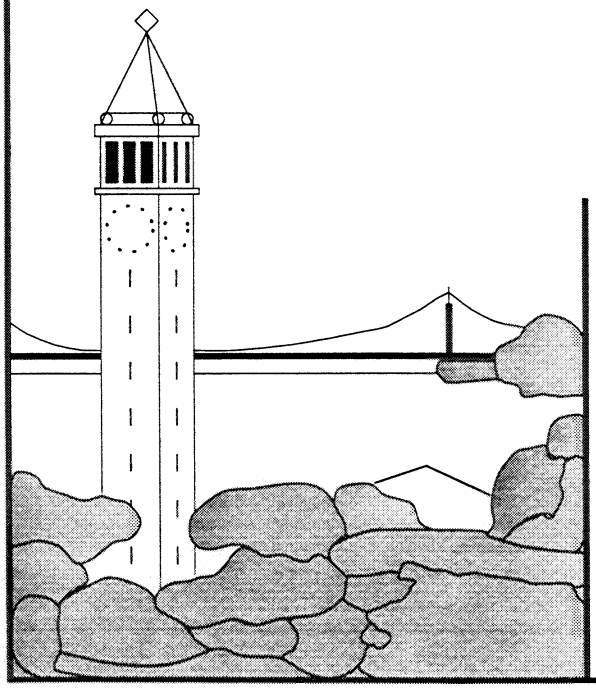
Adaptive Parallel Programs

Steven Lucco



Report No. UCB//CSD-95-864

August, 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Adaptive Parallel Programs

by

Steven Lucco

B.S. (Yale University) 1985

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Susan L. Graham, Chair
Professor James Demmel
Professor Paulo Monteiro

1994

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DOD) under contract No. DABT63-92-C-0026, and by the National Science Infrastructure Grant No. CDA-8722788 and CDA-9401156. The content of the information does not necessarily reflect the position or the policy of the Government.

Abstract

This dissertation describes a methodology for compiling and executing *irregular* parallel programs. Such programs implement parallel operations whose size and work distribution depend on input data. Irregular operations pose a particularly difficult scheduling problem because the information necessary to execute these operations efficiently can not be known at the time the program is compiled. This dissertation describes a set of four runtime scheduling techniques that can execute many irregular parallel programs efficiently. A common thread among these techniques is that they gather information about the work distribution of a program during its execution and use this information to adjust the allocation of processing resources.

The most important contribution of this dissertation is its identification and exploitation of work distribution locality properties. Previous work on irregular parallel program scheduling unearthed the following dilemma: compilers can not predict work distribution accurately enough to schedule programs efficiently; however, runtime load balancing solutions, while more accurate, incur prohibitive overhead. This dissertation shows how to avoid this dilemma whenever irregular loops within parallel programs have *work distribution locality*, that is, when a loop retains a similar distribution of individual iteration execution times from one *execution instance* to the next. An execution instance is simply an execution of the entire loop, possibly in parallel.

Where this common case arises, we exploit it through *work distribution caching*: guessing the work distribution of a loop execution instance based on earlier measurements. We also exploit work distribution locality through *deferred load balancing*: reducing the communication overhead and thrashing potential of load balancing algorithms by applying them across multiple execution instances of a loop.

We evaluated these scheduling techniques using a set of application programs, including climate modeling, circuit simulation, and x-ray tomography, that contain irregular parallel operations. The results demonstrate that, for these applications, the techniques described in this dissertation achieve near-optimal efficiency on large numbers of processors. In addition, they perform significantly better, on these problems, than any previously proposed static or dynamic scheduling method.

Contents

Chapter 1

Introduction

The aim of our research is to achieve efficient execution of parallel programs. Compiler research toward this goal has focused on four main areas: discovery of parallelism [2, 3, 48], static scheduling [18, 23, 47], linear transformation of iteration spaces to improve data locality or expose parallelism [64, 63, 62], and improved compiler technology to support the first three activities [13, 19]. In this dissertation we focus on a particularly intractable class of parallel programs for which static techniques such as these are not sufficient to achieve highly efficient execution. This is not a deficiency of the static techniques, which are often a prerequisite for efficient execution, but a property of the programs themselves. Such programs, which we call *irregular*, contain parallel operations whose size and work distribution depend on input data. Because of this property, a static schedule that performs well on one set of input data may perform poorly on others. Further, even static schedules that use random task assignments to avoid bias will have only moderate efficiency if the individual tasks have significant execution time variance.

Many basic computational techniques, such as adaptive mesh refinement [11], time-step subdivision [1, 40], tree traversal [25], and Monte-Carlo methods[60], yield irregular parallel programs. To execute such programs efficiently, we generate code that assigns tasks to processors adaptively. An adaptive scheduling policy gathers runtime information about the distribution of task execution times, and uses this information to improve the efficiency of the schedule. The thesis of this dissertation is that such adaptive scheduling methods can become highly efficient by exploiting *work distribution* locality properties. By work distribution we mean the distribution of execution times among a set of tasks, such as the individual iterations of a parallel loop. Identification and exploitation of work distribution locality can play a key role in reducing the communication and synchronization overhead incurred by *any* dynamic scheduling policy.

In this dissertation, we concentrate on creating efficient scheduling methods for distributed memory multiprocessors. The techniques we introduce are also applicable to shared memory multiprocessors. However, distributed memory architectures introduce two features that increase the difficulty of the scheduling problem. First, data sharing among cooperating processors can incur significant communication overhead. To reduce this overhead, a scheduling method must preserve *communication locality* among tasks. A parallel application has communication locality when most of the data sharing required by the

application takes place on a single processor or a set of neighboring processors. Second, scheduling methods that require centralized decision-making will incur high overhead on a distributed memory multiprocessor because the processor controlling scheduling decisions becomes a communication bottleneck. Because of these more stringent requirements, we focused primarily on distributed memory architectures.

This dissertation makes two contributions. First, it introduces some specific adaptive scheduling techniques and demonstrates that these techniques can provide more efficient parallel program execution than previously proposed methods. Second, it identifies some basic principles of work distribution locality than can be exploited within any dynamic scheduling framework.

1.1 Previous Work on Dynamic Scheduling

The goal of dynamic scheduling policies is to balance computational load among a set of cooperating processors, while avoiding excessive communication overhead. For the purpose of this discussion, we will divide previously proposed dynamic scheduling policies into two categories: self-scheduling policies [20, 49], and other load balancing policies [10, 17, 31]. Self-scheduling policies are a particular kind of load balancing policy; such policies break a group of N concurrent tasks into a set of *chunks*, where each chunk contains one or more tasks. Processors grab chunks on a first-come, first-served basis, execute the tasks in the chunk, and then grab the next available chunk. Load balancing methods have two phases. Given N concurrent tasks, a load balancing method first assigns each task to a particular processor. Processors then execute some of their tasks, while simultaneously exchanging information with other processors about their relative progress toward completion. The load balancing method tries to identify processors that are "heavily loaded", and offload some of their remaining tasks to processors that are "lightly loaded." Load balancing methods have two main variations: *local* and *global*. Local algorithms attempt to achieve overall balance through communication among neighboring processors. In global algorithms, a central manager gathers *load indices* from each processor and directs the transfer of tasks.

The source of variation among self-scheduling methods is in the number and sizes of the chunks of tasks. We give more details about these choices in Chapter 2. In Chapter 4, we show how to use runtime information about task execution time variance to improve the selection of chunk sizes for a given set of N tasks.

Despite these variations, load balancing and self-scheduling methods share a common limitation: they can incur prohibitive communication and synchronization overhead. This is especially true on distributed memory parallel architectures, where communication must take place not only to exchange meta-information about task completions but to transfer the data required to execute tasks when tasks must migrate to improve load balance. The root cause of this overhead is lack of advance information. If a runtime system knew in advance what the distribution of work would be within a parallel loop L, for example, it could assign the N concurrent iterations (tasks) of the loop to processors optimally with respect to executing L efficiently. However, with one exception [8], all previously proposed dynamic scheduling methods begin with the assumption that no information is available about the distribution of work within a loop and further that the only goal of the dynamic

scheduling method is to execute this single loop efficiently.

This assumption works well when roughly equal work is performed by each loop iteration. Under this assumption, methods can be devised that compensate for varying processor speeds [58], and uneven processor start times [49] (that might arise from previous load imbalances within the computation). When loop iterations perform significantly different amounts of work, this assumption leads to three performance problems. First, a scheduling method can often compensate for irregular execution times within one parallel loop by executing it concurrently with a regular loop or pipelining it with other concurrently executing tasks. Chapter 4 introduces some techniques for organizing this kind of pipelining.

Second, when a scheduling method decides to "offload" computation from a heavily loaded processor, it chooses a number of tasks (loop iterations) to transfer from the heavily loaded processor to the more lightly loaded processor. The unit of transfer must be "tasks" (i.e. how many loop iterations) rather than work (i.e. how much computation time) because the scheduling method has made no assumption about the distribution of work among the tasks. If some tasks perform much more work than others, the transfer of tasks may do more harm than good, and the scheduling method can not tell the difference.

For example, suppose that processor A and processor B were each initially assigned four tasks. Processor A takes twenty minutes to execute its first task, while processor B executes its first three tasks during the same time interval. The dynamic scheduling method being used specifies that, after twenty minutes, all processors will check how many tasks they have executed, compare this number with their neighbor processor and exchange tasks if there is a significant disparity. According to this policy, processor A decides that it should offload one of its remaining tasks to processor B. However, it sends to processor B a task that takes 1 minute to execute. Processor A's remaining two tasks will each take 1 second to execute whereas processor B's remaining task will take twenty minutes. The transfer has worsened load balance; further, both processors have wasted time communicating and synchronizing. If the two processors had some information about task execution times, they could avoid some of these useless or counter-productive task transfers.

The final performance problem with assuming that there is no work distribution information for each new parallel loop is that scheduling methods making this assumption tend to *thrash* as parallel loop execution nears completion. They must make increasingly fine-grained transfers of tasks to achieve even processor finish times for the loop. As processors finish, a load balancing method must perform an increasing amount of work to find processors still executing tasks, and to decide which of the finished processors should execute those tasks. Self-scheduling methods face the same problem in a different guise. They can make chunk sizes large, in which case many processors will remain idle while they wait for the last few processors to finish their large chunks. Alternatively, a self-scheduling method can make chunk sizes small, in which case it will incur prohibitive communication and synchronization overhead. The best self-scheduling algorithms balance these extremes by *tapering*: choosing large chunk sizes toward the beginning of loop execution and smaller chunk sizes toward the end of execution.

In any case, self-scheduling and load balancing methods must either restrict the grain size of task transfers or incur high synchronization and communication costs toward

the completion of a parallel loop execution. If grain size is restricted, then, relative to the ideal completion time, completion will be delayed due to load imbalance; otherwise, completion will be delayed due to scheduling overhead. Many proposals have been made as to the best compromise [20, 30, 49]. None of these proposals yield efficient distributed memory loop scheduling when iteration execution time variance is significant [38].

Hence, designers of compilation systems for irregular parallel programs face a dilemma. As we will demonstrate further in Chapter 2, static scheduling of irregular parallel programs is inherently limited in efficiency. On the other hand, we have just explained why dynamic techniques that do not have information about task execution times will suffer performance limitations. One solution to this dilemma is to have the programmer explicitly provide information about the expected completion time of each task [8]. Many of the specific methods proposed in this dissertation can straightforwardly incorporate such programmer-provided information. However, the goal of our research was to find a dynamic scheduling method that works well whether or not the programmer provides work distribution information.

Our hypothesis was that, for many parallel loops, the distribution of work within the loop will exhibit temporal locality. We further supposed that, in situations where work distribution exhibits locality, we could exploit this locality by caching information about past work distributions and using this cached information to guide scheduling decisions. We define a single execution of an entire parallel loop to be an *execution instance* of that loop. If we compare each iteration execution time for a loop execution instance A with the corresponding iteration execution time for an execution instance B of the same loop, we can obtain a measure of the *work distribution coherence* between the two instances. We say that two instances have strong work distribution coherence if the average difference in execution times among corresponding iterations is small compared to the average execution time of an iteration. A parallel loop L exhibits *temporal work distribution locality* if immediately following execution instances of L have strong work distribution coherence.

The main contribution of this dissertation is to define this locality property, to demonstrate that it is common among a sampling of irregular parallel applications, and to show how to exploit this property to achieve efficient scheduling of irregular parallel programs.

The remainder of the dissertation is organized as follows. Chapter 2 provides background information. First, it describes in more detail the features that characterize irregular programs. Second, it analyzes the performance of previously proposed static and dynamic scheduling methods on a set of irregular parallel applications, demonstrating quantitatively the performance limitations dictated by a lack of information about task execution times. Chapter 3 defines work distribution locality in detail. It then assesses whether this property holds for some of the important parallel loops executed by our set of benchmark applications. Chapter 3 concludes by demonstrating that scheduling methods can exploit work distribution locality to improve performance.

Chapter 4 presents two specific techniques, probabilistic grain size selection and parallel loop pipelining, that can benefit from enhanced runtime information about the distribution of work within parallel loops.

Chapter 5 discusses the system software necessary to support adaptive scheduling

of irregular parallel programs. It describes Delirium, an compiler intermediate form we used to concisely identify and package scheduling information required by the runtime system. Chapter 5 also presents some key features of Tarmac, a distributed shared memory toolkit that we developed to support adaptive scheduling. Finally, Chapter 6 provides a summary of the dissertation.

Chapter 2

Background

In this chapter, we describe in more detail the features that characterize irregular parallel programs. We introduce the most important of these features by presenting case studies of four application programs: a climate model, an adaptive vortex method for simulating turbulent fluid flow, a VLSI timing simulator, and a program for reconstructing x-ray tomographic images given incomplete image data. These applications illustrate that certain program loops can have work distributions that depend on input data. We will use these applications throughout the dissertation as a basis for evaluating irregular parallel program scheduling techniques.

