters can be either global variables (in a COMMON block) or *process common* variables that are created by the `task common` construct. A process common variable is allocated on the stack of the child process so that each child process has its own private copy of the variable. An example of a process common variable is the loop index of a parallel loop.

Cray refers to the above basic primitives as *macrotasking* routines, and they are intended for very coarse parallelism. Use of macrotasking is done manually by inserting calls in the user program. Deadlocks may arise when locks are used to enforce mutual exclusions, and deadlock prevention is again left to the user. The user must also perform a dependence analysis of the program and decide on the scope of the variables. In short, macrotasking routines can only be used if the user is willing to re-write parts of the program. The help that a compiler provides here is merely linking to the appropriate macrotasking subroutines at compile-time.

3.1.1 Microtasking

For more efficient exploitation of loop level parallelism, Cray provides *microtasking directives* [19] that are translated into the appropriate calls to the microtasking library by a preprocessor. We provide a few examples of microtasking directives to illustrate its use.

The (`guard, end guard`) directive delimits a critical region, thereby sparing the user of manipulating locks. A `do global` microtasking directive defines a DOALL loop. The

```
PROCESS
         CALL A
ALSO PROCESS
         CALL B
ALSO PROCESS
         CALL C
END PROCESS
```

Figure 3.1: Cray microtasking example for the creation of multiple threads of execution. This code segment has three independent threads of execution, executing the subroutines A, B and C.

(`process, also process, end process`) construct is provided for fork and join operations. For example, in Figure 3.1, A, B and C are independent threads which will execute in parallel. Each thread is executed by a process. Idle processes that are not assigned a thread busy wait. When all threads have completed execution, execution resumes *after* the `end process` directive.

Microtasking directives are again user supplied. Each directive is preceded by CMIC$, so they are ignored by the Fortran compiler. As with macrotasking, determination of the scope of variables is again left entirely to the user, who is also required to detect parallel loops. Hence, although the microtasking directives makes parallel processing possible, extensive code analysis is still required, and manual code modification is still necessary.

3.1.2 Autotasking

Currently, Cray supplies a new autotasking compiler which provides automatic parallelization (or concurrentization) in addition to automatic vectorization of programs. The autotasking compiler performs numerous functions [28], [34]:

- simple in-line expansion of small subroutines,
- parallelization of loops with reduction variables,
- recognization of certain common matrix operations and substitution of the code with a call to an optimized routine,
- parallelization of loops using several loop scheduling mechanisms, including Guided Self-Scheduling,
- automatically tagging variables used in the body of a parallelized loop as shared or private variables,
- generating multiversion loops for vectorization or serial execution, along with run-time tests,
- generation of run-time tests to check if parallel loops contain sufficient iterations to warrant parallel execution,
- generation of run-time tests to check if certain conditions hold at run-time so that loops can be parallelized,

- conversion of simple IF-GOTO loops to do loops where possible.

Although the autotasking compiler is capable of automatic parallelization of serial programs, there is a restriction on the extent of parallelization that it can perform. Only a single parallel loops are allowed in any loop nest. This restriction allows only the following two possibilities in any nest of loops:

1. A vector loop.

2. An outer parallel and inner vector loop.

In addition, interprocedural analysis is not performed by the autotasking compiler to extract subroutine level parallelism. Parameter aliasing is investigated only to the extent of checking if arrays have been equivalenced together.

3.1.3 Comments

Of the three types of multitasking support that Cray provides, autotasking is the most powerful. The use of the two earlier tools requires the user to manually declare the scope of variables and perform extensive dependence analysis on the program. However, autotasking allows only one level of parallelization, which may not be optimal for certain programs. For example, two DOALL loops that can be executed concurrently will be executed one after the other. Therefore, there is implicitly a barrier synchronization at the end of each parallel loop.

3.2 Alliant Multitasking

The Alliant FX/8 consists of 8 computational elements (CE). All eight CEs are connected to a shared cache via a cross-bar switch. Each CE contains eight 4-bit synchronization registers (cs0-cs7) [35]. Instructions are available to manipulate these registers so that the CEs can be synchronized when a DOACROSS loop is being executed.

For example, assume that cs0 is the chosen synchronization register. Each CE waits until the three lower order bits of cs0 matches the lower order three bits of the iteration number that it is executing, at which point the fourth bit of its own cs0 is set and and all other cs0 are incremented by one, which in turn allows another CE to advance. In this way, synchronization between successive iterations is achieved entirely through the hardware, with the help of a powerful instruction set.

3.3 Sequent Multitasking Library

Sequent provides multitasking support for at least three languages, Pascal, C and Fortran. The Sequent multitasking library [36] incorporates many synchronization primitives, including barriers, test-and-set, semaphores, and locks. Process creation is facilitated by means of the (unix-like) m_fork system calls, which create child processes that share the memory mappings of the parent process. Loop concurrency is available in the form of a generalized DOACROSS compiler directive, while a DOALL is simply a DOACROSS without any synchronization.

As with Cray macrotasking and microtasking, variables must be explicitly declared as shared or private in the program. Very little automatic parallelization of the source program is provided. Therefore, the user must perform dependence analysis and interprocedural analysis manually.

Chapter 4

MULTITASKING SUPPORT ON THE CEDAR

In this chapter, we describe the multitasking support available on the Cedar multiprocessor. Cedar is a research machine being developed at the Center for Supercomputing Research and Development of the University of Illinois at Urbana-Champaign. Xylem, its operating system, is based on the Alliant's Concentrix operating system [37]. Cedar microtasking support is provided in the form of run-time library routines that construct and maintain **mtask_queue**, a queue of user tasks. Tasks on **mtask_queue** can be executed in any order, subject only to the ordering imposed by the priority of the tasks.

4.1 The Architecture of Cedar

The architecture of Cedar is depicted in Figure 4.1. Cedar is a hierarchical shared-memory machine in that processors are organized into *clusters* [1] - [3]. A cluster is essentially an Alliant FX/80 processor cluster. Processors (*computational elements (CE)* in Alliant terminology) within a cluster share a *cluster memory (CM)* but do not have

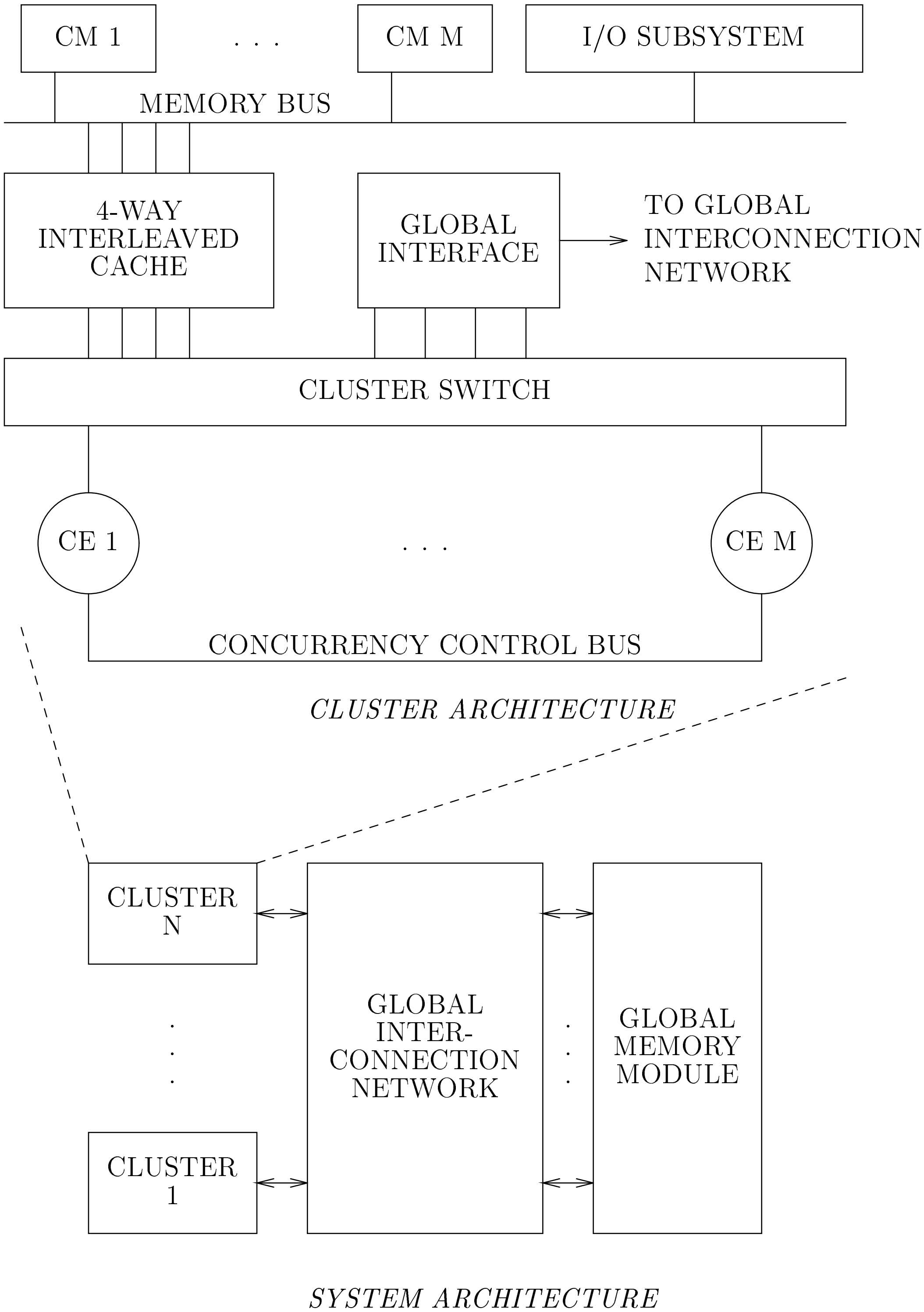

Figure 4.1: Cedar consists of clusters of CEs connected to a global shared memory. Each cluster is essentially a modified Alliant FX/8.

`CDOACROSS` | `SDOALL` | `CDOALL` *label*`[,]` $i = e_1, e_2[, e_3]$
    `[` *type statements* `]` `(1)`
    `[` *statements* `]` `(2)`
`[LOOP]`
    `[` *statements* `]`
`label` `statement` | `END CDOACROSS`

Figure 4.2: Syntax of parallel loops in Cedar Fortran. e_1, e_2, e_3 are integer expressions, and i is the name of an integer variable.

private local memory. Several clusters are connected to a global memory via an Omega interconnection network. Two levels of memory are therefore available to individual processors in the Cedar: global memory and cluster memory.