In the latter half of the chapter, we report the results of applying previous scheduling methods, both static and dynamic, to our four irregular applications. We use these programs to illustrate why static methods often fail when faced with irregular loops. We also demonstrate that existing dynamic scheduling methods produce only modest performance improvements over static methods when applied to our four sample irregular applications.

Figure 2.1 illustrates the kind of irregular parallel loop we found most frequently in our sample applications. The `FOR` keyword indicates a loop, in this case a serial outer loop whose index variable `Timestep` ranges between the values 1 and `Max`. The `FORALL` keyword also indicates a loop, but in a `FORALL` loop, all loop iterations can execute concurrently. In Figure 2.1 for example, the inner `FORALL` loop yields `N` computations that could be executed on separate processors. The other important feature of this inner loop is that it can perform either of two different computations, depending on runtime data. Hence, if the execution times of computation 1 and computation 2 differ significantly, the inner loop of Figure 2.1 will be irregular, since for some values of `I` computation 1 will be performed, while for other values of `I` computation 2 will be performed. We will use the loop in Figure 2.1 to illustrate how scheduling methods differ in their allocation of processing resources to irregular loops.

2.1 Case Studies

In this section, we present four case studies of irregular applications. For each application, we describe the features that lead to irregular computational behavior. We also specify the applications' communication patterns and typical problem sizes.

```
FOR Timestep = 1 to Max
  FORALL I = 1 to N
    if (condition on data)
      perform computation 1
    else perform computation 2
  ENDFORALL
ENDFOR
```

Figure 2.1: Canonical irregular parallel loop.

2.1.1 Climate Modeling

Our first irregular application is the UCLA General Circulation Model. Developed at UCLA over twenty years, this program has been adopted as the standard climate model by the Earth Systems Modeling project of the Department of Energy. We experimented with a version of the program, called Camille, that has been parallelized by physicists at Lawrence Livermore National Laboratories.

Camille tries to predict climatic trends by modeling atmospheric change over a period of months or years. Camille uses a difference method to evaluate the fluid dynamics of the atmosphere. The Earth's atmosphere is divided into a three-dimensional mesh. The dimensions of the mesh are latitude, longitude, and distance above the Earth. A typical grid size for this problem is 42 longitude elements by 72 latitude elements by 9 vertical elements, although the scientists involved would like to increase this resolution a thousand-fold or more. The fluid dynamics of air flow are confined to two-dimensional shells of constant distance from the Earth. On vector machines, the vertical layers of the atmosphere would be updated in the inner loop of the fluid dynamics calculation, giving moderate vectorization across vertical grid elements.

Because fluid dynamics are confined to two-dimensional shells, Livermore physicists chose to assign all vertical elements with the same latitude and longitude values to the same processor. This reduces the need for inter-processor communication. Camille's fluid dynamics calculation requires values only from neighboring grid points to update a particular grid point. Hence, if the computation required to update the entire grid is assigned to processors in groups containing several contiguous grid points (a *BLOCKED* assignment), considerably less inter-processor communication is required than for *RANDOM* or *CYCLIC* assignments of grid points to processors.

The fluid dynamics calculations done by Camille have a regular communication pattern and would have considerable communication locality when decomposed into latitude/longitude blocks on a distributed memory multiprocessor. However, Camille's fluid dynamics account for only a third of its floating point operations. Two-thirds of Camille's computational load comes from its *column physics* operations. These operations, which are confined to columns of vertical elements that share the same latitude and longitude, calculate the effects of solar radiation and chemical reactions within the atmosphere. For example, the short-wave radiation operation measures how much energy is converted from

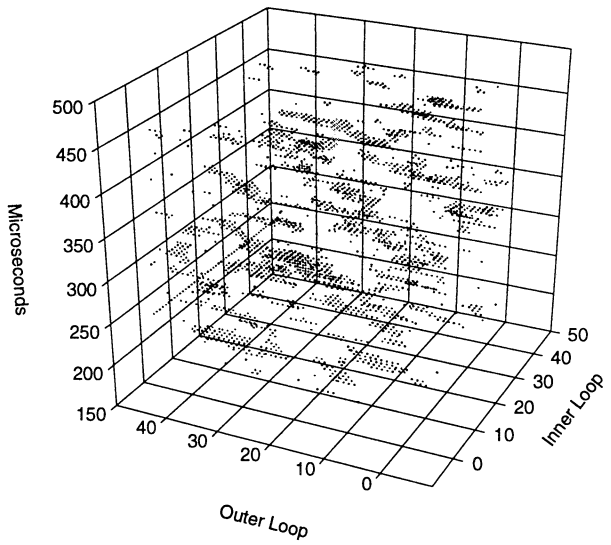


Figure 2.2: Irregular work distribution in climate modeling application.

short-wave solar radiation to heat within a given volume of air.

Unfortunately, the execution time of the column physics operations varies substantially depending on air column composition. For example, a computationally expensive part of the column physics package predicts the amount of radiation absorbed by water vapor. This calculation is only performed on a particular air volume when water vapor is present in sufficient density.

Figure 2.2 shows the distribution of work for a typical timestep within the Camille simulation. The X and Y axes represent latitude and longitude points respectively. The Z axis represents execution time. Each point in the plot indicates the execution time required to update a particular Camille grid point. As weather systems travel large distances across the Earth, dragging along their particular atmospheric conditions, the distribution of work also changes. For this reason, any static assignment of grid elements to processors will suffer significant inefficiency. In section 2.2, we demonstrate that such static assign-

ments are inefficient for typical problem sizes. To support these experiments, we simplified Camille's communication pattern. The simplified version retains the load balancing and communication characteristics described above.

2.1.2 Circuit Simulation

Our second application, EMU, is a timing simulator that is part of the MULGA circuit design system [1]. EMU divides a circuit into regions; elements of a particular region are connected by pass transistors. For each region, EMU uses a backward Euler integration to update voltage values. If this numerical method diverges, EMU subdivides the timestep and re-integrates. Because of this time-step subdivision, the distribution of work across EMU circuit regions is statically unpredictable.

Unlike Camille, EMU does not have regular communication. Circuit regions are connected in a graph that represents the electrical connections between regions. Though irregular, this graph does not change during the simulation, and the algorithm that assigns regions to processors can use static information about the graph to increase communication locality. There is a large amount of read-only data associated with each circuit region. Dynamic scheduling strategies can improve performance by replicating this information so that it does not have to be transferred when a circuit region is assigned to a new processor. The replication policy must also take into account the limited memory typically available on multiprocessor nodes.

2.1.3 Adaptive Vortex Methods

Our third example application is an adaptive vortex method for computing turbulent fluid flow (AVM) [4]. This is a three-dimensional method that uses a finer grid size wherever vortices are present. The computational structure of AVM is similar to the fluid dynamics portion of Camille. AVM performs considerably more floating point operations per memory access than Camille. As a consequence, AVM can expend considerable calculation at each timestep to determine how to refine its three-dimensional grid.

At its lowest level, the calculations performed by AVM have a regular work distribution. The program simply loops over grid elements, performing the same very expensive calculation for each element. However, at a higher level, this program exhibits exceptionally irregular work distributions. The amount of calculation required for a particular volume of space will vary drastically depending on the number of vortices present, and hence the grid refinement, within that space.

The abundance of floating point calculations at each element of turbulent flow calculations led Baden to suggest a library-based method for solving these problems [8]. In this method, the programmer divides the main grid into logical sub-grids. The programmer also provides a function that estimates at runtime the execution time of solving a particular sub-grid. Given these two components, a scheduling library can assign sub-grids to processors at runtime. This works well for adaptive turbulent flow calculations because the amount of computation involved dwarfs the overhead of communication among sub-grids or of re-assigning sub-grids to processors at each time-step of the fluid flow simulation.

The dynamic scheduling methods proposed in this dissertation are similar to Baden's library technique in that computations are assigned to processors on the basis of estimates of how long these computations will run. They are different in that they use application-independent metrics of sub-computation execution time, and we therefore expect that they will be generally applicable to irregular parallel programs. Also, the methods we propose base their execution time estimates on information collected at runtime, rather than on a programmer-provided function. Finally, our methods are incremental; they generally re-assign only a small proportion of the total work at each simulation time-step; in this re-assignment, they take into account communication locality. When the cost of floating point operations is less dominant, these algorithm features improve efficiency substantially.

Since the computations underlying the AVM application are regular, we explored two methods for obtaining execution time estimates. As with our other examples, we used direct measurement of execution times. In addition, we also tried to estimate execution times as a function of the number of iterations executed by the regular inner loops. This latter method is generally applicable to programs that explicitly change the sizes of iteration spaces to adapt to special conditions on the program's memory state.

2.1.4 X-ray Tomography

Our final example application is taken from the field of x-ray tomography. Computed tomography (CT) was originally developed for medical imaging, and applications of CT can now be found in wide use in most hospitals (x-ray transmission tomography, positron emission tomography, etc.). There is a strong and growing interest in non-medical uses of tomography as well, especially in non-destructive evaluation of solid objects such as concrete structures, machine parts, etc. For example, in civil engineering, there is a need for a non-destructive method for evaluating the integrity of structural elements such as concrete columns.

To take advantage of CT, a great deal of computation is necessary. Resolution must be fine enough to distinguish the features being examined, often requiring many hours of CPU time on supercomputers. The computational complexity of CT depends on the number of detectors, N, used to collect the x-rays. Computation cost grows as N^4, where N is currently on the order of 1024 but must grow much larger to achieve sufficient resolution. In addition, practical restrictions often prevent the collection of data from all angles. This *missing angle* problem must be addressed by using a reconstruction algorithm. Such algorithms are typically based on expensive iterations whose computational cost grows even faster than N^4.

We have been working with a new reconstruction program, called PSIRRFAN, developed by the UC/Berkeley civil engineering department [26]. PSIRRFAN exploits asymmetries in the image data to achieve better performance than previous reconstruction algorithms. However, in doing so, it introduces significant computational irregularity. The source of this irregularity is the distribution of missing input data (see Figure 2.3). In particular, where data is missing, PSIRRFAN uses sophisticated reconstruction techniques to try to recreate the missing information. These reconstruction techniques require significant execution time relative to the other operations performed by the program.

The image reconstruction algorithm uses redundant image data to reconstruct the

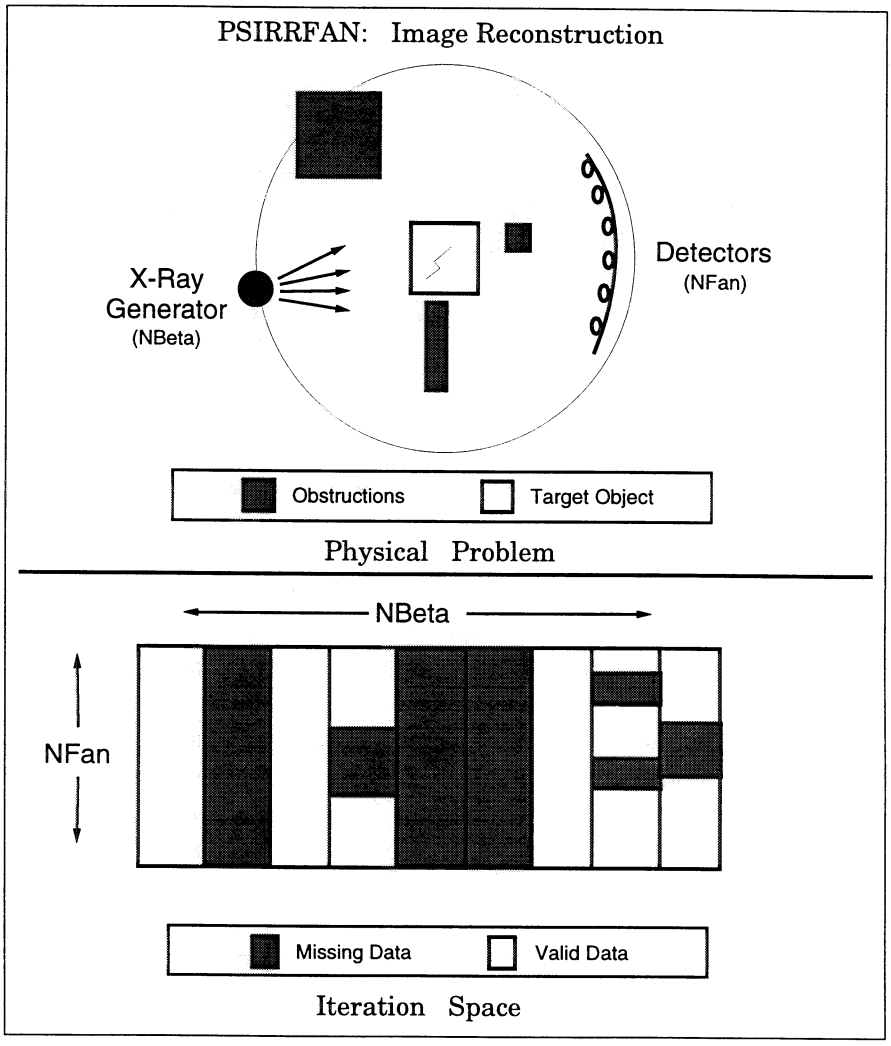


Figure 2.3: Irregular Work Distribution in PSIRRFAN

missing data. Because the pattern of missing data is not known until runtime, and can vary widely among inputs, it is impossible to predict statically how the work in PSIRRFAN's main reconstruction loops will be distributed.

PSIRRFAN is an especially interesting program because it has three computationally different phases of operation. First, PSIRRFAN maps input data into Fourier space. Second, PSIRRFAN performs image reconstruction on successive image slices. As a slice is updated, it is re-mapped into Fourier space. Finally, PSIRRFAN projects the image from Fourier space into a two dimensional array of pixels. In Chapter 4, we will introduce an adaptive runtime technique for pipelining interacting computational stages such as those found in PSIRRFAN.

2.2 Properties of Previous Approaches

Researchers have proposed many different approaches to scheduling parallel operations. Regular loops in scientific programs have been scheduled using delay insertion [18] or combinatorial exploration of possible processor assignments [47]. These approaches are not applicable to irregular parallel operations because they require loop bounds and loop body execution times to be known at compile time.