4.2 Cedar Fortran

Various library routines for synchronization and process manipulation can be directly called in Cedar Fortran [18]. Due to the architecture of Cedar, two levels of loop parallelism are available. A DOALL loop may be executed in parallel within a cluster (one iteration per CE) or it may be executed in parallel across clusters (one iteration per cluster). The former is called CDOALL for cluster_DOALL and the later SDOALL for spread_DOALL. A CDOACROSS loop is a DOACROSS loop that is executed on a cluster. The syntax of the three parallel loops is shown in Figure 4.2.

The statements in (1) declare the scope of the local variables used within the loop body, and the statements in (2) initialize them. This part of the parallel loop — the declaration and initialization of the local variables — is called the PREAMBLE of the loop.

The PREAMBLE is executed once by each helper process that participates in the execution of the SDOALL. The loop body of the parallel loop may only reference GLOBAL variables or local variables declared in its PREAMBLE. This restriction simplifies the implementation of parallel loops.

Process creation is facilitated by the **ctskstart** routine. Each call to **ctskstart** creates a heavy-weight *helper process*, which executes on a cluster and is known to and scheduled by Xylem. The parameters for **ctskstart** consist of a subroutine name and the corresponding list of parameters, and a number to specify the number of computational elements that will be allocated to the new process for the execution of CDOALL and CDOACROSS loops within the subroutine.

For synchronization, three sets of functions are provided. We shall describe briefly the simplest set of functions, as this is the only set used in this paper. This set of functions has the format **ifetch_and_op**, where **op** may be addition, subtraction or assignment, and the i stands for integer. These are *atomic* functions, meaning that they fetch the value of an integer i, perform the required operation on i and write the new value back into memory in one request to the global memory. The old value of i is the value returned to the process calling **ifetch_and_op**. Obviously, i must be a variable residing in global memory. The atomicity of the functions is ensured by the hardware of the global memory network.

4.3 Cedar Microtasking Library

The microtasking library uses helper processes which are provided by the operating system (Xylem). Within a cluster, CDOALLs are scheduled efficiently using the Alliant hardware scheduling mechanism [38]. Helper processes are expected to run-to-completion — in the sense that once a process has been assigned a cluster, the cluster will not be relinquished until the execution of the task is complete.

Initially, a parent process executes the user program. Any number of helper processes can be created by the parent process to help in the execution of the user program. Process creation is accomplished by calling the **ctskstart** routine. The first child process creates the rest of the helpers (if any) so that the parent process is free to continue execution.

mtask_queue is a data structure defined in the microtasking library. **mtask_queue** is essentially a queue of user tasks is stored in global memory so that it is accessible to all helper processes. Each entry in **mtask_queue** contains various fields storing information pertaining to the execution of the queue entry at run-time. Relevant fields for this discussion are discussed below:

- **entry_type** specifies whether this queue entry is a parallel or serial queue entry,
- **entry** contains the address of the entry point of the corresponding task,
- **stack_info** points to the area holding the stack image for this queue entry — helper processes working on this queue entry set up their stack space using this image,
- **loop_index** points to a loop index allocated in global memory,

- **end** and **stride** stores the upper bound and stride of a parallel loop,
- **active_helpers** contains the number of helper processes working on this queue entry,
- **num_args** and **arg_sizes** contain the size and address of the parameters of subroutines that are queued as serial tasks on **mtask_queue**.

Once created, and whenever it has acquired a cluster (as dictated by the time-slice allotted by Xylem), each helper process spin waits on **mtask_queue**, checking for work to do. At this point, a helper process has no knowledge of what the user program looks like; it will only execute user tasks as soon as one is placed in the **mtask_queue**.

4.3.1 Serial tasks

Two kinds of user tasks can be placed on **mtask_queue**, serial tasks or parallel tasks. Parallel tasks corresponds to SDOALL loops, while serial tasks correspond to those queued via a call to **mtskstart** in the user program. Such serial tasks are also called *user-defined ready tasks (URT).*

The syntax of the call to **mtskstart** is as follows:

```
mtskstart(sub, priority, [argument list])
```

where `sub` is the name of a subroutine, and `priority` is the priority of the queue entry. The above is translated by the Cedar Fortran compiler to

```
mtskstart$_(sub, priority, num_args, [argument1, size1, ...])
```

where `num_args` contains the number of arguments to subroutine `sub`. **mtskstart$_** is part of the Cedar microtasking library, and its function is to queue the subroutine onto **mtask_queue**. The first action of **mtskstart$_** is to store the parameters of `sub` in global memory (using the **stack_info** field in the **mtask_queue** queue entry). The size of the arguments is also stored in another array (using the arg_sizes field), which is also allocated in global memory. These activities effectively set up a *template* for the stack of a helper process executing the subroutine. Hence, once the queue entry has been set up and queued on **mtask_queue**, any helper process may work on the serial task by copying the stack template, and then jumping to the entry point of the subroutine.

Only by-value parameter passing is allowed in URTs. This restriction simplifies the implementation of the microtasking library, as URTs may be executed independently. The passing of an array into a subroutine to be queued as an URT is therefore extremely expensive, since the queueing process involves copying the entire array into global memory.

4.3.2 Parallel tasks

A helper process that enqueues an SDOALL is called the *parent process of the SDOALL loop.* The parent process of an SDOALL loop also has the responsibility of dequeuing the corresponding queue entry. By default, it begins to execute the SDOALL loop as soon as the queueing operation is completed. When the parent process of an

SDOALL completes execution, and there are no more iterations to execute, it must wait until all helper processes assisting in the execution of the loop complete execution.

An SDOALL queue entry is queued via a call to **mtask_queue**:

```
mtask_queue(queue_index, loop_index, end, stride)
```

where `queue_index` returns a value indicating whether the queueing of the parallel task is successful, `loop_index, end` and `stride` are the index, upper bound and stride of the parallel loop respectively. The loop index must be allocated in the global memory.

While an SDOALL remains in **mtask_queue**, idle helpers will try to execute iterations of the SDOALL. The template of an SDOALL contains information on what local variables need to be set up (recall that an SDOALL may have local variables declared and initialized in its PREAMBLE). This template is allocated in global memory and the address is stored in the stack_info field of its **mtask_queue** entry.

4.3.3 Activities of idle helper processes

Insertion of queue entries onto **mtask_queue** relies on the priority of the new work. SDOALLs have the same priority by default, while the priorities of URTs are user supplied. The **active_helper** field in each queue entry is used to keep track of the number of helper processes working on the queue entry. If the **active_helper** field of a serial task is already one, this queue entry will not be executed by any other helper process. The parent process of an SDOALL must wait for the **active_helper** to become zero before proceeding to dequeue its entry.

After executing a URT, a helper process removes the queue entry and spin waits on **mtask_queue** for more work. On the other hand, after executing an iteration of an SDOALL, a helper process increments (or decrements) the loop index, and checks if it is still in range. This part of the code in which a helper process increments the loop index and checks if it is within range is generated by the Cedar Fortran compiler. If so, the next iteration is executed. Otherwise, the **active_helper** field of the SDOALL queue entry is decremented. The helper process then checks if it is the parent process of the SDOALL. The parent process of the SDOALL waits until the **active_helper** field is decremented to zero and then proceeds to dequeue its entry. Other helper processes resume busy waiting on **mtask_queue** for more work.

In summary, the microtasking support on Cedar is provided in the form of a queue on which all helper processes spin-wait. Each parallel task can be executed without any calls to Xylem and with minimum interprocess communication. Synchronization is only necessary in the form of serializing accesses to **mtask_queue** when idle helper processes look for work to execute. Additional overhead is paid in the form of parameter and stack copying when an entry is queued. Explicit barrier synchronization is unnecessary as the parent process of an SDOALL always waits for all other helper processes to complete execution.

```
SDOALL 1 i = 1, n
LOOP
         ⋮
         LOOP BODY
         ⋮
1 END SDOALL
```

Figure 4.3: Example of an SDOALL in Cedar Fortran.

4.4 Translation of an SDOALL

We will illustrate the use of the microtasking library by an example of how an SDOALL would be translated [39]. Figure 4.3 shows a simple SDOALL. The translation of the SDOALL into microtasking library calls is shown in Figure 4.4. A parent helper process first executes the main program. When the parent process executes the subroutine call **mtask_queue**, an SDOALL queue entry is placed onto **mtask_queue**. If the queueing operation is successful, the parameter queue_index contains the index of the SDOALL queue entry in **mtask_queue**. **mtask_queue** sets up the information necessary for any helper process to work on the SDOALL, including stack template and entry point.

Any helper process that works on the SDOALL begins execution at the line following the call. The loop index is incremented atomically and checked for bounds. When all iterations have been dispatched, the subroutine **mtask_cleanup** is called, which directs helper processes to perform the following activities. For helper processes, **mtask_cleanup** decrements the active_helpers field and then cleans up the stack space of

```
        loop_index = 1
        end = n
        i = 1
        IF (1.gt.end) goto 99997

        CALL mtask_queue(queue_index, loop_index, end, 1)
        local_index = ifetch_and_add(loop_index, 1)
        IF (local_index.gt.end) GOTO 99995
        ⋮
        PREAMBLE if any exists
        ⋮
99996   CONTINUE

        ⋮
        LOOP BODY
        ⋮
        local_index = ifetch_and_add(loop_index, 1) --- (1)
        IF (local_index.le.end) GOTO 99996
99995   return_value = mtask_cleanup(queue_index) --- (2)
        i = end + 1
99997   CODE AFTER SDOALL
```

Figure 4.4: Code as translated by Cedar Fortran compiler.

the helper process. The helper process can now return to busy waiting on **mtask_queue**. For the parent process of an SDOALL, **mtask_cleanup** returns only after the completion of all other helper processes working on that SDOALL.

The microtasking scheme as implemented presently incurs high overhead from synchronizing accesses to **mtask_queue**. An extended version of the microtasking library which allows only one SDOALL in the queue and any number of user tasks in **mtask_queue** is being implemented. Run-time overhead is reduced with corresponding increase in speedup for the user. This approach, however, is less flexible since high level spreading which is possible in the earlier approach is not possible with this implementation.

Chapter 5

THE AUTO-SCHEDULING COMPILER

This chapter describes the functionality of an Auto-scheduling Compiler (ASC) [16]. The goal of auto-scheduling is to generate efficient parallel code which is self-scheduled at the task and loop iteration levels. An Auto-scheduling Compiler consists of four stages: (1) the vectorizing and parallelizing front-end compiler, (2) the program partitioner, (3) the pre-scheduler, and (4) the drive code generator and optimizer.

5.1 Front-end Compiler

The vectorizing and parallelizing front-end compiler parses the input program and performs dependence analysis [12] to construct the data dependence graph for the program. It also constructs important data structures such as the call graph and the flow graph. Constant propagation, interprocedural analysis, in-line expansion, static program analysis, and various other loop transformations techniques — such as loop distribution, loop vectorization, identification of DOALL or DOACROSS loops from DO loops, loop interchanging, loop spreading and cycle shrinking — can be used by the front-end to restructure the program for parallel execution. An example of such a front-end compiler is the Parafrase-2 compiler [9].

5.2 Partitioner

The data structures maintained by the front-end, such as the parse tree, the dependence graph and the flow graph, are used by the program partitioner to construct an acyclic task graph [40]. Each node in the task graph represents either a serial or a parallel task, and arcs represent task precedence relationships. Assuming that structured programming is used [41], the cycles in the flow graph of the program would be entirely due to DO loops (including DOALL and DOACROSS loops in various Fortran dialects). Furthermore, by automatically converting recursion to iteration [42], the call graph is also made acyclic.

The construction of the task graph (from a source program without GOTOs and recursion) then proceeds as follows. The main routine may be considered as being enclosed by an "imaginary" outermost serial DO loop with only one iteration — this is consistent with dependence analysis for multiply nested loops. Any block of statements without subroutine calls and loops is made into a single task graph node, while loops and subroutine calls become *composite task nodes.* Composite task nodes can be "zoomed in" to reveal another task graph. For example, when an outer loop (or subroutine call) is made into a composite task node, the entire task graph of the inner loop (or subroutine) is represented collectively by the composite task node.

5.2.1 Composite task nodes

A task subgraph representing a composite task node consists of a graph $G_{sub} = (V_{sub}, E_{sub})$, which has a unique start node v_s and a unique end node v_e. All edges from nodes outside G_{sub} to nodes of G_{sub} enter through v_s, and all edges from nodes of G_{sub} to outside nodes exit through v_e. In other words, there are no edges to or from the nodes in the subgraph to the rest of the hierarchical task graph except to the entry and from the exit nodes. Such a graph is called a *hammock* in [43]. A node of a hammock which is neither a start node nor an end node is called an *interior node.*

A composite task node represents either a DO loop or a subroutine. If a composite task node represents a subroutine, then its corresponding hammock represents the task graph for the body of the subroutine. This task graph can be executed serially. When all interior nodes and the exit node of a hammock have been executed, the subroutine returns.

Consider now a composite task node which represents a DO loop. Although its corresponding hammock H is acyclic, the execution of the task nodes of H may follow a cyclic path. In particular, consider the task graph of the doubly nested loop as shown:

```
         DO 100 I = 1 TO N
                  SDOALL J = 1 TO M
                           ⋮
                  END SDOALL
100      CONTINUE
```

The DO statement is made into a composite task node n_{do}. Since the DO body is a single SDOALL loop, it seems that the hammock for n_{do} consists of only one parallel task node. In fact, the hammock for n_{do} is slightly more complicated. When the code is translated by a compiler, the DO statement may be translated to the following (the SDOALL loop and its body are left unchanged):

```
        I = 0;
        I = I + 1;
        IF (I > N) THEN GOTO 998;
999     SDOALL J = 1 TO M
                  ⋮
        END SDOALL
        I = I + 1;
        IF (I > N) THEN GOTO 999;
998     ⋮
```

Although a more efficient translation can be found (using only one statement to increment I and only one IF test), the above translation is closer to the translation in Figure 4.4. The first three statements can be placed in the start node of the hammock, while the last two statements can be placed in the end node. The SDOALL is represented by a comosite task node.

Recall that the run-time library of Cedar accepts serial tasks that are encapsulated in subroutines. For our example above, we may encapsulate the three task nodes in three different subroutines, TMP1, TMP2, and TMP3. In Figure 5.1, I is allocated in global memory. The code for the SDOALL loop is encapsulated in subroutine TMP2. In general, TMP2 represents the loop body of the DO loop. The IF statements surrounding the **mtskstart** calls can be considered the exit blocks of the corresponding tasks.

```
SUBROUTINE TMP1
I = 0
I = I + 1
IF I < N THEN
 mtskstart(tmp2, I)
ELSE
 mtskstart(tmp3)
END IF
```

```
SUBROUTINE TMP2 (I)
INTEGER I
Code for SDOALL
mtskstart(tmp3)
```

```
SUBROUTINE TMP3
I = I + 1
IF I < N THEN
 mtskstart(tmp2, I)
END IF
```

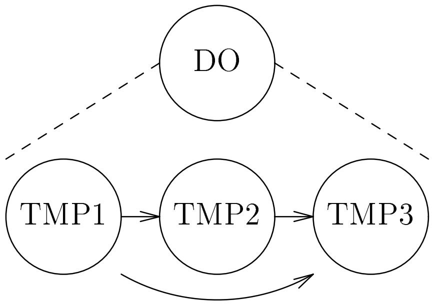

Figure 5.1: Construction of the task graph for an SDOALL loop enclosed by a serial DO loop.

When the DO loop is executed, TMP1 is called first. TMP1 initializes I and then checks if I is less than N. If so, TMP2 is called; otherwise, TMP3 is called. For the latter, as I is already out of range, TMP3 simply terminates (and the exit block of TMP3 is executed). This corresponds to the case where the body of the DO loop is not executed because the loop index falls out of range. In the former case, the loop body of the DO loop — which happens to be an SDOALL loop here, is executed via the call to TMP2. After the SDOALl is executed, TMP3 is called. After I has been incremented, if I is still less than N, TMP3 calls TMP2. Thus there is a control flow cycle between TMP2 and TMP3. Due to the above, there should be a task precedence edge from the task node for TMP3 to TMP2, thereby creating a cycle in the hierarchical task graph.

This simple cycle is inherent in a DO loop, and corresponds to the dispatching of the next iteration. In general, there is a back edge from v_e to each successor node of v_s. We note that the back edge dispatches all task nodes in the hammock except v_s and that the loop index is incremented and checked in v_e. For this reason, we may insert a dummy node v_{dummy} which has only one incoming edge (from v_s) and has outgoing edges to all successors of v_s. The back edge from v_e then goes to v_{dummy}. Therefore, the cycle between v_{dummy} and v_e essentially is repeatedly executed until all the iterations of the DO loop has been dispatched.

The presence of such back edges contradicts the earlier definition of an acyclic hierarchical task graph. However, as tasks outside the DO loop will only be executed when the DO loop is executed to completion, the presence of this cycle does not affect their execution. Hence, for the remainder of the thesis, we shall ignore the presence of this cycle unless stated otherwise.

5.2.2 **Hierarchical task graph**

The result is a *hierarchical task graph* (HTG) representation of the program. Let there be m levels in the hierarchical task graph. At level 1, there is exactly one node, consisting of the entire program. Each *composite task node* at level i, $i = 2 \ldots m-1$, represents a level $i+1$ task subgraph.

Level m task graphs therefore has only simple task nodes. We may recursively zoom in on the composite task nodes of the HTG until eventually there is no more composite

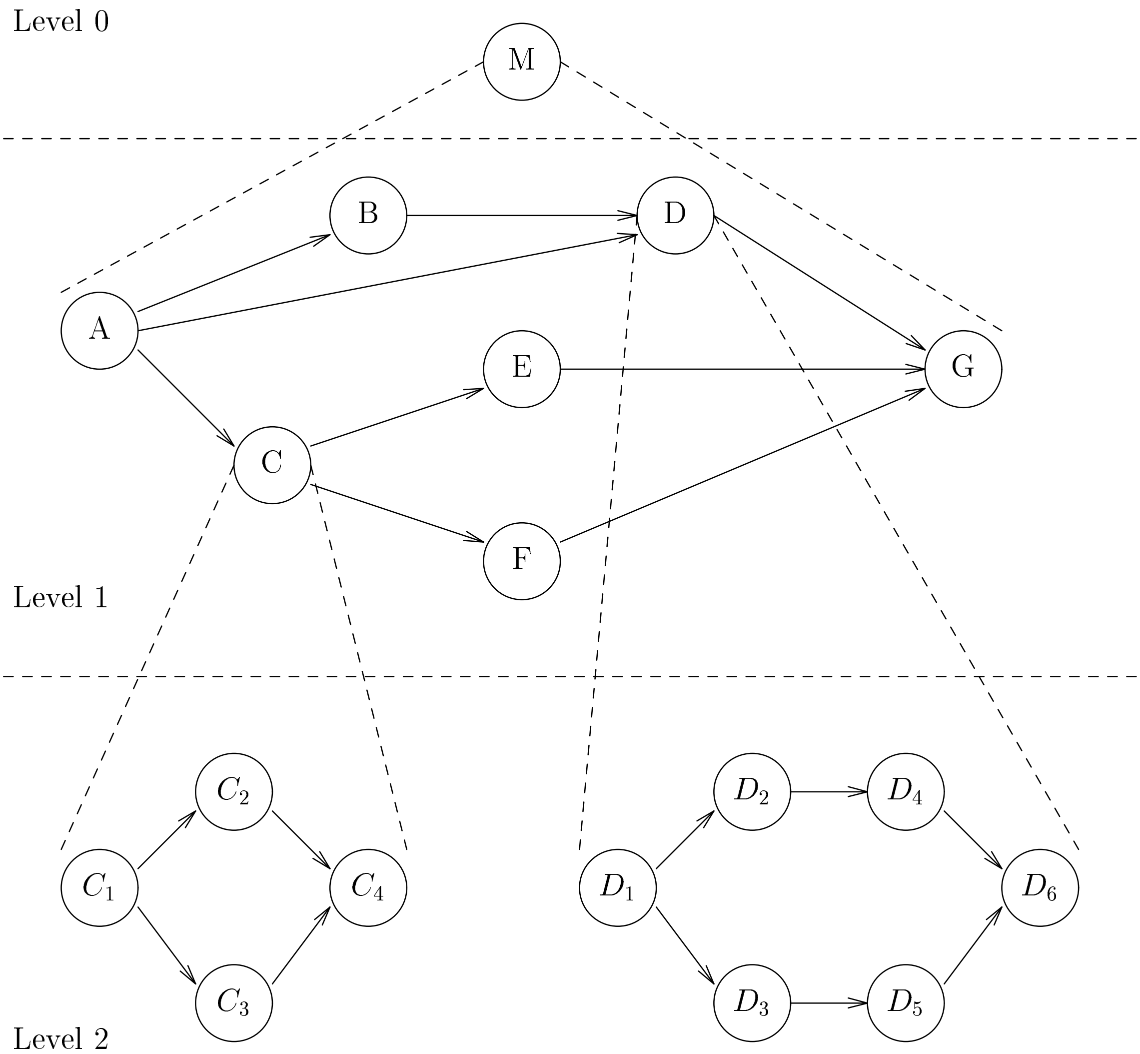

Figure 5.2: Example of a 3-level hierarchical task graph.

task node in the graph; this is the graph which we will informally call the task graph of the program.