Sarkar and Hennessy have used execution time estimates and a critical path algorithm to partition parallel programs statically [53]. Our algorithm for determining a minimum grain size is similar to Sarkar's partitioning algorithm in that it balances communication costs against execution time estimates. Also, we use the algorithm reported by Sarkar [52] to estimate execution time variance. Our scheduling algorithm is different in that it uses runtime information about execution time variance to guide scheduling.

A somewhat different approach to scheduling is *load balancing*[10, 17]. Load balancing algorithms run in parallel with a computation; their goal is to re-assign tasks such that each processor ends up with the same number of pending tasks. There are two main types of load balancing algorithms: *local* and *global*. Local algorithms attempt to achieve overall balance through communication among neighboring processors. In global algorithms, a central manager gathers *load indices* from each processor and directs the transfer of tasks.

These algorithms have two features which make them more applicable to distributed operating systems than to the scheduling of a single parallel program. First, they disregard communication relationships between tasks. Second, the goal of these load balancing algorithms is to maintain even numbers of pending tasks on each processor. In contrast, the goal of the algorithms presented here is to have all processors finish a parallel operation at the same time. Load balancing algorithms keep throughput high when parallelism is abundant, but lack the precision necessary to make a single parallel operation efficient.

Tang and Yew have proposed a load balancing method called self-scheduling (SS) [59]. Self-scheduling follows the *first available* rule in assigning tasks to processors. Whenever a processor finishes executing a task, the processor becomes available and requests a new task. Self-scheduling produces even processor finishing times, even with uneven processor starting times. However, if the cost of scheduling a task is h then the expected finishing time for N tasks is $N(\mu + h)/p$, where μ is the mean task execution time and p is the number of processors (see Table 2.1). This formula incorporates the assumption that the scheduling costs will be shared equally among the processors and hence the total scheduling

Scheduling Parameters	
Parameter	**Description**
h	scheduling overhead
N	number of tasks
p	number of processors
μ	average task time
σ	task time std. dev.

Table 2.1: Parameters used to describe scheduling methods.

```
FOR Timestep = 1 to Max
  WHILE iterations remain
      obtain chunk of K loop iterations
      FOR I= StartChunk to EndChunk
         if (condition on data)
           perform computation 1
         else perform computation 2
      ENDFOR
  ENDWHILE
  synchronize with all other processors
ENDFOR
```

Figure 2.4: Canonical irregular parallel loop modified to perform self-scheduling on each processor.

cost, Nh, is divided by the number of processors, p. Unless h is significantly less than μ, overhead of self-scheduling will be prohibitive.

Self-scheduling can be generalized so that processors are given a *chunk* of K tasks whenever they become available. Kruskal and Weiss suggest a technique (SSO) [34] for determining the optimal value of K, given N, p, σ, and μ, where σ is the standard deviation of task execution times. However, any method that uses a single chunk size K has expected unevenness of $K\mu/2$ in its processor finishing times. If we choose a large value of K, this unevenness will be significant. If we choose a small value, we incur a large total scheduling overhead.

Figure 2.4 shows how a self-scheduling method would execute the canonical irregular parallel loop presented in Figure 2.1. The loop in Figure 2.4 would be executed by each of the p processors. Note that, since the original outer loop was serial, all processors must be synchronized at the end of each parallel inner loop execution. This is why it is critical that the p processors finish the parallel inner loop at roughly the same time.

One would like to have a strategy that combines the low runtime overhead of large chunk sizes with the even finishing times of self-scheduling. In Chapter 4, we investigate the

Scheduling Methods	
Method	**Description**
SS	self-scheduling
SSO	SS; "optimal" single chunk
FAC	factoring
GSS	guided self-scheduling
LLB	local load balancing
GL1	global load balancing (1)
GL2	global load balancing (2)

Table 2.2: Key to scheduling method performance figures.

finishing times of *tapering* methods, which use the first available rule but reduce runtime overhead by scheduling large chunks at the beginning of a parallel operation and successively smaller chunks as the computation proceeds. In theory, the smaller chunks should smooth out uneven finishing times left by the larger chunks.

Polychronopoulis and Kuck suggest a tapering method, guided-self-scheduling (GSS) [49], that chooses chunk size $K_i = \lceil R_i/p \rceil$ where R_i is the number of tasks remaining after the $i-1^{st}$ chunk has been scheduled ($R_1 = N$). The goal of this tapering rule is to smooth out uneven processor start times. It is optimal when $\sigma = 0$ (where σ is the standard deviation of task execution times), but does not address the problem of variable task execution times. Hummel *et. al.* [30] propose a similar method called Factoring (FAC) that chooses chunks more conservatively than GSS, and hence performs somewhat better when task execution times are variable.

Chapter 4 presents a new grain-size selection technique, TAPER, that incorporates information about task execution variance. It chooses smaller chunk sizes where variance is high, and larger chunk sizes where variance is low.

2.2.1 Shared Memory Performance

In this section, we analyze the shared memory performance of these previously proposed static and dynamic scheduling methods. When contrasted with the performance measurements in the next section, these measurements illustrate that the performance of a scheduling method on a shared memory multiprocessor does not necessarily predict its performance on a distributed memory multiprocessor.

We applied a representative set of static, self-scheduling and load balancing methods to our four benchmark applications. Figures 2.5 through 2.16 compare the efficiencies of these methods at scheduling each of these applications on an eight processor Cray Y/MP.

The X axis for each of these figures represents problem size. The Y axis represents *efficiency* on eight processors. We define efficiency as *speedup* divided by *ideal speedup*. Speedup is defined as the time required to execute an optimized sequential version of the application divided by the time required to execute the parallel version of the application. Ideal speedup is defined as the the number of processors used to execute the parallel version,

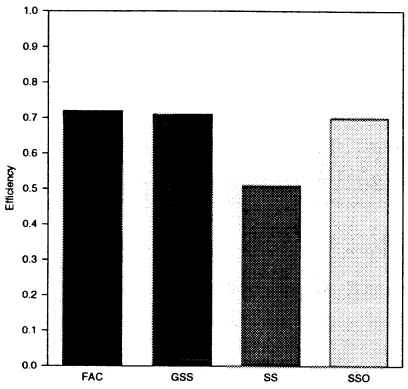


Figure 2.5: Climate on 8 processor Cray Y/MP; self-scheduling methods.

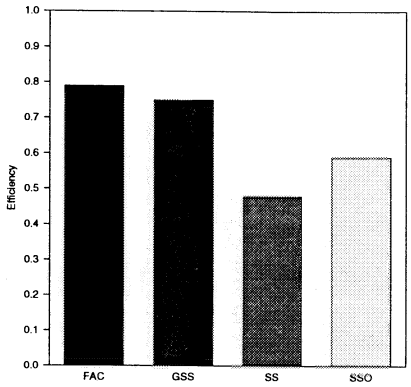


Figure 2.6: AMR on 8 processor Cray Y/MP; self-scheduling methods.

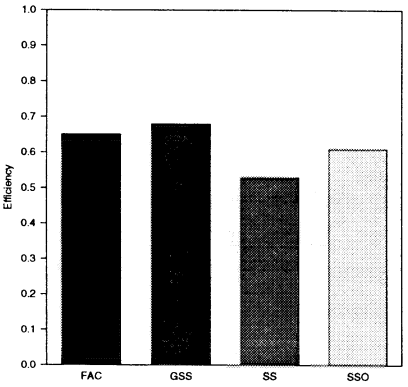


Figure 2.7: EMU on 8 processor Cray Y/MP; self-scheduling methods.

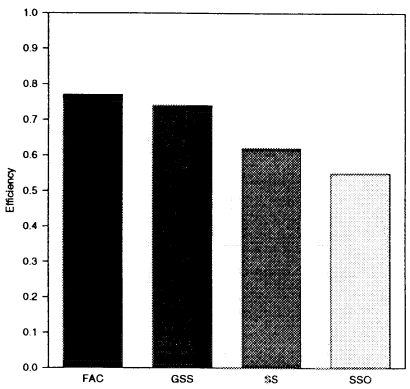


Figure 2.8: Psirrfan on 8 processor Cray Y/MP; self-scheduling methods.

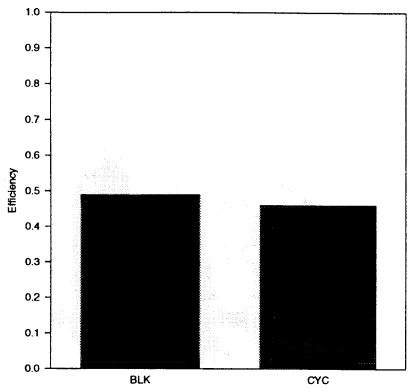


Figure 2.9: Climate on 8 processor Cray Y/MP; static methods.

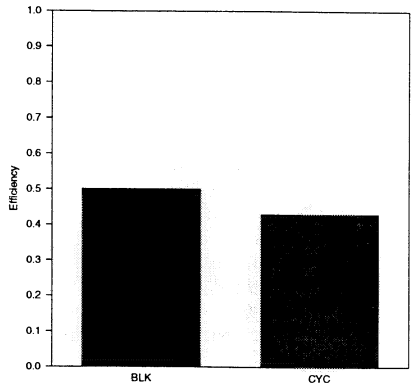


Figure 2.10: AMR on 8 processor Cray Y/MP; static methods.

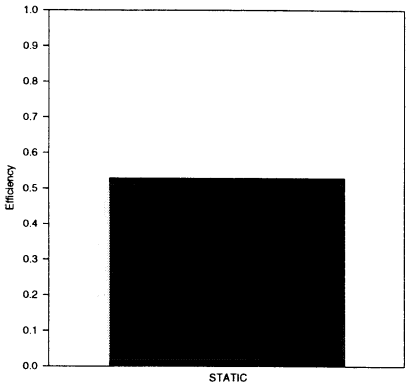


Figure 2.11: EMU on 8 processor Cray Y/MP; static methods.

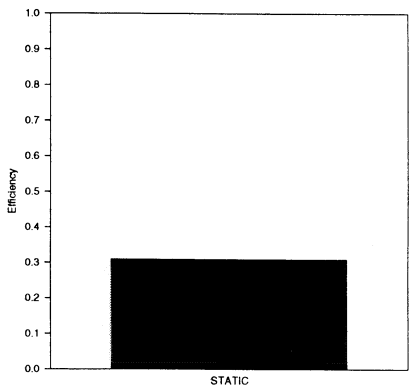


Figure 2.12: Psirrfan on 8 processor Cray Y/MP; static methods.

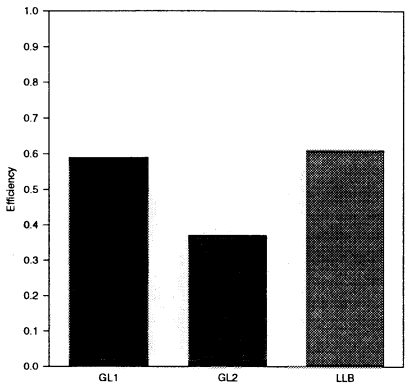


Figure 2.13: Climate on 8 processor Cray Y/MP; load balancing methods.

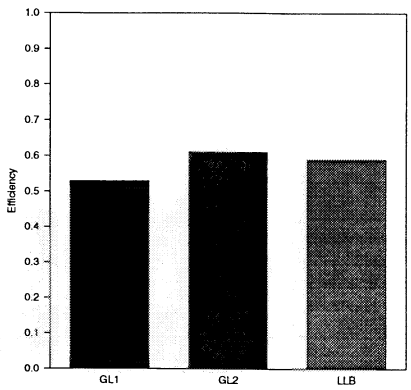


Figure 2.14: AMR on 8 processor Cray Y/MP; load balancing methods.

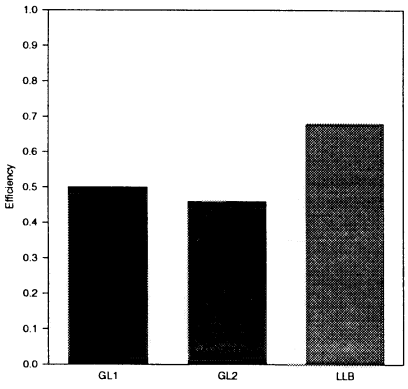


Figure 2.15: EMU on 8 processor Cray Y/MP; load balancing methods.

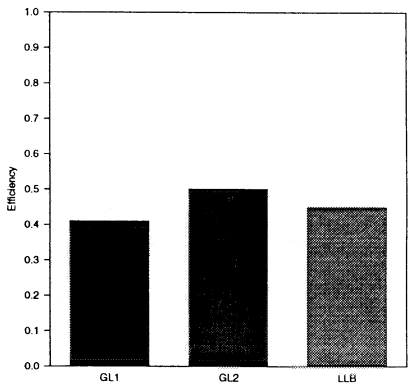


Figure 2.16: PSIRRFAN on 8 processor Cray Y/MP; load balancing methods.

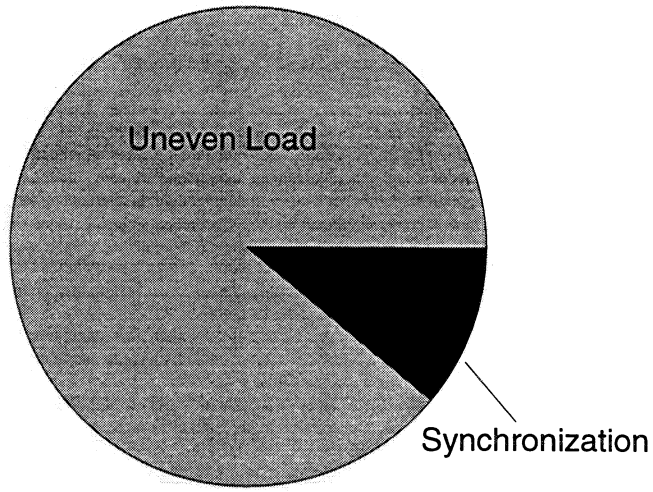


Figure 2.17: Contributions to static scheduling overhead in the X-ray Tomography Example.

i.e. in the ideal case, the sequential computation time is divided evenly among the processors with no additional overhead time. Efficiency is a metric of scheduling method effectiveness. The perfect scheduling method incurs no overhead and maintains perfect load balance, yielding efficiency of 1.0.