Figure 5.2 shows an example of an HTG. At level 1, the program consists of a single node task graph (node M). At level 2, node M is zoomed in to reveal nodes A, B, $\ldots$, G, reflecting the structure of the loop body within the imaginary outermost loop. Composite task nodes in the task graph of level 2 can be further decomposed to reveal two more sets of nodes at level 3, corresponding to the subtask graphs of nodes C and D of level 2. In other words, the level 3 graph actually consists of 15 nodes, namely, $A, B, E, F, G, C_1, C_2, C_3, C_4, D_1, D_2, \ldots, D_6$.

5.3 Pre-Scheduler

The pre-scheduler performs a modified breadth-first search to partition the task graph into levels. We may assume that the task graph has a unique start node and a unique exit node. The *Descend* procedure outlined in [16] is called on each node, starting from the unique exit node bottom-up. The goal of this step is to alleviate any bottleneck effect that serial tasks could pose at run-time. This is done by delaying the execution of parallel tasks as much as possible so that they maximally overlap with the execution of serial tasks at the lower level of the task graph. For more information, the reader can refer to [16].

5.4 Drive Code Generator and Optimizer

The drive code generator and optimizer transforms the task graph so that task scheduling is performed at run-time by the user program. This can be accomplished with the help of a run-time library which maintains a queue of ready tasks where helper processes look for work. For each task, two blocks of code are generated by the drive code generator and optimizer, the *entry block* and the *exit block*.

5.4.1 Removal of transitive edges

Before generating entry and exit blocks, transitive edges in the task graph are removed since these are redundant with respect to self-scheduling. For example, consider the edge from A to D in Figure 5.2. Task D waits for both A and B, but since B must wait for A, the completion of B implies that task A has completed. Hence the edge (A, D) is redundant and can be deleted.

The removal of transitive edges in a graph is equivalent to computing the *transitive reduction* of a graph, and involves a complexity of $O(n^{lg_2 7})$. The algorithm is outlined in [44]. A few definitions are necessary before we proceed.

A *directed graph* $G = (V, E)$ is composed of a set of nodes, $V = \{v_1, v_2, \ldots, v_n\}$ and a set of ordered tuples called *arcs*, $E = \{(v_i, v_j), v_i, v_j \in V\}$. A *directed path* in G from node u to v is a sequence of nodes, $x_0 = u, x_1, x_2, \ldots x_p = v$ satisfying $\alpha_k = (x_k, x_{k+1}) \in E, 0 \leq k \leq p-1$. A *cycle* is a directed path beginning and ending at the same node, that is, a directed path with $u = v$. An *acyclic* graph contains no cycles.

G is *transitive* if, for every (u, v), u, v not necessarily distinct, $(u, v) \in E$ whenever there is a directed path in G from u to v. The *transitive closure* G^T of G is the least subset of $V \times V$ which contains E and makes G transitive. A *transitive reduction* of G is a graph G^t such that $(G^t)^T = G^T$ and every proper subgraph H of G^t satisfies $H^T \neq G^T$. For example, the transitive reduction of the level 1 task graph of Figure 5.2 is obtained when the edge (A, D) is removed, and the transitive closure of the same graph is obtained when we add the following edges: (A, G), (A, E), (A, F), (C, G) and (B, G). It has been shown that the transitive reduction of an acyclic graph is unique [44].

The procedure for computing the transitive reduction of an *acyclic graph* G is as follows [44]:

1. Compute M, the adjacency matrix for G, and M^T, the adjacency matrix for G^T [45], [46].

2. Compute $M' = M * M^T$, where the * denotes boolean matrix multiplication[1]. Let G' be the graph corresponding to the adjacency matrix M'.

3. G^t, the transitive reduction of G, is given by $G^t = G - G'$.

Theorem 5.1 *G' contains the transitive edges of G. Removing the edges contained in G' from G results in the transitive reduction of G.*

Proof. An arc (u, v) is in G^t if and only if $(u, v) \in G$, and there is no path from u to v which does not include (u, v). Such a path exists if and only if there is some w,

[1] A boolean matrix multiplication $A = B * C$, where A, B, C are $n \times n$ matrices with only 1 or 0, is defined to be $[a_{ij}] = \bigvee_{k=1}^{n} b_{ik} c_{kj}$.

$w \neq u, w \neq v$, such that there is an arc (u, w) and a path from w to v in G. The $[u, v]$th element of M' is equivalent to the dot product of the row corresponding to u in M, $[u_i]$, and the column corresponding to v in M^T, $[v_i]$, that is, $[u_i]^T \cdot [v_i]$, which equals 1 if and only if there exist an element, e.g., the wth element, of $[u_i]$ and $[v_i]$ are 1. As G is acyclic, $w \neq u, w \neq v$. Now, the wth element of $[u_i] = 1$ implies that there is an arc from u to w in G. The wth element of $[v_i] = 1$ implies that in the transitive closure of G, there is a path from w to v, which in turns implies that there is a path from w to v in G. But as G is acyclic, this path must not include (u, v). $\square$

5.4.2 **Entry and exit blocks**

When a task T is executed by a helper process Q_h, various registers of Q_h must be set up. For example, the amount of stack space that must be set up for Q_h and where the stack is allocated is part of the context of Q_h. In addition, if Q_h is executing a parallel task (for example, an SDOALL), Q_h must know what loop scheduling policy to use. Above specifications are provided in the entry block of T. Hence, the execution of T begins with the execution of the code contained in the entry block of T, from which the stack space of Q_h is set up. If T is a parallel task, Q_h also obtains information regarding the loop scheduling policy of T by executing the entry block code of T.

To accomplish task level self-scheduling, various functions must be performed at the completion of the execution of a task T. If T is a parallel task, barrier synchronization is required to ensure that task precedence relationships are not violated. One of the helper processes, Q_h, is selected to dequeue T from the user task queue maintained by the run-time library (if T is a serial task, then by default the helper process executing T is selected). Other helper processes simply go to the user task queue for more work, or spin wait if the queue is empty.

The responsibility of Q_h is then to update task precedence relationships and queue the successors of T that have become executable due to T's completion. For example, when task A in Figure 5.2 has completed execution, tasks B and C_1 are free to execute and are both queued onto the user task queue. On the other hand, if only the set of tasks belonging to the composite task node D has completed execution, but not tasks E and F, then task G cannot be queued.

Such updates can be accomplished by a set of *task precedence counters*. Each task in the task graph has a task precedence counter associated with it. The counter for a task T counts the number of predecessor tasks upon which T depends. At compile-time, these counters are initialized to the number of incoming edges of each task node. As each predecessor task executes to completion, the counter is decremented, and when the counter reaches zero, task T is queued onto the task queue. This activity is part of the exit block code, which is inserted just after the code of T.

5.4.3 Initialization of task precedence counters

Task precedence counters are initialized at compile-time. Usually, each task precedence counter is used only once during program execution. Once the counter counts to zero, it will no longer be used as the corresponding task would have been queued. This is not true in the case of the cycle inherent in DO loops. In this case, the counters of all the nodes involved in the cycle must be initialized by the dummy node inserted after the start node of the hammock. As the execution of each iteration begins with the execution of the dummy node, the counters of the interior nodes will be initialized appropriately before they are used.

Chapter 6

IMPLEMENTATION ON CEDAR

This chapter discusses the implementation of an ASC on Cedar for the case of Fortran. We intend to use the Parafrase-2 compiler [9] as the front-end compiler. In this case, the partitioner is merely another pass[1] in Parafrase-2 to construct the task graph. What remains to be discussed is only the pre-scheduler and the drive code generator and optimizer, both of which can be implemented as separate passes. As the pre-scheduling algorithm has already been outlined in [16], we shall discuss how the Cedar run-time library can be used to implement the drive code generator and optimizer.

The following discussion, therefore, assumes that the task graph (i.e., the hierarchical task graph with each composite task node replaced by the hammock it represents) is provided by Parafrase-2. We assume that the pass that constructs the task graph also defines the scope of each variable used in a task.

[1] A pass of Parafrase-2 operates on the internal data structures to transform the program or to construct more data structures [9].

6.1 Entry Blocks

The functionality of the entry blocks includes the following: (1) allocate private variables and/or stack space, (2) copy parent stack (optional), (3) execute initialization code (optional), (4) compute the number of iterations for a helper process, and (5) update loop indices.

In the current implementation of the Cedar microtasking library, allocation of local variables for the body of an SDOALL or a subroutine is performed by the library routines each time a helper process decides to work on a queue entry in **mtask_queue**. Parameters of a subroutine are stored in a template in the corresponding queue entry, and they are copied by the helper process executing the subroutine.

In Cedar Fortran, the body of an SDOALL is not allowed to have local variables outside the scope of the SDOALL. Thus, if an SDOALL is contained within a function f, the local variables of f cannot be referenced by the body of the SDOALL. Only the loop index and the local variables of the SDOALL can be referenced within the loop body. Since the variables on the stack of the parent process are neither accessible to the subroutine nor the SDOALL, the copying of the parent stack is unnecessary.

The initialization code must be executed by each iteration, and can therefore be placed in the body of the SDOALL. On the other hand, if the initialization code can be moved outside the loop, it will be executed by the parent process of the SDOALL. The remaining tasks of the entry block are computing the number of iterations for a helper process and updating the loop indices.