We can observe from the measurements shown in Figures 2.9 through 2.12 that none of the static scheduling methods discussed above execute the benchmark programs with high efficiency. Note that for the climate model and adaptive mesh refinement, both BLOCKED and CYCLIC data decompositions are possible; for PSIRRFAN and EMU, only a BLOCKED static decomposition makes sense and so there is only one static scheduling measurement for these applications. Figure 2.17 shows the relative contributions of load imbalance and synchronization overhead to the reductions in static scheduling efficiency for PSIRRFAN. Figure 2.17 shows that for the BLOCKED method used on PSIRRFAN and the

other applications (contiguous groups of tasks assigned to processors) load imbalance causes most of the inefficiency. This was also true for both applications that could use a CYCLIC decomposition (tasks assigned to processors periodically). As illustrated by Figures 2.5 through 2.8 and Figures 2.13 through 2.16, dynamic methods perform relatively well on the Cray Y/MP. These methods, such as guided-self-scheduling (GSS), achieve their main goal of reducing load imbalance, at moderate synchronization expense.

2.2.2 Distributed Memory

Unfortunately, these dynamic methods are communication-intensive, and their performance suffers on distributed memory multiprocessors. On distributed memory machines, we can no longer assume that the scheduling of a chunk of tasks has a fixed overhead. Each task may require a certain amount of data for its computation, so there will be a per-task as well as a per-chunk transfer cost. Further, we often need to preserve communication locality by maintaining a minimum chunk size. Since, for most of our example applications, neighboring tasks communicate much more frequently than distant tasks, separating most neighboring tasks would drastically increase communication cost. A scheduling method can avoid this situation by maintaining a minimum contiguous block size for transfer of work.

Self-scheduling methods such as GSS and Factoring (FAC) were designed for shared memory. Since they centralize the pool of remaining work, they can be extremely inefficient on distributed memory multiprocessors. For example, we will show that, on our four sample applications, no self-scheduling method achieved more than 23% efficiency.

To investigate their effectiveness on distributed memory machines, we created distributed memory versions of shared memory self-scheduling algorithms such as SS and GSS. In the distributed memory versions, we avoid placing all the tasks in a central queue, because we found that centralizing the scheduling of tasks created a synchronization bottleneck. Instead, we begin with some original data decomposition (assignment of tasks to processors).

In our distributed memory algorithm the p processors are logically connected as a binary tree with p leaves. Some of the processors act both as leaves and as internal nodes of the tree. We modify each self-scheduling algorithm so that each processor chooses the same sequence of chunk sizes. When a processor chooses a new chunk size and starts executing tasks from that chunk, we say it is beginning a new *epoch*. SS and FAC already yield p chunks of the same size. In SS, all chunks are of size 1. In FAC, the p chunks from epoch i are of size $R_i/2p$, where R_i is the number of tasks remaining to be executed (in the remaining epochs). For GSS, we modified the chunk size selection algorithm $K_i = R_i/p$, to $K_i = 2R_i/3p$ for each epoch.

All processors start in epoch 0. When a processor begins executing a chunk it sends its current epoch value (called a *token*) to its parent, which passes the token to its parent (possibly combining messages from both children). When the root receives p tokens from the same epoch, it increments the global epoch value and broadcasts (through the tree) a message to all processors. The message tells the processors to increment their epoch value and may also tell some processors to transfer a chunk of tasks (and their associated data) to another processor.

Processors compete for the p chunks of each epoch. If processor a can get two

```
FOR Timestep = 1 to Max
  WHILE iterations remain
      receive task transfer instructions
      initiate task transfers, if any
      compute K[i], the size of next chunk
      FOR I= StartChunk to StartChunk+K[i]
         if (condition on data)
           perform computation 1
         else perform computation 2
      ENDFOR
      notify parent that chunk i is done
  ENDWHILE
  synchronize with all other processors
ENDFOR
```

Figure 2.18: Canonical irregular parallel loop modified to perform self-scheduling on each processor of a distributed memory multiprocessor.

tokens of value i to the root before processor b can send one token of value i, then the root will re-assign to processor a the chunk of size K_i that would have gone to processor b. Processor b is then forced to re-interpret the chunk it is currently executing as belonging to some later epoch. If most of the actual task cost is on a few processors, this scheme will degenerate into a centralized scheduling algorithm. If task costs are independent then we expect most tasks to remain on the processor owning them at the beginning of the parallel operation; thus, the algorithm reduces task transfer costs and maintains communication locality.

Figure 2.18 shows the loop that each processor would execute under our distributed self-scheduling method. This method is different from regular self-scheduling in that each processor begins with N/p of the loop iterations. Also, each processor computes how many loop iterations to perform locally. After completing a chunk of loop iterations, each processor notifies its parent. If a processor is excessively tardy, it may be notified by its parent to offload some of its loop iterations. If it has completed its loop iterations more quickly than average, it may receive a message giving it responsibility to execute an extra chunk of tasks. Such messages will also contain the data required to execute these tasks.

2.2.3 Distributed Memory Performance

In this section, we present the results of repeating our suite of shared memory benchmark experiments, this time using a distributed memory multiprocessor, the Ncube-2. Figures 2.19 through 2.22 compare the efficiencies of several self-scheduling methods on a 512 processor Ncube-2. Figures 2.23 through 2.26 compare the efficiencies of static scheduling methods on the Ncube. Figures 2.27 through 2.30 compare the efficiencies of representative local and global load balancing methods with some of the more efficient self-

scheduling methods. These figures show that no previously proposed static or dynamic scheduling method can efficiently execute more than two of the four example applications. Most of the methods do not exceed 60% efficiency for any of the four example applications.

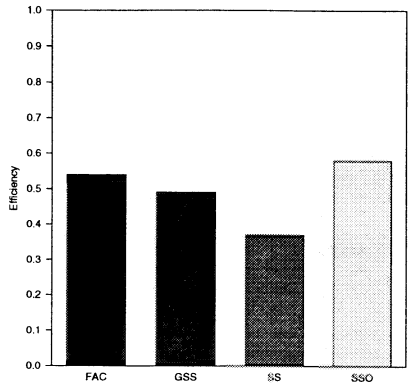


Figure 2.19: Climate on 512 processor Ncube-2; self-scheduling methods.

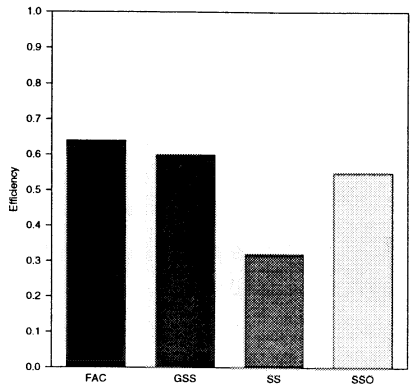


Figure 2.20: AMR on 512 processor Ncube-2; self-scheduling methods.

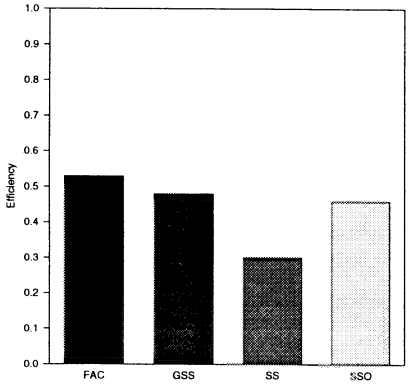


Figure 2.21: EMU on 512 processor Ncube-2; self-scheduling methods.

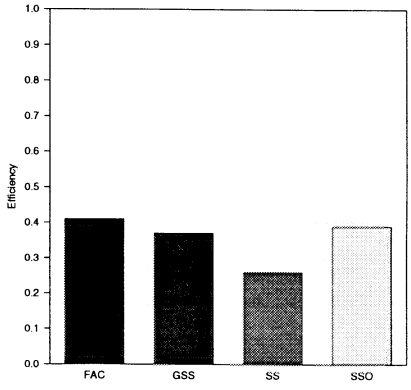


Figure 2.22: PSIRRFAN on 512 processor Ncube-2; self-scheduling methods.

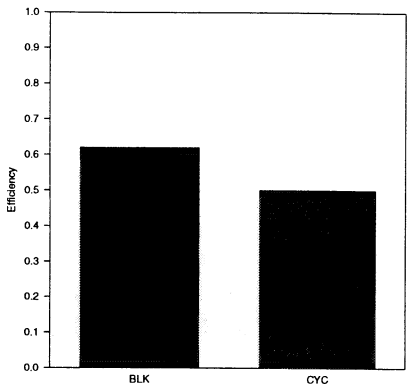


Figure 2.23: Climate on 512 processor Ncube-2; static methods.

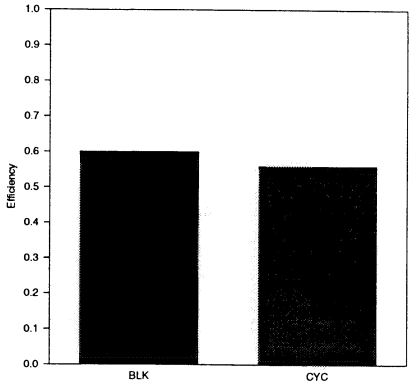


Figure 2.24: AMR on 512 processor Ncube-2; static methods.

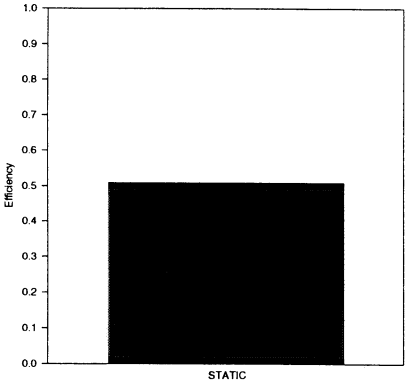


Figure 2.25: EMU on 512 processor Ncube-2; static methods.

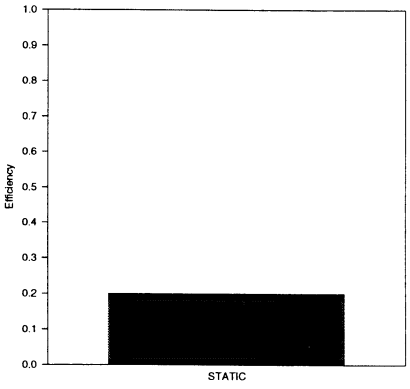


Figure 2.26: PSIRRFAN on 512 processor Ncube-2; static methods.

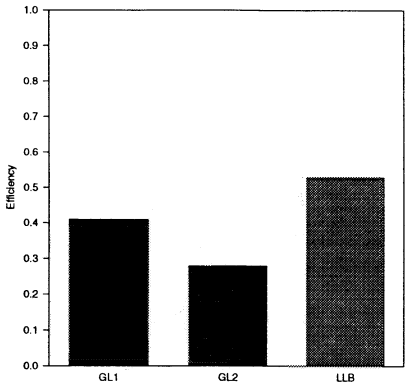


Figure 2.27: Climate on 512 processor Ncube-2; load balancing methods.

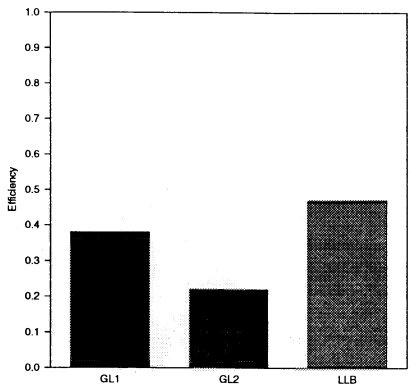


Figure 2.28: AMR on 512 processor Ncube-2; load balancing methods.

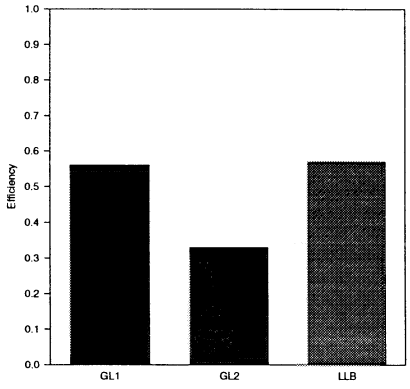


Figure 2.29: EMU on 512 processor Ncube-2; load balancing methods.

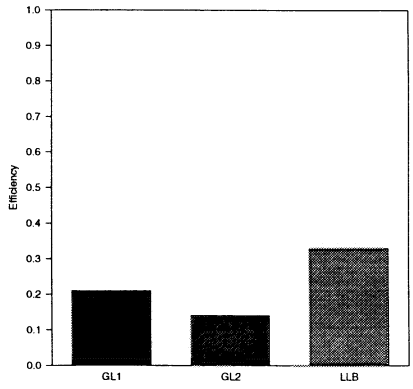


Figure 2.30: Psirrfan on 512 processor Ncube-2; load balancing methods.

In this section, we also present the results of applying several load balancing methods to our sample applications: one local load balancing method and two global load balancing methods. The local load balancing method and the first global load balancing method are hybrids, designed to capture the best features of previously proposed methods. The second global load balancing method is a recently proposed method that claims substantial performance improvement over previously proposed methods.

The local load balancing method (LLB) has each processor in the hypercube communicate only with its neighbors. At the beginning of parallel loop execution, subsets of the loop's iterations are assigned equally among the participating processors. Each processor then executes some number of loop iterations K_{sync}, without synchronization. After K_{sync} iterations, processors query their neighbors to find out how many iterations each of their neighbors have executed. If a disparity of greater than $2K_{min}$ iterations is found, the laggard processor donates half of its iterations to the processor that is ahead. The results presented here are for the values of K_{min} and K_{sync} that yielded the highest efficiency.

Global load balancing method GL1 begins with the same initial assignment of iterations to processors as LLB. Each processor then executes K_{sync} of its iterations (this may be all the iterations assigned to the processor) without communication. After K_{sync} iterations, the processor sends a load index (the number of iterations remaining) to its parent processor in a tree of processors like that used by the distributed self-scheduling algorithm. If a given node in this tree detects a disparity of greater than $2K_{min}$ iterations between any of its descendent processors, it sends messages to these processors dictating an exchange of half the disparity in iterations.

Global load balancing method GL2 [51] also uses the same initial assignment of iterations to processors as LLB. However, this time each processor executes all its iterations before initiating communication. This is a sub-case of the GL1 scenario where K_{sync} is equal to the number of iterations initially assigned to the processor. Unlike in GL1, when a processor A finishes its iterations, it chooses another processor B at *random*, and requests that B transfer half its remaining iterations to A. This method keeps throughput high as long as parallelism is abundant. However, it thrashes tremendously toward the completion of parallel loop execution. As most of the processors finish, they compete to find the few processors still executing loop iterations, creating a storm of random communication, usually netting little improvement in load balance and incurring large overhead.

For both Climate and X-ray Tomography, GL2 had the worst efficiency. Most of the overhead of this strategy was due to thrashing in the later stages of loop scheduling. Figure 2.31 shows how GL2 scheduling overhead increases dramatically during the latter stages of loop execution. The loop executed is one of the main loops found in the X-ray Tomography application. This figure also shows that LLB and GL1 suffer significant thrashing toward the completion of this parallel loop. The profile of overhead given in Figure 2.31 is typical of the behavior we observed on all the major irregular parallel loops of our four applications.

Figures 2.32 and 2.33 summarize the important causes of overhead for both sets of dynamic scheduling methods: self-scheduling and load balancing. These figures depicts overheads on the X-ray Tomography application, but are typical of the other three applications as well. Together with Figures 2.19 through 2.30, these figures illustrate the dilemma

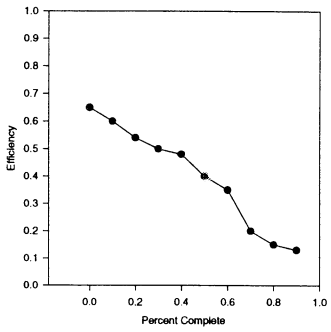


Figure 2.31: Degradation in GL2 efficiency for typical PSIRRFAN loop.

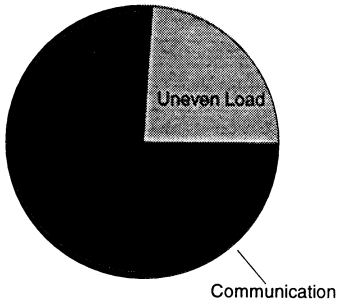


Figure 2.32: Contributions to load balancing overhead of X-ray Tomography.

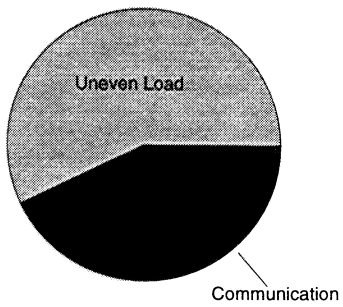


Figure 2.33: Contributions to self-scheduling overhead of X-ray Tomography.

we faced when we tried to apply previous dynamic and static scheduling methods to irregular parallel programs: static methods incur too much load imbalance because they can not predict uneven loop iteration execution times; dynamic methods must choose between load imbalance and high scheduling overhead. To solve this dilemma, we looked for a way to predict loop iteration execution times, so that we could balance computational load without incurring high scheduling overhead. Chapter 3 presents our attempt to solve this problem.

Chapter 3

Work Distribution Coherence

In this chapter, we analyze the work distributions of our four example applications. We observe that the important parallel loops in these applications exhibit considerable temporal work distribution locality. We define a single execution of an entire parallel loop to be an *execution instance* of that loop. If we compare the execution times of each iteration from a loop execution instance A with the execution time of the corresponding iteration from an execution instance B of the same loop, we can obtain a measure of the *work distribution coherence* between the two instances. We say that two instances have strong work distribution coherence if the average difference in execution times among corresponding iterations is small compared to the average execution time for an iteration. A parallel loop L exhibits *temporal work distribution locality* if successive execution instances of L have strong work distribution coherence.

Once we have established that this locality property exists in our set of benchmark applications, we demonstrate two techniques for exploiting it. First, we show that any dynamic scheduling technique can benefit from *work distribution caching*, in which a *cost function* is stored that maps iteration number to execution time for each of the important parallel loops in the program. A dynamic scheduling method can use this cost function to scale task transfers so that the intended amount of *work* is transferred. Second, we show that many dynamic scheduling methods can benefit from *deferred load balancing* in which decisions about task transfers are made during one execution instance of a parallel loop, but carried out across several instances. Deferred load balancing is especially effective when the underlying multiprocessor architecture supports overlapping of communication with computation.

3.1 Work Distribution Locality

We analyzed the work distributions of our four example applications by tracing the execution time of each loop iteration for each execution instance of each important parallel loop. A parallel loop was deemed important if it represented more than one percent of the parallel program's execution time. We post-processed the traces by comparing corresponding loop iterations of successive execution instances of each parallel loop. We also looked at trends across several successive parallel loop execution instances.

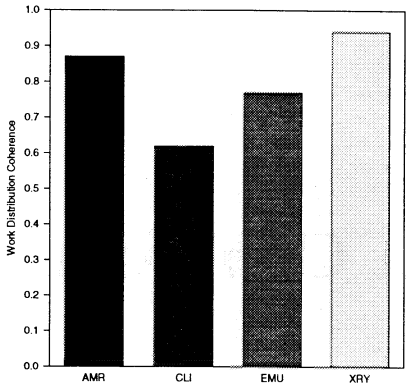


Figure 3.1: Average distribution coherence over all important loops for each parallel application

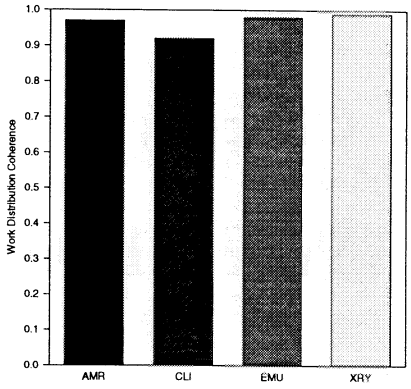


Figure 3.2: Average distribution coherence over all important loops for each parallel application; loops subdivided by call-site.

For each application we measured the average distribution coherence between all pairs of successive parallel loop instances. For example, suppose an application contains a single parallel loop. Further suppose that the parallel loop is executed E times during a run of the application program. Then there would be $E - 1$ pairs of successive execution instances for this loop. Each execution instance has N iterations, and hence each pair of execution instances has N pairs of corresponding iterations. We define the distribution coherence between a pair of corresponding iterations to be the ratio of the smaller execution time to the larger execution time. Hence, if a pair of iterations have the same execution time, they will have a distribution coherence of 1.0. If they have radically different execution times, their distribution coherence will be near zero.

Figure 3.1 summarizes these results. It presents, for each application, the average work distribution coherence between all pairs of successive parallel loop execution instances. Our initial results showed strong work distribution locality in two applications, X-ray Tomography and AMR, but weaker locality in the other two applications.

We then refined our instrumentation system to cache parallel loop work distributions according to the dynamic call graph environment of the parallel loop. A function enclosing a parallel loop may be called from many different call sites. Each call site may pass different inputs to the function and hence different parameters to the parallel loop. If we separate cached work distribution information by dynamic call-site, we can compensate for this effect. Figure 3.2 summarizes the work distribution locality we observed using the refined instrumentation system. For this figure, execution instances were separated into equivalence classes defined by a 3 function deep call trace. Successive execution instances in

the same equivalence class were measured for work distribution coherence in the same way that all successive instances of each important parallel loop were measured for Figure 3.1. The work distribution coherence between successive instances in the same equivalence class was extremely strong.

3.2 Work Distribution Caching

When programs exhibit strong work distribution locality, like our four example applications, we can use *work distribution caching* to build a *cost function* for each parallel loop.[1] We can use this cost function to adjust task transfers so that the intended amount of work is transferred. We call the number of tasks that would have been transferred by a given scheduling method n_{orig}. To obtain the number of tasks to transfer, we add or subtract tasks until we obtain an amount of work equal to $n_{orig}\mu$ where μ is the average amount of work done to execute a task. This type of adjustment based on a cost function can be applied to any dynamic scheduling method, to avoid the unnecessary or counter-productive task transfers that are inevitable when a method has no information about task execution times.

In Chapter 1, we presented an example in which processor A and processor B were each initially assigned four loop iterations. Processor A takes twenty minutes to execute its first loop iteration, while processor B executes its first three loop iterations during the same time interval. The dynamic scheduling method described in Chapter 1 specifies that, after twenty minutes, all processors will check how many loop iterations they have executed, compare this number with their neighbor processor and exchange loop iterations if there is a significant disparity. According to this policy, processor A decides that it should offload one of its remaining loop iterations to processor B.

Without loop iteration execution time information, processor A sends to processor B a loop iteration that takes 1 minute to execute. Processor A's remaining two loop iterations will each take 1 second to execute whereas processor B's remaining loop iteration will take twenty minutes. The transfer has worsened load balance; further, both processors have wasted time communicating and synchronizing.

In this example, and in the presence of work distribution information, a dynamic scheduling method can use the cost function built from previous execution instances of the same parallel operation (*i.e.* loop) to avoid this type of counter-productive loop iteration transfer. The method would use the cost function to observe that, in the last execution instance of the loop, processor B's single remaining loop iteration required much more time to execute than processor A's three remaining loop iterations. Hence, it would not initiate a loop iteration transfer.

3.2.1 Implementation

In our prototype system, we implemented work distribution caching using the microsecond counters available on both the Cray Y/MP and the Ncube-2. Using these

[1] From this point, when we refer to a parallel loop, we are referring to a single call-site equivalence class of a parallel loop.

memory-mapped counters, the overhead of sampling a loop iteration execution time was only a few machine instructions. Using a FORTRAN preprocessor we associated a unique identifier with each static call-site in the program. We also modified the call sequence to maintain a runtime stack trace (3 deep from the innermost frame). Using pre-processing coupled with profiling information for some programs (such as X-ray Tomography) and hand compilation for other programs, we identified the important parallel loops in each program, and gave each a unique identifier.

At runtime, our scheduling system combines parallel loop identifiers with call-stack traces to decide the equivalence class of each parallel loop execution instance. The scheduling system builds a cost function for each parallel loop equivalence class. On distributed memory architectures, this cost function is *distributed*, in that different processors may store different portions of the cost function. In general, a processor will store at least that portion of the cost function which corresponds to the loop iterations it is currently assigned.

In addition, each processor maintains summary information about the cost function of each parallel loop. This summary information includes the mean loop iteration execution time (μ), and the variance in loop iteration execution times (σ^2).

On each processor, cost functions are built by *execution time sampling*. Suppose that a given processor P has been assigned loop iterations i through $i+k$ of a parallel loop. If k is large, or μ is small relative to the overhead of sampling, P will choose a random subset of the iterations between i and $i+k$ to sample. This random subset is called the *base sample*. Using a base sample rather than sampling all iterations reduces the execution time overhead of sampling. In practice, this overhead was usually insignificant. However, using a base sample also reduces the amount of memory required to store a cost function. This was very important in practice, because most parallel loops that we encountered iterated over large arrays, hence the iteration space contained millions of elements and a complete cost function would require several megabytes to store.

Adaptive Work Distribution Sampling

In practice, using a base sample rather than a complete sample impacts performance only positively. This is because our scheduling methods rely much more heavily on aggregate characteristics such as the mean and variance of execution times than on individual loop iteration execution times. However, in some cases portions of a loop will exhibit wide variation in execution times. In such cases, a uniformly random base sample does not contain enough information to prevent counter-productive task transfers.

To handle these cases without incurring prohibitive memory overhead, we introduced *adaptive work distribution sampling*. Under this sampling scheme, each processor computes a base sample as before. In addition, each processor computes summary information about its base sample that it uses to decide where to sample further. In particular, sections of the sample which have a high variance compared to the variance of the parallel loop as a whole are sampled further. Also, sections of the sample which have a high average iteration execution time compared to the average iteration execution time of the parallel loop as a whole are sampled further. Variance and average execution times for parallel loops are computed and maintained during program execution.

There are many possible choices for how to implement an adaptive sampling scheme

such as this. In practice, we found it was sufficient to simply sub-divide the base sample into four sections with equal numbers of samples and to compute the mean and variance of the execution times for each of these sections. If the product of the mean and variance for a particular section exceeded the scaled product of the mean and variance for the entire loop ($\alpha\mu\sigma^2$), we applied the adaptive sampling procedure recursively to this section by measuring additional samples from within this section's range of loop iteration indices.[2]

This adaptive sampling scheme works well because of a general property of parallel inner loops we found in our benchmark application programs: either the loop is only executed a few times and hence does not contribute to the overall runtime of the program, or it is executed so many times that the execution time of its first few execution instances is unimportant, because the cost of analysis during these first few iterations is amortized over the much longer total execution time for the application. The results we present in this chapter apply best to parallel applications that contain mostly these cases.

In the parallel programs that we have encountered so far, we have observed that, if a parallel loop executes only a few times but its execution time is significant, it contains nested loops that execute more frequently and whose work distribution can be cached through adaptive work distribution sampling. In this situation, we can obtain a very good picture of each parallel loop's work distribution using only a fraction of the execution time and memory required by a full sampling of the work distribution.

Scaled Task Transfer

Once a cost function can be established and maintained during the course of parallel program execution, it can be used by any dynamic scheduling method to avoid counterproductive task transfers, and hence to increase the efficiency of the scheduling method. The basic technique for exploiting a cost function is called *scaled task transfer*. Load balancing and self-scheduling methods share a common basis for measuring progress toward parallel loop completion: number of loop iterations remaining. They also share a method for estimating the amount of time it will take to execute a given set of loop iterations: the number of loop iterations in that set. As we have demonstrated above, these estimates are inaccurate when a parallel loop has significant loop iteration execution time variance.