6.2 Scheduling Algorithms

At present, the Cedar microtasking library has neither knowledge nor control over the scheduling algorithms that will be used for a parallel task. Scheduling algorithms are implemented by the Cedar Fortran compiler which transforms an SDOALL to include calls to the microtasking run-time library. As shown in Figures 4.3 and 4.4, self-scheduling is the only loop scheduling algorithm employed by the Cedar Fortran compiler at the moment.

Self-scheduling incurs a run-time overhead of $O(N)$, where N is the upper bound of the normalized loop index[2] . If the loop body has a constant execution time and the system is relatively lightly loaded, chunk scheduling may be preferred. In general, however, GSS is preferred since it incurs a lower run-time overhead than self-scheduling, and yet it benefits from the automatic load balancing behavior of a dynamic scheduling algorithm. To implement each different scheduling algorithm, an ASC will have to substitute the code in line (1) of Figure 4.4 as necessary.

6.2.1 Implementing GSS

GSS works by dispatching $\lceil R/P \rceil$ iterations to the next idle helper process, where P is the number of helper processes and R is the number of remaining iterations. R will be a shared variable stored in global memory, while P can be obtained from the

[2] A normalized loop is one that iterates from 1 to N, with an increment of 1.

active_helpers field of the SDOALL queue entry. The implementation of GSS using a lock, L, can be as follows:

```
LOCK L
R1 = R
x = ⌈ R/P ⌉
R = R - x
UNLOCK L
DO I = R1 - x + 1, R1
         ⋮
END DO
```

In the above solution, a process requesting lock L waits as long as another process is executing the critical section. The implementation of locks usually incorporates a queue of processes for each lock to keep track of the processes that have outstanding requests. Queue manipulation is usually very expensive. One may attempt to eliminate the critical section (and thereby avoid queue manipulation) by using **ifetch_and_op** synchronization:

```
99999    old = R;
         temp = ⌈ old/P ⌉
         new = ifetch_and_sub(R, temp);
         IF (old .ne.  new) THEN
                  ifetch_and_add(R, temp)
                  GOTO 99999
         ENDIF
         DO I = old - temp + 1, old
                  ⋮
         END DO
```

This method works by making sure that the value of the shared variable R is the same as that used to compute temp. If not, the effect of the **ifetch_and_sub** is reversed and another attempt is made. Although this solution does not give rise to deadlock, infinite postponement may result. We shall show that this implementation of GSS no longer dispatches a non-increasing number of iterations. This may lead to load imbalance among the processes executing the DOALL loop.

Consider a loop that is executed by three helper processes, Q_1, Q_2, Q_3, and initially $R = 27$. Let $new_i, old_i, temp_i$, $i = 1, 2, 3$ be the private copies of new, old and temp for the three processes Q_1, Q_2, Q_3 respectively. Initially, $old_1 = old_2 = old_3 = 27$, and $temp_1 = temp_2 = temp_3 = 9$. Without loss of generality, let the atomic subtraction operations return $new_1 = 27, new_2 = 18, new_3 = 9$ ($R = 0$ at this point). Only $old_1 = new_1$; therefore, only Q_1 proceeds. Q_2 and Q_3 will each have to add 9 back to R and retry.

If Q_2 adds 9 back to R, reads $old_2 = R = 9$, calculates $temp_2 = 3$, and performs the atomic subtraction before Q_3 completes the atomic addition to R, then Q_2 will proceed with 3 iterations instead of $18/3 = 6$ iterations that it is supposed to obtain. Q_3 will then proceed with $(9 + 6)/3 = 5$ iterations; this implementation of GSS, therefore, does not dispatch a non-increasing number of iterations. Furthermore, the iteration space executed by different processes may overlap. Using the same example, Q_2 proceeds with $old_2 = 9$ and $temp_2 = 3$, and Q_3 proceeds with $old_3 = 15$ and $temp_3 = 5$. Therefore Q_2 executes iterations 7 to 9 and Q_3 executes 11 to 15. Now, $R = 10$, and the next idle process will execute iterations 6 to 10.

Infinite postponement results if the operations of Q_2 and Q_3 occur in lock-step fashion. Consider the following sequence of events:

1. Q_2 adds 9 back to R, reads $old_2 = R = 9$, calculates $temp = 3$,

```
compare_and_swap(address, old_value, new_value)

temp = value at memory location address;
if (temp == old_value) {
        location address = new_value;
        return success;
}
else {
        old_value = temp;
        return failure;
}
```

Figure 6.1: The `compare_and_swap` synchronization instruction compares an old value and the current value of a memory location. The computed value (stored in the parameter `new_value`) is written into the memory location only when the comparison is true. Otherwise, the present value is returned in `old_value`.

2. Q_3 adds 9 to R, reads $old_3 = R = 18$, calculates $temp = 6$,

3. Q_2 performs the atomic subtraction and finds $new_2 = 18$, which is different from old_2,

4. Q_3 performs the atomic subtraction and finds $new_3 = 15$, which is also different from old_3.

At this point both Q_2 and Q_3 must add back some number of iterations to R and the lock-step execution may repeat.

The correct implementation of GSS will must ensure that the atomic subtraction is performed only when $old = new$. This can be achieved using the `compare_and_swap` synchronization [20], [33][3] . The definition of the instruction is shown in Figure 6.1 and the code for the implementation of GSS using the `compare_and_swap` instruction follows:

```
        oldR = R
88888   x = ⌈ oldR/P ⌉
        newR = oldR - x
        result = compare_and_swap(R, oldR, newR)
        IF (result = failure) GOTO 88888
        DO I = oldR - x + 1, oldR
                 ⋮
        END DO
```

This solution generates only one atomic instruction to the memory for each attempt. First, the number of iterations to be assigned, x, is computed based on $oldR$, the current value of R. Using this value of x, $newR$ is computed. Next, we call `compare_and_swap` using $oldR$ and $newR$ as parameters. Now, if the present value of R is the same as $oldR$, then the calculation of $newR$ is correct and the value of $newR$ is written into R. On the other hand, had the value of R been modified in between, then $newR$ is computed incorrectly and another attempt is made.

6.2.2 Implementing chunk scheduling

Chunk scheduling dispatches $k = \lceil N/P \rceil$ iterations each time. For simplicity, let N be an integer multiple of P, so that $k = N/P$. Line (1) of Figure 4.4 is changed to

```
        local_index = ifetch_and_add(loop_index, k) --- (1)
```

[3]A similar synchronization primitive is available in Cedar Fortran as part of the Zhu-Yew synchronization primitives [18].

6.3 Exit Blocks

When a task completes execution, one of the helper processes executing the task is required to execute the exit block. The exit block code directs the helper process to update task precedence counters, and queue each successor task with the task precedence counter of zero as a result of the update. The activities of exit blocks are, therefore, (1) barrier synchronization for parallel tasks, (2) selection of a helper process to dequeue current task from task queue, (3) update task precedence counters and queue each task that has a zero in its task precedence counter as a result of this update.

Barrier synchronization for SDOALL loops[4] is simplified by having helper processes decrement the **active_helpers** field in the SDOALL queue entry. The parent process of the SDOALL waits until that field is zero before proceeding. Effectively, a non-blocking barrier synchronization is performed. The use of task precedence counters for enforcing a task precedence relationship has been described in the previous section. Thus only one activity of the exit block remains to be considered, namely, the selection of a helper process to execute the exit block.

Exactly one helper process works on a serial task, which also executes the exit block code. The selection of a helper process for parallel tasks is not as simple. One alternative is to select the helper process that last worked on an SDOALL to execute the exit block. Each SDOALL queue entry keeps track of the number of helper processes executing

[4] Barrier synchronization for CDOALLs is achieved more efficiently by utilizing Alliant hardware, and is not discussed here.

iterations of the SDOALL in the **active_helpers** field. Helper processes **ifetch_and_sub** the **active_helpers** field when they have executed their share of iterations (see Figure 4.4). The last helper process to work on an SDOALL is, therefore, the helper process that finds a zero value in the **active_helpers** field *after* the atomic subtraction has been performed. That helper process will then execute the exit block code.

To apply the above approach, the run-time library must be changed. As the Cedar microtasking library stands at present, helper processes simply spin-wait on **mtask_queue** once they have decremented the **active_helpers** field. In addition, the **active_helpers** field does not count the parent process of an SDOALL.

We explore a third solution. Recall that in the case of Cedar, the parent process of an SDOALL must wait for the completion of all other helper processes before proceeding. Thus the parent process can be selected to execute the exit code of a parallel loop. This solution incurs some delay as the parent process must wait for the completion of all other processes, but it does not require major changes to the run-time library.

This works as follows. We first queue a parallel task as a serial task T_s. When a helper process Q works on T_s, it eventually reaches the SDOALL of T_s. Subsequently, Q queues an SDOALL entry onto **mtask_queue**, thereby becoming the parent process of that SDOALL. Other idle helper processes can now work on the loop. Once the SDOALL has been executed to completion, Q will be the last process to work on that SDOALL. Thus, Q executes the exit block code.

6.4 Transformation of a Task Graph

We shall illustrate how an ASC can transform a user program that has been partitioned into a hierarchical task graph, similar to the one given in Figure 5.2. From the hierarchical task graph, we obtain the task graph of the program by substituting each composite task node with the hammock it represents. The transformation of a hierarchical task graph (which contains composite task nodes) is discussed in the next chapter.

The task subgraph that we use to illustrate the transformation process is the subgraph of $D_1, D_2, \ldots, D_6$ in Figure 5.2. We assume that the original user program consists of the following sequence of code segments:

CODE of D_1
CODE of D_2
CODE of D_3
CODE of D_4
CODE of D_5
CODE of D_6

Each task node $D_i,\ i = 1, \ldots, 6$ may represent a parallel or a serial task. If a task node represents a serial task, we assume that the code in the task node has been encapsulated in a subroutine by the front-end compiler. The parameters and local variables of the subroutine are identified and their scope declared by the front-end compiler. On the other hand, if the task node represents a parallel task (an SDOALL loop), we can encapsulate the code for the task node in a "dummy" subroutine. Recall that the only variables that can be referenced within an SDOALL body are either local to the SDOALL (declared in the PREAMBLE), or global variables (allocated in the global memory).