A dynamic scheduling method can use a cost function to compensate for this inaccuracy by scaling loop iteration execution times. For instance, suppose that the mean loop iteration execution time for a parallel loop is μ_g. We call this value the *global mean* for the loop. Now suppose that a pair of processors, A and B have synchronized in order to determine whether a transfer of loop iterations should take place between the two processors. Processor A has executed all but three of its assigned iterations while processor B has eight iterations remaining.

In this situation, a cost function can have two benefits. First, both processors can use the cost function to compute how much *work* is remaining. In doing so, they might discover that the processor with fewer iterations remaining has the same or even more *work* left to perform, and can compute the correct amount of work that should be transferred, given the overhead of the transfer. Second, both processors can compare their local mean

[2]All measurements done for this dissertation used $\alpha = 2.0$

loop iteration execution times to the loop's global mean, μ_g. In doing so, they might discover that the amount of work remaining on *both* processors is insignificant so that a transfer is not warranted no matter what the disparity in work on the two processors. To make such determinations even more accurate, our prototype system periodically updates another value R_g which is the total remaining work that must be performed to complete the parallel loop.

3.3 Performance Improvement

Before presenting the particular set of scheduling methods used by our prototype system, we present the results of experiments which show that work distribution caching can benefit any dynamic scheduling method, at least as applied to our four benchmark applications. We modified our representative set of self-scheduling and load balancing methods to use a cost function, maintained at runtime using the work distribution caching techniques outlined above. For these measurements, we used the same runtime instrumentation system that we used to implement the dynamic scheduling methods described later in this chapter and in Chapter 4.

We modified each self-scheduling method to take advantage of the cost function by scaling the computation of chunk sizes. For example, SS uses chunk size 1. We changed the method so that each chunk contained μ_g work. We modified the load balancing methods so that K_{sync} represented an amount of work rather than a number of loop iterations to be performed before synchronization. K_{min} was similarly scaled. Further, we required each load balancing method to initiate a transfer only if it involved more work than $R_g/2p$, where p is the number of processors executing the parallel loop. In other words, if a transfer involved less than half the average remaining work per processor, it was disallowed.

Figure 3.3 shows how the canonical self-scheduling loop executed by each processor is changed to incorporate work distribution caching. The lines marked by an asterisk are new; the rest are identical to those in Figure 2.18. The other load balancing methods are modified analogously: work transfers are calculated in terms of *execution times* rather than *numbers of tasks*. These execution times are estimated using a loop iteration cost function built by adaptively sampling actual loop iteration execution times.

Figures 3.4 through 3.7 compare the efficiencies of some load balancing and self-scheduling methods with and without use of a cost function. They show that, in every case, applying a cost function to a previously proposed scheduling method improved the performance of that method on all four example programs. From these results, we conclude that simply applying a cost function to existing scheduling methods can substantially improve the efficiency of these methods.

3.4 Deferred Load Balancing

In this section, we present the dynamic scheduling method used by our prototype scheduling system: *deferred load balancing*. Three key ideas shaped the development of this method. First, we designed the system to make the best possible use of cost function information. Second, we made the assumption that any one execution instance of a parallel

```
FOR Timestep = 1 to Max
  WHILE iterations remain
      receive task transfer instructions
      receive global mean execution time info.    *
      initiate task transfers, if any
      compute K[i], the size of next chunk
      scale K[i] using cost function              *
      FOR I= StartChunk to StartChunk+K[i]
         if (I part of base sample)               *
            start execution time sampling         *
         if (condition on data)
           perform computation 1
         else perform computation 2
         if (I part of base sample)               *
            finish execution time sampling        *
            incorporate sample into base sample   *
      ENDFOR
      notify parent that chunk i is done
      notify parent of local mean execution time  *
  ENDWHILE
  synchronize with all other processors
ENDFOR
```

Figure 3.3: Canonical irregular parallel loop modified to perform self-scheduling on each processor of a distributed memory multiprocessor. Uses work distribution caching to scale chunk sizes.

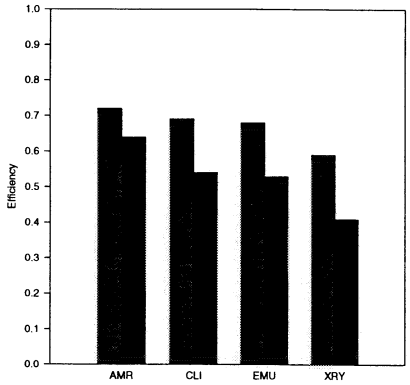


Figure 3.4: Factoring with and without work distribution caching on 512 processor Ncube-2 for each benchmark application (dark bar is without caching).

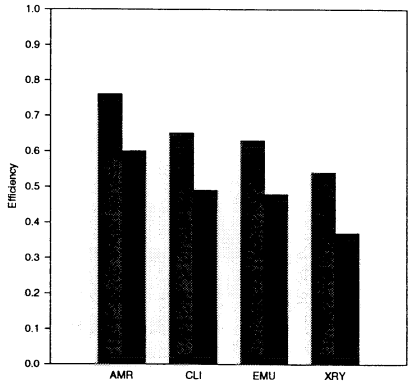


Figure 3.5: GSS with and without work distribution caching on 512 processor Ncube-2 for each benchmark application (dark bar is without caching).

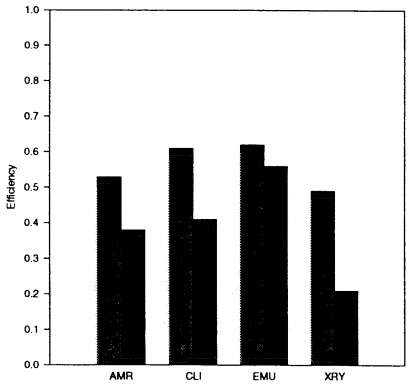


Figure 3.6: GL1 with and without work distribution caching on 512 processor Ncube-2 for each benchmark application (dark bar is without caching).

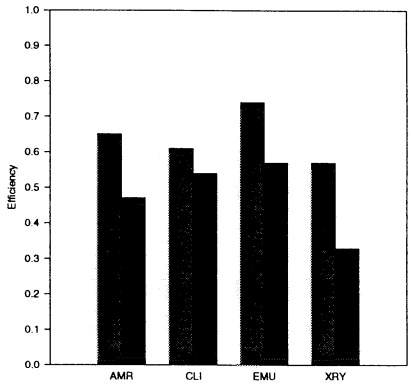


Figure 3.7: LLB with and without work distribution caching on Ncube-2 for each benchmark application (dark bar is without caching).

inner loop takes an insignificant amount of time compared to the total time required to execute the parallel application. Hence, deferred load balancing will sometimes sacrifice performance of a particular loop execution instance to collect work distribution information that may improve the execution of future instances. Further, deferred load balancing will initiate transfers of loop iterations during one execution instance, expecting that those transfers will only take effect during future instances. In doing so, our prototype took advantage of the Ncube-2 multiprocessor system's capability of overlapping communication with computation.

Finally, deferred load balancing avoids destroying communication locality by always transferring tasks along communication lines. For example, in the climate modeling application, communication takes place along a two-dimensional grid. Groups of loop iterations corresponding to sections of this grid are initially assigned to each processor (BLOCKED decomposition). As work distribution information becomes available, some processor's grid sections are augmented by transferring neighboring grid subsections from other processors. Because grid sections can change shape, more total communication overhead may be incurred. However, if the net effect of a transfer would be to delay loop completion because of increased communication cost, it will not be initiated.

The goal of deferred load balancing is to migrate tasks gradually among processors so that the *steady-state efficiency* of each parallel loop is optimized. The model underlying this technique is that the first few execution instances of a parallel loop will be inefficient, owing to the collection and communication of cost function information, and the use of cost function information in initiating loop iteration transfers.

The effect of this method is that each parallel loop reaches a stable level of efficiency called its *steady-state efficiency*. Regular loops and loops with strong work distribution locality (*coherent loops*) will have a high steady-state efficiency, and irregular loops with weak work distribution locality (*chaotic loops*) will have somewhat lower steady-state efficiency. Chapter 4 shows how to mask some of the inefficiency of chaotic loops by pipelining them with regular or coherent loops.

3.4.1 Implementation

Deferred load balancing works (DLB) works by constantly tuning the efficiency of a parallel loop until it reaches its steady-state efficiency. Once a loop has reached steady-state efficiency, DLB maintains efficiency by incrementally adjusting the allocation of loop iterations to processors. Our prototype DLB system assumes a fixed, regular communication pattern, based either on a grid or on a graph (in the case of the circuit simulator, EMU). The DLB system is provided with the information necessary for it to estimate the cost of communication between any pair of loop iterations, if those loop iterations are on separate processors. The DLB system is also provided with the information necessary for it to estimate the cost of migrating a particular group of loop iterations from one processor to another; this cost depends on the amount of data necessary to execute each loop iteration.

Ideally, this communication overhead information would be encoded for the runtime system by the high-level language compiler. In our prototype system, the information was provided manually, in the Delirium intermediate form. This intermediate form is described in Chapter 5. For the X-ray Tomography application, this information was produced

by our prototype FORTRAN compiler [38]. It would be straightforward to extend our FORTRAN compiler to handle the other applications as well. Once the information is specified in Delirium, the Delirium compiler translates it into a compact form for use by the runtime scheduling system.

The runtime scheduling system combines this communication cost information with the cached work distribution information it has collected for each loop. Processors are organized into a tree structure, similar to the organization described in Chapter 2. For each parallel loop, each processor sends summary information to its parent in the tree. Chapter 4 describes a grain-size selection algorithm called TAPER that is used to decide how often this information should be updated. Processors representing internal nodes of the tree evaluate the current assignment of loop iterations to each of their children. Whenever new information arrives at an internal node, it evaluates whether to initiate a transfer among two or more of its children.

To investigate whether to initiate a transfer, each node N performs the following decision procedure. First, it uses cost function information to estimate the amount of work on each child; it then sorts the children by estimated work completion time. If the completion time range among the children is greater than some constant α, then the node considers a transfer between the first and last children on the sorted list. For the measurements reported in this dissertation, we set α equal to 3% of the total estimated completion time for the parallel loop.

Once this threshold has been exceeded, the node N calculates the expected benefit, in improved load balance, of completing a transfer of work between the two children. This expected benefit is quantified as follows. First, N calculates the *local* load balancing benefit, β. β is defined as the percentage that the transfer would decrease the completion time of N's children. Second N calculates the *local* load balancing costs, γ_1 and γ_2. We define γ_1 to be the sum of two one-time costs: the cost of transferring the data necessary to execute the transferred loop iterations and the cost of the synchronization necessary to organize the transfer. We define γ_2 to be the *sustained* communication cost of the transfer; this is the time required to perform any extra communication that results in separating some loop iterations from their neighbors.

Some processor architectures, like the Ncube-2, allow coarse-grained overlapping of communication with computation. Such architectures support the implementation of deferred load balancing naturally, since much of the γ_1 and γ_2 costs are communication costs that can be overlapped with useful computational work. We have implemented deferred load balancing on some architectures such as the Thinking Machines CM-5. However, the lack of support of overlapped communication and computation on that architecture cost between ten and twenty percent in performance on the applications we tried.

For the remainder of this discussion, we assume that we can overlap communication with computation. We believe that this feature will become nearly universal among parallel processor architectures. In particular, the newest architectures from Thinking Machines, Intel, and Cray all support this feature. Even more important, networks of workstations, the fastest growing supercomputing platform, commonly support this feature.

Given the ability to overlap communication and computation, the limiting communication resource becomes the available bandwidth during any given execution instance

of a parallel loop. As long as the communication required to support the next execution instance can be completed during the current execution instance, then the cost of communication will have little effect on the completion time of each execution instance. This will be true as long as the bandwidth allocated to the parallel loop is sufficient to perform the necessary communication. In Chapter 4 we will present methods that divide communication bandwidth and processing resources among concurrently executing parallel loops. In this chapter, we will assume that we are dealing with a single parallel loop.

Given this assumption, the costs γ_1 and γ_2 are computed to be zero except for two cases. First, sufficient bandwidth may not be available, in which case the completion time of the current parallel loop execution instance will increase to accommodate the increased communication overhead. Second, the data communicated between two separated loop iterations may be generated only toward the completion of the current parallel loop execution instance. In this case, the communication will increase the completion time of the loop even though there is sufficient bandwidth during the entire loop to accommodate the communication. In Chapter 4 we will present some methods for eliminating this situation, by pipelining the execution of parallel loops. However, even when the best possible pipelining has been arranged, this situation can occur.

Once the local benefits and costs of a particular work transfer have been computed, the tree node N performs the transfer if the local benefits exceed the local costs, *and* if the local transfer is considered useful relative to the global situation. Two criteria determine the usefulness of a local transfer. First, the change in completion time among the node's children must be significant relative to the overall parallel loop execution time. Second, the communication bandwidth required by the transfer must be available; that is, not reserved by a more important transfer taking place elsewhere in the system. If these two criteria are met, N sends a message to both children, dictating the transfer. It also sends a message to its parent in the tree, reserving the communication bandwidth required for the transfer.

Transfers of work may span several execution instances of a parallel loop. That is, a transfer may be initiated during one execution instance and completed during a later execution instance. The work transfer only takes effect when all data necessary to execute the transferred loop iterations has arrived at the target processor, and the target processor has acknowledged its receipt of this data to the sending processor. The runtime scheduling system takes advantage of this asynchrony to piggyback work transfer messages with other messages whenever possible.

3.4.2 Performance

Next, we investigate the performance of the distributed load balancing (DLB) method. Figures 3.8 through 3.11 compare this performance with that of other scheduling methods. Each figure gives the performance of DLB, compared with the performance of the previously proposed scheduling method that performed best for the particular application measured. These results demonstrate that, for our four benchmark applications, DLB can achieve near-optimal efficiency on large numbers of processors. In addition, DLB performs significantly better, on these problems, than any previously proposed static or dynamic scheduling method.

The benefits of using deferred load balancing are greatest when parallelism is least

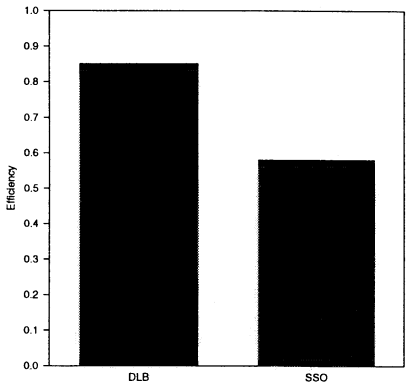


Figure 3.8: Performance of Deferred Load Balancing on climate model, compared to best previous approach.

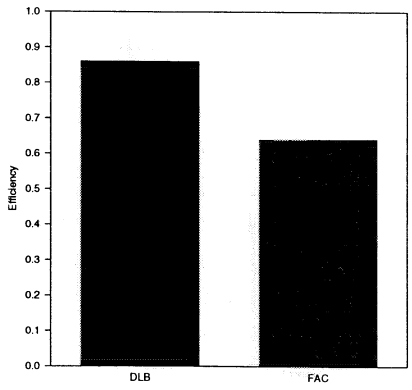


Figure 3.9: Performance of Deferred Load Balancing on AMR, compared to best previous approach.

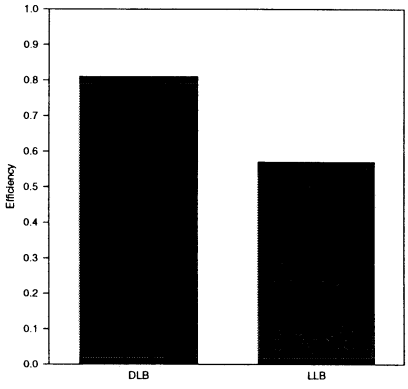


Figure 3.10: Performance of Deferred Load Balancing on EMU, compared to best previous approach.

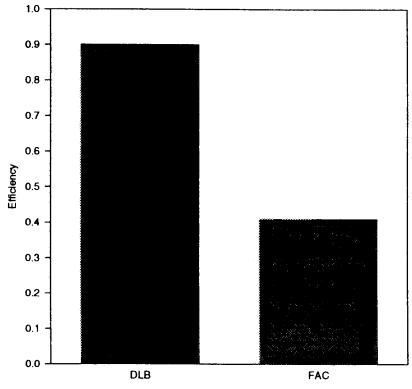


Figure 3.11: Performance of Deferred Load Balancing on Psirrfan, compared to best previous approach.

abundant. This is because, as the number of processors applied to a given loop increases, the danger of thrashing toward completion of the loop increases correspondingly. By amortizing task transfer costs across multiple parallel loop execution instances, and by using cached work distributions to guide scheduling decisions, DLB achieves high efficiency.

Chapter 4

Scheduling

The previous chapter described our main techniques for adaptively scheduling irregular parallel programs: work distribution caching and deferred load balancing. In this chapter, we describe two other techniques that play important supporting roles in achieving efficient execution of irregular parallel programs. First, we describe *probabilistic grain size selection*, a method for determining how much work each processor should perform before synchronizing with other processors. Second, we present techniques for orchestrating the interactions among concurrently executing parallel loops.

4.1 Tapering Methods

We present a probabilistic method called TAPER for choosing an appropriate grain size at which to schedule chunks (groups) of tasks in a parallel program. TAPER can be used as a shared memory self-scheduling algorithm, similar to GSS or SS, and it can be extended to distributed memory using the techniques described in Chapter 2. Our deferred load balancing method uses TAPER to determine how often processors should exchange execution information about a particular loop. In this context, each processor uses TAPER to select a chunk size. It then executes the number of loop iterations in the chunk before updating its parent processor with information about the number of loop iterations that remain and the changes to the cost function.

The remainder of the discussion will be in terms of groups of *tasks* that can execute in parallel. The typical source for such a group of tasks is a parallel loop; each loop iteration is considered a separate task. TAPER is a method for selecting a chunk size K_i, given the number of remaining tasks R_i, number of processors p, and work distribution parameters(μ,σ). The method selects K_i using an expression of the form $K_i = \max\left(K_{min}, f(\frac{\sigma}{\mu}, K_{min}, \frac{R_i}{p}, h)\right)$, where K_{min} is the minimum chunk size and f is a function we will derive below. This expression yields GSS as a special case when $\sigma = 0$.

The goal of any grain size selection technique is to achieve optimally even finishing times while scheduling the smallest possible number of chunks. When scheduling the i^{th} chunk, we would like to pick the largest possible number of tasks K_i such that our expected maximum finishing time does not increase. To estimate K_i, we define $finish_{i,j}$ to be the time at which processor j has finished with any chunks numbered less than or equal to

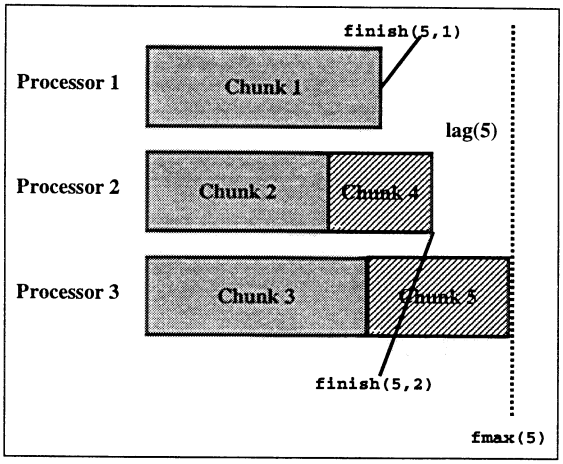


Figure 4.1: A Scheduling Scenario

i. We let $fmax_i = \max_j(finish_{i,j})$, and $lag_i = \sum_j(fmax_i - finish_{i,j})$ (lag_0 is the initial unevenness in processor start times). If we schedule n chunks then $fmax_n$ is the finishing time of the computation, and lag_n measures the total amount of processor idle time (the inefficiency) during the computation (see Figure 4.1 for an example).

4.1.1 The Fill Lemma

Using these definitions, we can derive a grain-size selection algorithm that provides a near-optimal choice for K_i. To support this derivation, we introduce a lemma that illustrates a key property of the first available rule. The first available rule states that the next group of tasks to be scheduled will be assigned to the first processor available. The goal of the Fill Lemma is to show how lag_{i+1} depends on lag_i. That is, given a particular unevenness in finishing times after scheduling chunk i, the lemma tells whether the scheduling of chunk $i+1$ will cause smoothing or increase unevenness, and by how much. In the following, $cost_i$ denotes the execution time of chunk i (note that the size of all chunks could be 1, so this lemma applies to the scheduling of individual tasks).

Lemma 1 If $fmax_{i+1} = fmax_i$ then $lag_{i+1} = lag_i - cost_{i+1}$. Otherwise, $lag_{i+1} \leq (p-1)cost_{i+1}$.

Proof. **Case I:** $fmax_{i+1} = fmax_i$:
The condition for this case states that the maximum finishing time does not increase as a result of the scheduling of chunk $i+1$. But, if this is true, there is some processor j such that $finish_{i+1,j} = finish_{i,j} + cost_{i+1}$ and $finish_{i+1,j} \leq fmax_i$. If chunk $i+1$ is assigned to processor j then for $k \neq j, finish_{i+1,k} = finish_{i,k}$. Thus,

$$\begin{aligned} lag_{i+1} &= \sum_{k \neq j}(fmax_i - finish_{i,k}) + \\ &\quad fmax_i - (finish_{i,j} + cost_{i+1}) \\ &= lag_i - cost_{i+1} \end{aligned}$$

□

Case II: $fmax_{i+1} > fmax_i$:
When the maximum finishing time, $fmax_{i+1}$ *does* increase, then there is some processor q such that $finished_{i,q} = \min_j(finished_{i,j})$ and $finished_{i+1,q} = finished_{i,q} + cost_{i+1} = fmax_{i+1}$. Since $finished_{i,q}$ is the minimum finishing time after chunk i and only processor q's finish time changes as a result of the scheduling of chunk $i+1$, $\sum_j(fmax_{i+1} - finish_{i,j}) \leq (p-1)cost_{i+1}\Box$.

We call Case I the *fill-in* rule, and Case II the *excess* rule. The point of this lemma is that, whenever the excess rule applies (*i.e.* $fmax_{i+1} > fmax_i$), we can bound lag_{i+1} independent of lag_i.

Corollary 1 *Let l be the index of the last chunk for which $fmax_l > fmax_{l-1}$. Then we can bound lag_n independent of $lag_{i|i<l}$. Further,*

$$lag_n = lag_l - \sum_{j=l+1}^{n} cost_j$$

Corollary 1 suggests that our analysis need only consider the state of the computation from the last time the excess rule applies. Corollary 2 gives us a procedure for deciding when a chunk can be the last chunk for which the excess rule applies.

Corollary 2 *If $(lag_i - \sum_{j=i+1}^{n} cost_j) < 0$ then chunk i can not be the last chunk for which the excess rule applies.*

4.1.2 Probabilistic Tapering

We can use the fill lemma as the basis for a probabilistic tapering method. Given execution time variance, we cannot know the exact time necessary to execute any task or group of tasks. Therefore, given a bound b that expresses the maximum allowable inefficiency, we seek a scheduling method that makes $lag_n < b$ with high probability. We will express the new scheduling method as a rule for computing a set of chunk sizes $K_1, \ldots, K_n$ such that, if tasks are handed out with these chunk sizes, $lag_n < b$ with high probability.

We denote as $fill_i$ the term $\sum_{j=i+1}^{n} cost_j$ in Corollary 2 and assert a final corollary to Lemma 1. This corollary is what makes a probabilistic analysis of tapering methods tractable.

Corollary 3 $lag_n \leq \max_i((p-1)cost_i - fill_i)$

We can use this expression to limit the effect of each chunk K_i on lag_n such that lag_n is less than the given bound b (b is the maximum allowable inefficiency) with high probability. Suppose that at time i, there are R_i tasks remaining and that our tapering method chooses to schedule a chunk containing K_i tasks. Then we can rewrite corollary 3 as $lag_n \leq \max_i((p-1)t(K_i) - t(R_{i+1}))$, where $t(K_i)$ and $t(R_{i+1})$ are random variables representing the actual time taken to execute K_i and R_{i+1} tasks, respectively. In other words, we assume that the tasks are chosen arbitrarily and that their execution time varies according to some distribution t.

Let $Z_i = (p-1)t(K_i) - t(R_{i+1})$. We would like to find the largest value of K_i such that $\Pr[Z_i > b] < \epsilon$ for a given bound b and probability ϵ. Using Chebychev's inequality we

find $\Pr[Z_i - \mathcal{E}(Z_i) > a] < \sigma_{Z_i}^2/a^2$, where $\sigma_{Z_i}^2$ is the variance of Z_i and $\mathcal{E}(Z_i)$ is the expected value of Z_i. If we let $a = b - \mathcal{E}(Z_i)$, then we have $\Pr[Z_i > b] < \sigma_{Z_i}^2/(b - \mathcal{E}(Z_i))^2$. Let $\Pr[Z_i > b] = \epsilon$. To obtain an expression in terms of the original scheduling parameters, we eliminate Z_i from our derived inequality by substituting $\mathcal{E}(Z_i) = (pK_i - R_i)\mu$ (using $R_{i+1} = R_i - K_i$) and $\sigma_{Z_i}^2 = \sigma^2(((p-1)^2 - 1)K_i + R_i)$ (where σ^2 and μ are the variance and mean of the original task cost distribution). This substitution yields

$$\epsilon < \sigma^2(((p-1)^2 - 1)K_i + R_i)/(b - (pK_i - R_i)\mu)^2 \tag{4.1}$$

The partial derivative of (4.1)'s right hand side with respect to K_i is always positive. Thus, if we set ϵ equal to the expression on the right-hand side of equation (4.1) and solve for K_i, we will find the largest K_i such that $((p-1)cost_i - fill_i) < b$ with probability at least $1 - \epsilon$.

A Practical Algorithm Chebychev's inequality is valid for all distributions. In practice, this means that it yields a needlessly conservative value for K_i. Note that both $t(K_i)$ and $t(R_{i+1})$ are sums of individual task costs. By the Central Limit Theorem, the sum of K independent variates approaches a normal distribution as K increases. For the distributions found in irregular programs (normal, multinomial, uniform, exponential) a normal distribution is a very good approximation, even for small values of K.

Our current runtime system uses the following method for choosing K_i, based on the assumption that the distribution of Z_i is normal. Using a table we can find a value α such that $\Pr[Z_i - \mathcal{E}(Z_i) > \alpha\sigma_{Z_i}] < \epsilon$ for a given value of ϵ. Put simply, we are guessing that Z_i will not exceed its expected value by more that α standard deviations. Since we want $\Pr[Z_i > b] < \epsilon$ for some bound b, we have $\mathcal{E}(Z_i) + \alpha\sigma_{Z_i} = b$. Substituting for $\mathcal{E}(Z_i)$ and σ_{Z_i} as before and letting $b = 0$ we have

$$(R_i - pK_i)\mu = \alpha\sigma\sqrt{((p-1)^2 - 1)K_i + R_i} \tag{4.2}$$

Let $T_i = R_i/p$, and $v_\alpha = \alpha\sigma/\mu$. Then (4.2) becomes $T_i - K_i = v_\alpha\sqrt{\frac{((p-1)^2-1)}{p^2}K_i + \frac{R_i}{p^2}}$. Intuitively, K_i is less than $T_i = R_i/p$ by an amount related to the variance of the work distribution. If we approximate $((p-1)^2 - 1)/p^2$ as 1 and R_i/p^2 as 0 (since the other term under the radical, K_i, will be close to R_i/p and hence much larger), then solving for K_i yields

$$K_i = \left\lceil T_i + \frac{v_\alpha^2}{2} - v_\alpha\sqrt{2T_i + v_\alpha^2/4} \right\rceil \tag{4.3}$$

Figure 4.2 shows how equation (4.3) maintains the invariant $\Pr[Z_i > 0] < \epsilon$. Z_i is the difference between two random variables $(p-1)t(K_i)$ and $t(R_{i+1})$. Expression (4.3) sets the distance between the means of these two variables to be some number of standard deviations, specifically α. When $\sigma = 0$ then the expression sets the means to be the same value: $(p-1)R_i/p$ (like GSS). As the variance increases the relationship of equation (4.3) increases the distance between the means to maintain the invariant.