Hence, the rest of the discussion assumes that tasks are already encapsulated in subroutines, so that they can be easily queued onto **mtask_queue**. If any of these tasks contain an SDOALL, for example, D_2 contains SDOALL L_2, the execution of the task proceeds as follows. When the "dummy" subroutine for D_2 is queued and dispatched by a helper process (say Q_2), D_2 is initially executed serially. When Q_2 encounters the SDOALL loop — which has been translated into a sequence of calls to the microtasking library similar to Figure 4.4, its queue entry will then be created on **mtask_queue**. Idle helper processes may work on L_2 via the SDOALL queue entry, and Q_2 becomes the parent process of L_2. When L_2 is executed to completion, Q_2 waits for all other helper processes to complete before proceeding.

Returning to our example, the original task graph of the program now consists of a sequence of CALL statements. Some of the subroutines contain SDOALL loops. For simplicity, the name of the subroutines are the same as the name of the corresponding task nodes. We shall assume that D_2, D_3, D_4, D_5 are parallel tasks, while D_1 and D_6 are serial. The code for this task graph is displayed in Figure 6.2.

Figure 6.3 illustrates the transformed code. The dashed lines indicate the execution path that would have been taken when the entire code segment is executed serially. Each SDOALL loop is transformed in a fashion similar to the example in Figure 4.4, although we show the loop without the actual translation for simplicity. At compile time, the task precedence counter for D_6 is set to 2, while that of G is set to 3. The execution of the entire subgraph then proceeds as described. A call to the **mtskstart$_**

```
SUBROUTINE D_1 ( ...)
⋮
END

SUBROUTINE D_2 ( L_2 )
INTEGER L_2
SDOALL L_2 = 1 TO N_2
⋮
END

SUBROUTINE D_3 ( L_3 )
INTEGER L_3
SDOALL L_3 = 1 TO N_3
⋮
END

SUBROUTINE D_4 ( L_4 )
INTEGER L_4
SDOALL L_4 = 1 TO N_4
⋮
END

SUBROUTINE D_5 ( L_5 )
INTEGER L_5
SDOALL L_5 = 1 TO N_5
⋮
END

SUBROUTINE D_6 ( ...)
⋮
END
```

Figure 6.2: Code for the subgraph consisting of $D_1, \ldots, D_6$. Each task node has been encapsulated into a subroutine, although some tasks are in essence parallel tasks (such as D_2). L_2, L_3, L_4, L_5 and N_2, N_3, N_4, N_5 are the loop indices for D_2, D_3, D_4, D_5, respectively.

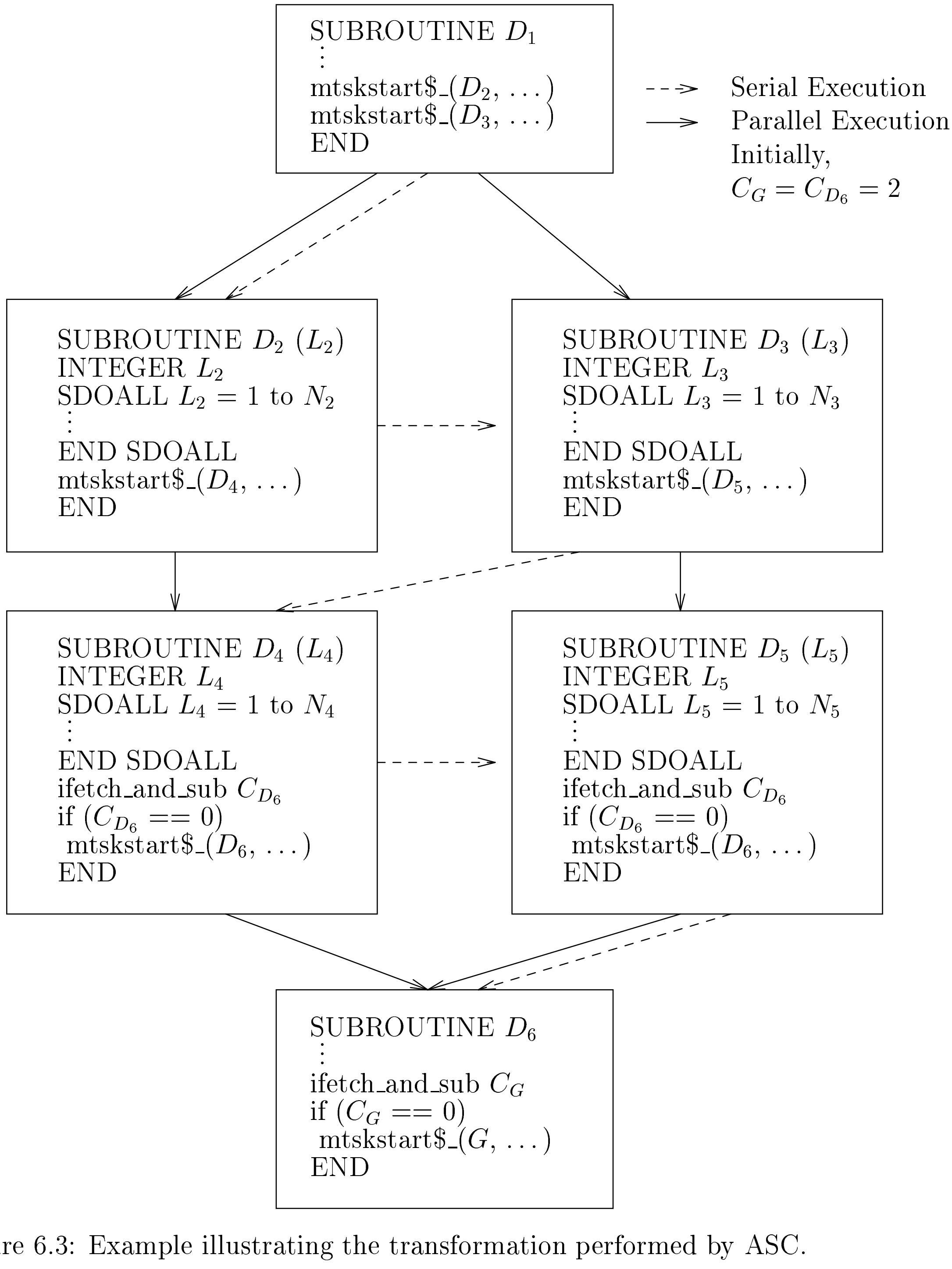

Figure 6.3: Example illustrating the transformation performed by ASC.

microtasking routine, or a call to **ifetch_and_sub**, marks the beginning of the exit block of the corresponding task node. We note that when the exit block of D_6 is executed, queueing of task G (see Figure 5.2) depends on whether the task precedence counter of G has been decremented to zero.

6.5 Future Improvements to the Run-Time Library

In the above section, we discussed how the run-time library as it now stands can be used to implement an ASC on Cedar. However, there are some features that one can improve upon for more effective exploitation of parallelism.

6.5.1 Spin waiting

Spin-waiting occurs in two places during execution in the above environment:

1. The parent process of an SDOALL must busy wait until all helper processes working on it complete.

2. When **mtask_queue** is empty, idle helper processes are forced to spin wait.

Spin-waiting is implemented in the Cedar run-time library as follows: a helper process does longer and longer sequences of NO-OPs until it reaches a predetermined maximum length. Thereafter, if a helper process finds that it still must wait, it calls the operating system to suspend itself. When the helper process executes again (by acquiring time quantum from Xylem), the whole process repeats.

In cases where there is significant load imbalance at run-time and a parent process of an SDOALL is forced to wait, or where **mtask_queue** remains empty for a long time (possibly due to the temporary lack of parallelism in the program), many context switchings may occur. This means the job incurs high run-time overhead, which is unacceptable for some applications.

As implemented, the application code has no control over how the run-time library decides to suspend helper processes due to long periods of busy waiting. Future implementations may include provisions to allow the application code to determine how such situations can be dealt with. In conjunction with the concept of auto-scheduling, an application will not rely on the run-time library for multitasking support. Instead, the ASC generates the appropriate code which accomplishes essentially the same results as a run-time library. However, this code can be tailored to the needs of each application and each task within an application. In this case, spin waiting can be eliminated whereever necessary.

6.5.2 **Tasking within clusters**

Presently, helper processes work on SDOALLs, whose iterations are to be executed concurrently in separate clusters. CDOALLs, in which each iteration is to be executed by a CE, use the hardware synchronization mechanism provided by Alliant. Although the hardware synchronization mechanism entails very low overhead, in some cases it may still be inefficient. For example, the loop scheduling algorithm employed in the hardware is self-scheduling, although in some situations GSS is preferable for CDOALL and Shortest-Delay Self-Scheduling [47] for CDOACROSS.

The above can be improved by changing the concept of a helper process to one that runs on a CE instead of a cluster. In this case, instead of relying on the hardware for loop scheduling, an ASC selects the appropriate loop scheduling algorithm. The granularity of parallelism is improved, reduced from a CDOALL nested inside an SDOALL to a simple CDOALL loop.

Chapter 7

GENERAL ISSUES OF AUTO-SCHEDULING

This chapter considers general implementation issues of an ASC. We explore the possibility of automatically adjusting the granularity of parallelism at run-time. In place of a run-time library, we assume that the code for implementing a user task queue is generated by ASC. For our discussion, we shall use a subroutine `enqueue$_` which is similar in function to the `mtskstart$_` subroutine of the Cedar microtasking library.

The task graph considered here is the hierarchical task graph defined in the previous chapter with simple and composite task nodes. Recall that composite tasks can be zoomed in to reveal the task graph of a subcomputation represented by that node. All task subgraphs have unique entry and exit task nodes (hammocks), and they in turn may be composite task nodes. Zooming in on a composite task node results in creating more tasks of lower granularity. For example, using the running example of Figure 5.2, zooming in on node C results in the queueing of tasks C_2 and C_3 on **mtask_queue** after C_1 completes.

7.1 Interaction with the Operating System

In the implementation of the microtasking run-time library, scheduling of the processes (that is, performing functions such as allocating processors and the corresponding time slices) can be left to the operating system. How processes are employed depends on how the job places tasks on the task queue using the run-time library. This is an example of a *two-level scheduler*, where the operating system schedules processors for jobs, and the processors are in turn scheduled by the jobs [48], [49], [50][1]. Under such circumstances, it is therefore possible that although a job spawns P processes, not all of them will execute at the same time. If the user program spawns P processes, and the parent process executing the user program gets suspended by the operating system before it queues tasks onto the task queue, all P processes may eventually expire after busy waiting for a predetermined time on **mtask_queue**.

In order to maximize the utilization of processors, a *coscheduling* [48] mechanism is required. Coscheduling is a static scheduling policy for an operating system. In coscheduling, time quantums are given to all processes working on the same job. Another static scheduling policy is run-to-completion [49], where each process would not be preempted until the job completes its execution. Static scheduling policies allow each job to work with a constant number of processes during the duration of its execution. On the other

[1]Using the notation of [49], we shall call the portion of the user program that schedules processors that it has acquired the *task dispatcher* of the application.

hand, dynamic scheduling policies allow the number of processes of a job to vary. Examples include the dynamic policy in [49] and the multi-level feedback queue of [14]. Note that these policies are enforced at the operating system level.

7.1.1 Static and dynamic scheduling

If a nonpreemptive scheduling algorithm (e.g., the run-to-completion policy) is used by the operating system, then an approach similar to the microtasking library of Cedar would be sufficient. In fact, a task dispatcher is not required in this approach. As exit blocks are executed, ready tasks are queued automatically. In effect, the task dispatcher is invoked whenever the exit blocks are executed.

On the other hand, if the operating system uses a dynamic scheduling algorithm, the number of processors acquired by the job varies during the execution of the job. Under high system load, the operating system may decide to run all jobs serially, or it may reduce the number of processes assigned to a job. In this case, a job that spawns tasks will experience delays for having excessively queue and dequeue operations. In order to avoid such delays, the task dispatcher must be able to decide at run-time if tasks should be decomposed and queued, or clustered together and executed serially. These decisions will depend on system load at run-time.

7.1.2 Run-time task queueing

Another reason that program parallelism must be dynamically adjusted at run-time is that user programs may contribute to adjusting the tradeoff between system utilization and run-time overhead through smarter use of system resources. If too little work is placed on the task queue, then some processes should be returned to the operating system. This scenario occurs when a user program has been partitioned into large, independent serial tasks, and has acquired more processes than tasks. On the other hand, if there are already too many queue entries but too few processes, then the program should restrict future "explosion" of parallelism. For example, in Figure 5.2, if all tasks in the hierarchical task graph are serial tasks, and the program is executed with only two processes, then the composite task nodes C and D should be executed serially. In this case, 8 atomic accesses to the queue (for the queueing of $C_2, C_3, C_4, D_2, D_3, D_4, D_5, D_6$) would be avoided.

When the composite task D in Figure 5.2 is being executed, execution begins with D_1. During the execution of the exit block of D_1, a decision is made on whether to execute the rest of the subgraph serially or as individual tasks. For the moment, we may assume that a subroutine `run_time_conditions` exists which returns a non-zero value if the run-time conditions when it is called favor breaking up a composite task node. This subroutine can be either part of a run-time library or it can be generated by ASC to interact with the operating system in order to obtain information about the system load.

If we decide to execute the rest of the hammock serially, then $D_2, D_3, \ldots, D_6$ are executed in succession as a single task by a process, involving a single queueing operation. On the other hand, if we decide to distribute the execution of the remainder of the hammock, then D_2 and D_3 are first queued as serial tasks. Upon their completion, D_4 and D_5 are queued, and finally when D_4 and D_5 are completed, D_6 is queued.

7.1.3 Composite task nodes

Let us consider the execution of a general composite task node n_c. A hammock has a unique start node v_s and a unique end node v_e. Assume that v_s and v_e are not composite task nodes. When n_c is executed, instead of queueing the entire n_c, only v_s is queued. The execution of n_c then continues with the execution of v_s, followed by the execution of the exit block of v_s. At this point, the exit block decides whether n_c should be executed serially or be broken down into its subtasks. In either case, the end node v_e is always the last task node to be queued. When v_e is executed, the exit block code of v_e updates the task precedence counters of the *successors of* n_c, and queues any tasks that become ready as a result of the update.

Recall that there is no explicit exit block for a composite task node. The exit block for a composite task node is the same as the exit block for the end node of the corresponding hammock. If the end node is again a composite task node, then the same process is recursively applied until we reach a simple end node. The exit block for all the preceding composite task nodes is that of the last end node. Similarly, the execution of a composite task node begins with the start node of its hammock. If the start node is again a composite task node, then execution begins with its start node. In addition, the task

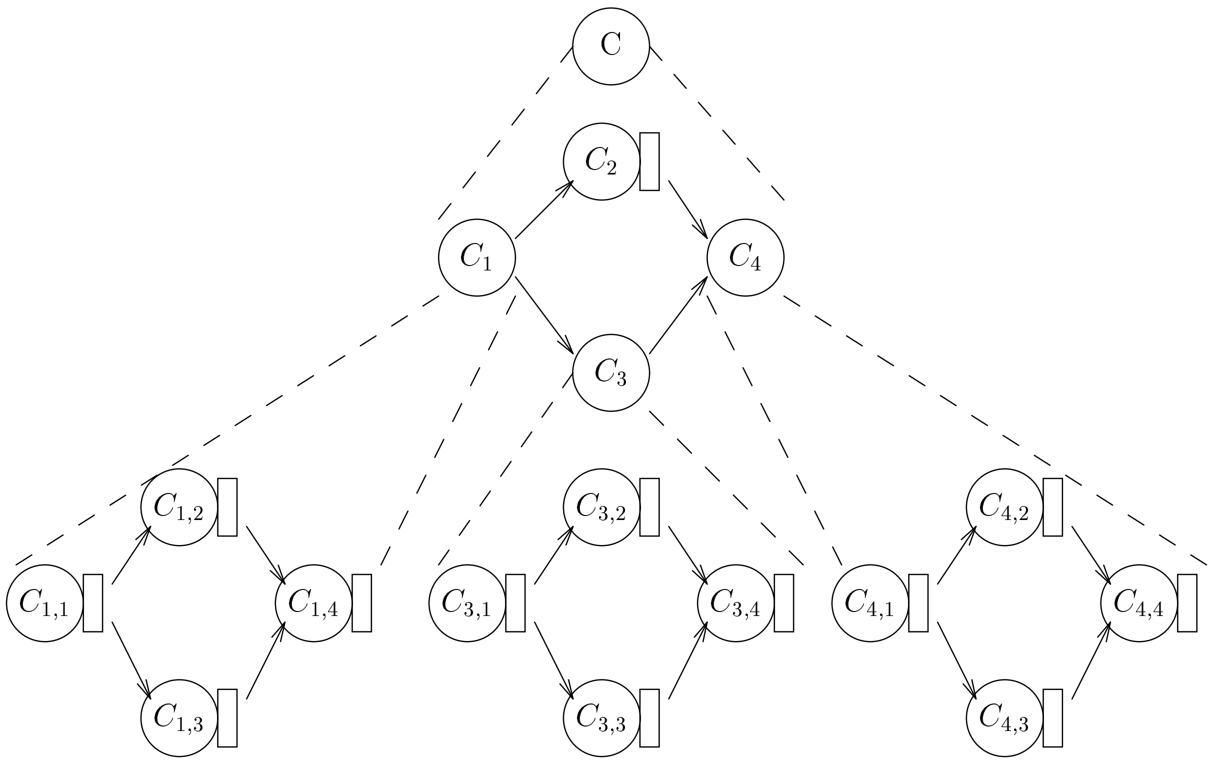

Figure 7.1: The exit block of a composite task node is the exit block of the end node of the hammock representing the composite task node.

precedence counter for a composite task node is the same as the task precedence counter for the start node of the hammock it represents.

The above is illustrated in Figure 7.1, which shows only part of the hierarchical task graph of Figure 5.2. In this diagram, however, we make tasks C_1, C_3 and C_4 composite task nodes and add simple hammocks corresponding to them. Rectangular boxes represent the exit block of the corresponding task nodes. Notice that there are no exit blocks for C_1, C_3 and C_4. The functions that need to be performed when C_1 is executed to completion are performed by the exit block for $C_{1,4}$. Similarly, the exit block for C_3 is equivalent to that of $C_{3,4}$, and the exit block for C is equivalent to that of C_4.

7.2 Automatic Adjustment of Granularity

As mentioned above, the decision on whether to break down a composite task node is made at the exit block of the start node of its hammock. This decision also determines if the exit blocks of the rest of the nodes of the hammock should be executed. For example, if it is decided that task D is to be executed serially, the exit blocks of D_2, D_3, D_4, D_5 should not be executed. This problem here is to modify the code generated for the exit blocks, so that their execution is conditioned based on whether the composite task is to be executed serially or not.

Our scheme is based on the use of a bit array $\mathbf{Z}$, which is used by each task to check if the exit block of the task should be executed. All elements of $\mathbf{Z}$ are initialized to zero at the beginning of the execution of the program. Each composite task node in the hierarchical task graph has a corresponding entry in $\mathbf{Z}$. For example, we have three hammocks in Figure 5.2, representing the composite task nodes M, C, D, and three more in Figure 7.1, representing the composite task nodes C_1, C_3, C_4. Therefore, 6 bits are required for the entire hierarchical task graph.

When the exit block of the start node for a hammock H has decided to distribute the execution of H, it sets the entry in $\mathbf{Z}$ for H to one. The exit block of an interior node of H is then executed only when the entry in $\mathbf{Z}$ for H has been set.

7.2.1 Exit block of start nodes

The exit block of a start node v_s decides if the rest of the nodes of a hammock H should be queued. This is done by calling the subroutine `run_time_conditions`, which returns a non-zero value if H is to be broken up. If `run_time_conditions` returns a non-zero value, the start node sets the entry of H in $\mathbf{Z}$ to one.

7.2.2 Exit block of interior nodes

Consider now an interior node v of H. Assume for now that v is a simple task node. If the execution of H is distributed, the exit block of v should be executed, which performs the updating of task precedence counters and the queueing of subsequent ready tasks. Otherwise, a call to the next subroutine should be performed.

For example, consider the exit block of task C in Figure 5.2. If the exit block of task A has decided to execute the hammock M (consisting of A, B, C, D, E, F, and G) serially, the exit block of C must not queue tasks E and F; instead, it should simply execute a subroutine call to either E or F. Assume a call to subroutine E is executed. When E has been executed to completion, the exit block of E again should not be executed. This time, a call to subroutine F is performed. Finally, when F is executed to completion, a call to G is performed. On the other hand, if the execution of M is distributed, the exit block of C will queue tasks E and F. The exit blocks of tasks E and F would then decrement the task precedence counter for task G and one of them will eventually queue G. The former version of the code, where straightforward subroutine calls is executed, is referred to as the serial code segment of the exit block. The code generated for an interior node v of hammock H is, therefore,

```
if (Z[H] ≠ 1) {
        serial code segment
}
else {
        exit block code
}
```

In the case where v is a composite task node, then we can find a simple task node v' of another hammock H' which has the same exit block code as v (as explained in the previous section). The exit block of v' then performs the functions of the exit block of v.

7.2.3 Exit block of end nodes

The exit block of an end node performs the functions of the exit block of the highest level composite task node that it represents. For example, referring to Figure 7.1, the exit block of $C_{4,4}$ performs the functions of the exit block of C. Therefore, $C_{4,4}$ checks the bit for M (instead of C_4 or C).

7.2.4 Code translation

In this section, we illustrate the code generated for a few selected nodes of the hierarchical task graphs of Figure 5.2 and Figure 7.1.

The code for task E is

```
if (Z[M] ≠ 1) {
        CALL F
}
```

```
else {
        ifetch_and_sub(Counter(G));
        if (Counter(G) == 0)
                enqueue$_(G, ...);
}
```

The code for task F is

```
if (Z[M] ≠ 1) {
        CALL G
}
else {
        ifetch_and_sub(Counter(G));
        if (Counter(G) == 0)
                enqueue$_(G, ...);
}
```

The code for task $C_{3,1}$ is

```
if (run_time_conditions() ≠ 0) {
        Z[C_3] = 1;
        enqueue$_(C_{3,2}, ...);
        enqueue$_(C_{3,3}, ...);
}
else {
        CALL C_{3,2};
}
```

The code for task $C_{3,2}$ is

```
if (Z[C_3] ≠ 1) {
        CALL C_{3,3};
}
else {
        ifetch_and_sub(Counter(C_{3,4}));
        if (Counter(C_{3,4}) == 0)
                enqueue$_(C_{3,4}, ...);
}
```

The code for task $C_{3,3}$ is

```
if (Z[C_3] ≠ 1) {
        CALL C_{3,4};
}
else {
        ifetch_and_sub(Counter(C_{3,4}));
        if (Counter(C_{3,4}) == 0)
                enqueue$_(C_{3,4}, ...);
}
```

The code for task $C_{3,4}$ is

```
if (Z[C_3] ≠ 1) {
        CALL C_4;
}
else {
        ifetch_and_sub(Counter(C_{4,1}));
        if (Counter(C_{4,1}) == 0)
                enqueue$_(C_{4,1}, ...);
}
```

The code for task $C_{1,4}$ is

```
if (run_time_conditions() ≠ 0) {
        Z[C] = 1;
        enqueue$_(C_2, ...);
        enqueue$_(C_{3,1}, ...);
}
else {
        CALL C_{1,2};
}
```

The code for task $C_{4,4}$ is

```
if (Z[M] ≠ 1) {
        CALL E;
}
else {
        enqueue$_(E, ...);
        enqueue$_(F, ...);
}
```

Finally, the code for task A is

```
if (run_time_conditions) ≠ 0) {
        Z[M] = 1;
        enqueue$_(B, ...);
        enqueue$_(C_{1,1}, ...);
}
else {
        CALL B;
}
```

Note that in this case, queueing of task C is actually effected by the queueing of task $C_{1,1}$, since both C and C_1 are composite task nodes.

7.3 Conclusions

The above solution using the bit vector **Z** can be easily implemented by modifying the exit code generation phase of an ASC. At run-time, the order in which hammocks are considered for distributed execution follows a pre-order search of the task nodes of the hierarchical task graph. In the future we may investigate other algorithms. The implementation of the subroutine `run_time_conditions` is also a matter for future consideration.

Chapter 8

CONCLUSIONS

Although multiprocessor supercomputers possess tremendous amounts of computing power, harnessing the power of multiprocessor systems is not an easy task. Few compilers for multiprocessor supercomputers provide automatic parallelization of user programs. This thesis has outlined the implementation of an auto-scheduling compiler that automatically partitions a user program into self-scheduling tasks. Using the Cedar run-time microtasking library, which implements a task queue unknown to the user, and the Parafrase-2 compiler as the front-end restructurer and partitioner, the paper discussed how an auto-scheduling compiler can be implemented on Cedar. This thesis also considered the issue of automatic adjustment of parallelism at run-time, which can be accomplished with a bit array which stores the status of each task node.

REFERENCES

[1] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel supercomputing today and the Cedar approach," *Science*, pp. 967–974, February 1986.

[2] P. Yew, "Architecture of the Cedar parallel supercomputer," Tech. Rep., Univ. of Illinois at Urbana-Champaign, CSRD, August 1986.

[3] E. Davidson, D. Kuck, D. Lawrie, and A. Sameh, "Supercomputing tradeoffs and the Cedar system," Tech. Rep., Univ. of Illinois at Urbana-Champaign, CSRD, May 1986.

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

[5] M. Chastain, G. Gostin, J. Mankovich, and S. Wallach, "The Convex C240 architecture," in *Supercomputing 88*, IEEE Computer Society Press, 1988.

[6] T. Jones, "Engineering design of the Convex C2," *Computer*, pp. 36–44, January 1989.

[7] D. Reed and R. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing.* Cambridge: The MIT Press, 1987.

[8] C. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, pp. 22–33, 1985.

[9] 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," *International Journal of High Speed Computing*, pp. 45–72, 1989.

[10] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system," *Journal of Parallel and Distributed Computing*, pp. 617–640, October 1988.

[11] M. J. Wolfe, "Optimizing supercompilers for supercomputers," PhD dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1982, Report UIUC-CDCS-R-82-1105.

[12] U. Banerjee, "Data dependence in ordinary programs," M.S. thesis, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, November 1986, Report UIUCDCS-R-76-837.

[13] M. Haghighat, "Symbolic dependence analysis for high performance parallelizing compilers," M.S. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1990.

[14] H. M. Deitel, *An Introduction to Operating Systems.* Reading: Addison-Wesley, 1984.

[15] *Multitasking User Guide, Cray Computer Systems, Technical Note SN-0222*, Cray Research, Inc., February 1984.

[16] C. Polychronopoulos, "Toward auto-scheduling compilers," *Journal of Supercomputing*, vol. 2, pp. 297–330, May 1988.

[17] B. W. Kernighan and R. Pike, *The UNIX Programming Environment.* Englewood Cliffs: Prentice-Hall, 1984.

[18] M. D. Guzzi, "Cedar Fortran programmer's manual - draft," Tech. Rep., Univ. of Illinois at Urbana-Champaign, CSRD, January 13, 1987.

[19] M. Booth and K. Misegades, "Microtasking: a new way to harness multiprocessors," *Cray Channels*, pp. 24–27, Summer 1986.

[20] H. S. Stone, *High-Performance Computer Architecture.* Reading: Addison-Wesley, 1987.

[21] M. Wolfe, "Multiprocessor synchronization for concurrent loops," *IEEE Software*, pp. 34–41, January 1988.

[22] C. Beckmann, "Reducing Synchronization and Scheduling Overhead In Parallel Loops," M.S. thesis, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, October 1989, Report UILU-ENG-89-8014.

[23] C. D. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," PhD dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1986, Report UIUC-ENG-86-8006.

[24] P. Tang and P. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *Proc. 1986 International Conference on Parallel Processing*, pp. 528–535, August 1986.

[25] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[26] J. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[27] R. G. Cytron, "Doacross: beyond vectorization for multiprocessors," in *Proc. 1986 International Conference on Parallel Processing*, pp. 836–844, August 1986.

[28] *Autotasking User's Guide, Cray Computer Systems, Technical Note SN-2088*, Cray Research, Inc., October 1988.

[29] D. A. Schouten, "An Overview of Interprocedural Analysis Techniques for High Performance Parallelizing Compilers," M.S. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, December 1989.

[30] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *Computing Surveys*, vol. 13, pp. 185–221, 1981.

[31] S. E. Madnick and J. J. Donovan, *Operating Systems.* New York: McGraw-Hill, 1974.

[32] C. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: a practical scheduling scheme for parallel supercomputers," *IEEE Trans. on Computers*, vol. C-36, pp. 1425–1439, December 1987.

[33] M. Dubois and C. Scheurich, "Synchronization, coherence, and event ordering in multiprocessors," *IEEE Computer*, pp. 9–21, February 1988.

[34] M. Furtney, "Parallel processing," September 1988. Technical Report, Cray Research, Inc.

[35] *FX/Series Architecture Manual*, Alliant Computer Systems Corporation, January 1986.

[36] *Guide to Parallel Programming On Sequent Computer Systems*, Sequent Computer Systems, Inc., 1987.

[37] *Concentrix System Reference Manual*, Alliant Computer Systems Corporation, January 1986.

[38] M. D. Guzzi, "Multitasking runtime systems for the Cedar multiprocessor," Tech. Rep., Univ. of Illinois at Urbana-Champaign, CSRD, July 1986.

[39] J. Davies. Private Communication following seminar given at CSRD forum, May 1989.

[40] M. Girkar, "PhD dissertation in preparation," Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1990.

[41] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* New York: Academic Press, 1972.

[42] E. Horowit and S. Sahni, *Fundamentals of Computer Algorithms.* London: Pittman, 1978.

[43] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," Tech. Rep., Michigan Technological Univ., August 1986.

[44] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, pp. 131–137, June 1972.

[45] S. Warshall, "A theorem on Boolean matrices," *Journal of the Association of Computing Machinery*, vol. 9, pp. 11–12, 1962.

[46] J. I. Munro, "Efficient determination of the transitive closure of a directed graph," *Information Processing Letters*, vol. 1, pp. 56–58, 1971.

[47] P. Tang, P. Yew, and C. Zhu, "Impact of self-scheduling order on performance of multiprocessor system," *International Conference on Supercomputing*, pp. 593–603, July 1988.

[48] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. 3rd International Conference on Distributed Computing*, pp. 22–30, October 1982.

[49] J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors," Tech. Rep., Dept. of Computer Science and Engineering, Univ. of Washington, September 1989.

[50] C. D. Polychronopoulos, "Multiprocessing versus multiprogramming," in *Proc. 1989 International Conference on Parallel Processing*, pp. 223–230, August 1989.