Thus, for a given scheduling event i, we can ensure that $\Pr[Z_i > 0] < \epsilon$ for any ϵ by selecting the corresponding number of standard deviations, α, from a table. However,

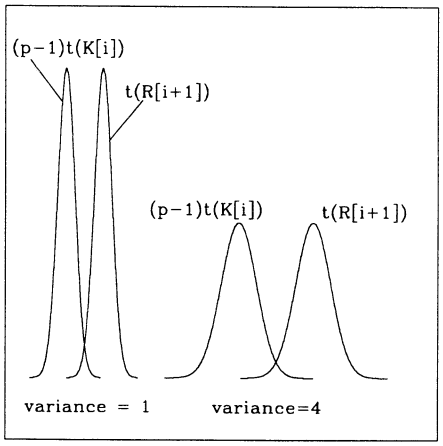


Figure 4.2: Illustration of K_i selection.

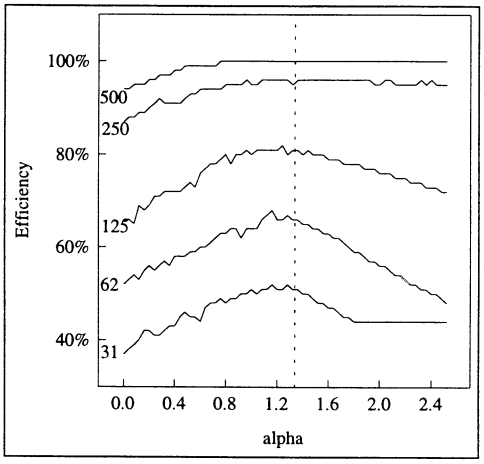


Figure 4.3: Effect of α. Lines labeled with N/p.

we would like to derive a single value of α for the entire parallel operation. To do this we require an expression that, given ϵ, yields $\Pr[\max_i(Z_i) > b]$. It is an open question whether such an expression can be found; without it, determining the best value for α is analytically intractable.

In practice, it is possible to discover a sufficiently accurate value for α empirically. Two parameters, the scheduling overhead (h) and the ratio of tasks to processors (N/p) can affect the optimum value of α. Using a normal distribution, we measured the optimum value of α over the entire possible range of both parameters, varying h from 0 to ∞ and N/p from 1 to ∞. Figure 4.3 summarizes the results for some typical values of N/p and with $h = 1$. Over all combinations of h and N/p, we found that the value $\alpha = 1.3$ was within 3% of optimum. All of the performance results reported in this paper were obtained using this single value for α.

Figure 4.3 also indicates that a scheduling method using equation (4.3) can withstand considerable inaccuracy in the value of $\frac{\sigma}{\mu}$, since v_α in equation (4.3) is just the scaled

coefficient of variation $\alpha\frac{\sigma}{\mu}$. For example, we found that simply using equation (4.3) and setting $v_\alpha = 3$ yielded better performance than the other scheduling methods tested. In all cases, runtime measurement of $\frac{\sigma}{\mu}$ further improved performance.

Incorporating Overhead Equation (4.3) implicitly addresses overhead by selecting the largest number of tasks K_i that meets the constraint $\Pr[Z_i > 0] < \epsilon$. If we explicitly account for scheduling overhead, we can improve our value for K_i. We represent overhead through the parameter K_{min}, the minimum chunk size. To determine K_{min}, we use two additional parameters: K_{band} and K_{sched}.

We noted in Chapter 3 that the compiler computes the minimum chunk size, K_{band}, necessary to ensure that communication between processors containing logically adjacent chunks does not exceed the bandwidth requirements of the machine (K_{band} is set to zero on shared memory multiprocessors). The other parameter, K_{sched}, is smallest chunk size such that the mean time to execute K_{sched} tasks exceeds h, the overhead of scheduling a chunk. We set $K_{min} = \min(K_{band}, K_{sched}, N/p)$. If we are using TAPER to compute chunk sizes for deferred load balancing, we set K_{band} to zero, because communication overhead is taken into account by the higher-level scheduling decision algorithm. Hence, the only interesting parameter is K_{sched} the cost of updating scheduling information upon the completion of a group of tasks.

When $K_{min} = N/p$, we can't improve on a static schedule for the parallel operation. However, we can modify equation (4.3) such that it yields a dynamic scheduling method that outperforms static scheduling whenever $K_{min} < N/p$. We know that if all chunks contain K_{min} or more tasks, the average value for lag_n will be at least $K_{min}p\mu/2$. Hence, there is no reason to require $Z_i \leq 0$ and therefore the bound on acceptable lag_n $b = 0$ in the derivation above. If we set $b = K_{min}p\mu/2$, then equation (4.3) still holds, provided we set $T_i = \frac{R_i}{p} + K_{min}/2$. Our final expression for computing K_i becomes

$$K_i = \max\left(K_{min}, \left\lceil T_i + \frac{v_\alpha^2}{2} - v_\alpha\sqrt{2T_i + v_\alpha^2/4}\right\rceil\right) \tag{4.4}$$

Combining our techniques for selecting α and K_{min} with equation (4.4) we have an algorithm for dynamic scheduling. We call the algorithm TAPER. Figures 4.4 through 4.6 compare the performance of TAPER with guided self-scheduling (GSS), self-scheduling (SS), and static assignment (SA) for some synthetic work distributions.

Even Starting Times The derivation and simulation results given above demonstrate that TAPER yields a near-optimal schedule assuming only $lag_0 < N$ (*i.e.* processors can start at different times) and that task costs are independent random variables. Further, TAPER is stateless. Adding tasks only increases $fill_i$; removing processors only decreases $(p-1)t(K_i)$.

If the scheduling algorithm is given additional information about processor starting times, it can do better. For example, suppose we know that p processors will execute the entire parallel operation and that all processors start at the same time. We can use this information to choose larger chunk sizes than TAPER. Let s_i be the time at which the i^{th}

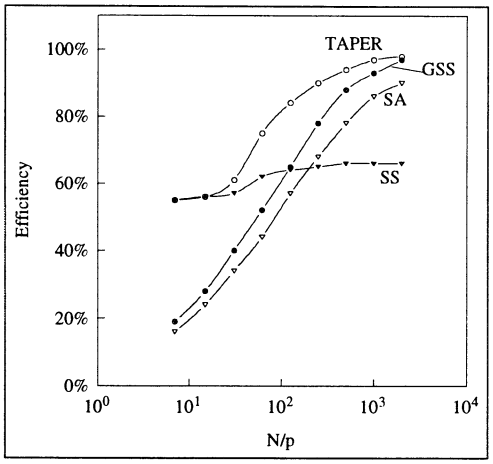


Figure 4.4: Binomial Distribution Pr[60000]=0.1, Pr[200]=0.9

chunk is assigned. Let $D_i = \frac{N}{p} - \frac{s_i}{\mu}$. D_i is the distance in tasks between s_i and the expected finishing time of the computation. Taking into account the variance in possible finishing times yields $K_i = \max(K_{min}, \lceil D_i - v_\alpha \sqrt{D_i} \rceil)$.

We call this method for selecting K_i the DISTANCE method. The difference between DISTANCE and TAPER is greatest for the first p chunks scheduled. Further, it is expensive to maintain globally meaningful values for s_i. For these reasons, we use a hybrid method (called EVENSTART) in situations where all processors begin simultaneously. This method uses DISTANCE for the first p chunks and TAPER thereafter. Figure 4.7 illustrates how EVENSTART maintains a small unevenness in chunk finish times throughout a parallel operation.

Determining the Distribution When profiling information is not sufficient to provide values for μ and σ, we need to discover these values at runtime. This is done by having each processor randomly select a few tasks from its large initial chunk and measure the execution times for these tasks. This information is accumulated as processors request later chunks. We therefore need a method to pick the first p chunk sizes. One technique is to use $K_i = N/2p$ for the first p chunks, since TAPER will always allocate more than half the work in the first p chunks. Hummel and Shoenberg have proposed a similar scheduling rule based on a different analysis [30]. This method is excessively conservative for large N/p. We chose instead to estimate that $\frac{\sigma}{\mu} = 3$. This *ad hoc* technique works well in practice because the estimate is quickly updated on each processor as sampling information accumulates.

4.2 Orchestrating Interactions Among Parallel Computations

Finally, we describe our methodology for orchestrating interactions among parallel computations. Many parallel programs contain multiple sub-computations, each with distinct communication and load balancing requirements. The traditional approach to

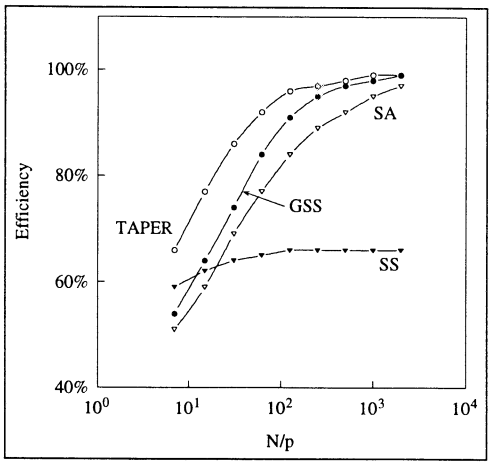


Figure 4.5: Uniform Distribution on [0,10]. Overhead = 0.5μ.

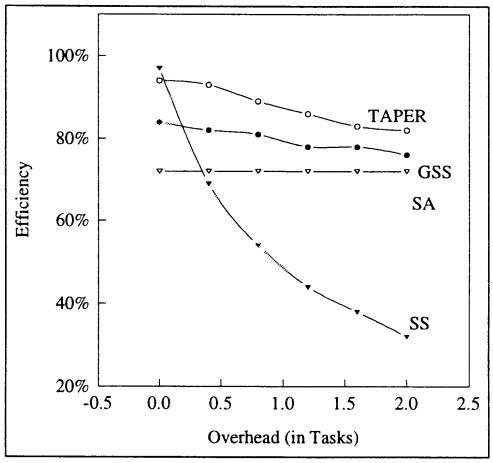


Figure 4.6: Performance on Pr[10] = 0.9, Pr[1] = 0.1 for different overheads.

compiling such programs is to impose a processor synchronization barrier between sub-computations, optimizing each as a separate entity. In this section, we describe a methodology for managing the interactions among sub-computations, avoiding strict synchronization where concurrent or pipelined relationships are possible. Several application studies [21, 41, 55] have identified such interactions as a key target for manual optimization. The contribution of this dissertation is to automate the runtime support for these optimizations.

For example, suppose we have two loop nests with the potential to execute concurrently. If each computation is handled independently, with synchronization between the two loop nests, efficiency of CPU usage is limited by the less regular of the two. One possible remedy is to use loop fusion, coalescing the two loops into one. However, the resulting parallelization is incomplete, since fusion discards information about the more regular component of the new loop. With the computations separated, a runtime system can use the additional parallelism from the more regular loop nest to smooth the load balance of the

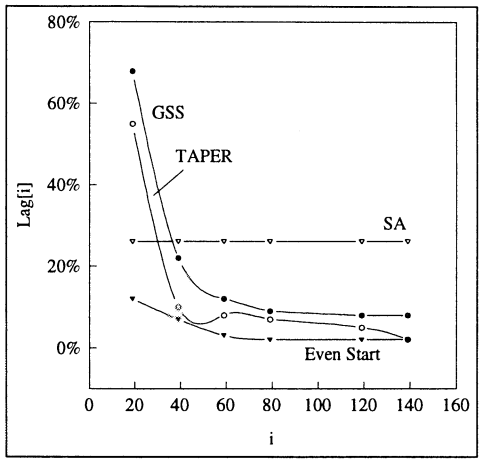


Figure 4.7: As chunks are scheduled, lag_i decreases. SA denotes static assignment of tasks to processors.

computation as a whole.

Our research into the runtime optimization of loop nest interactions was done in the context of a FORTRAN 77 compiler being developed by the Berkeley Coordination project. On the PSIRRFAN program, we used this compiler to expose interactions among parallel loop nests.

To expose this type of interaction automatically, we use a new method for data access summarization called a *symbolic data descriptor* [24]. We use symbolic data descriptors to implement a key transformation, called *split*, which reduces synchronization constraints by sub-dividing computations. We have incorporated *split* into our compiler, which outputs FORTRAN 77 augmented with both calls to library routines written in C and a coarse-grained dataflow graph summarizing the exposed parallelism.

The compiler encodes symbolic information such as loop bounds and data sizes. The runtime system uses this information, as well as information gathered from the running program, to guide two key scheduling decisions: grain-size selection and processor allocation. The grain-size selection algorithm is detailed elsewhere [38]. In this section, we present the runtime processor allocation algorithm, which uses sub-computation finishing time estimates to ration processing resources correctly among concurrently executing sub-computations.

4.2.1 Example Interaction

Figure 4.8 shows interaction among sub-computations. It is a loop whose induction variable `col` iterates over the columns of a data array `q`. If the associated element of the array `mask` is non-zero, the iteration performs the computation A. A reads all of `q` and modifies column `col`. After the loop is completed, B computes the array `output` from `q`.

Because of the conditional in A, the compiler cannot determine an efficient schedule for A statically. Depending on the values in that array, adaptive techniques may find an efficient schedule. However, on large numbers of processors, efficiency may still be poor if there is not enough parallelism (i.e. too few mask elements are non-zero) or if the time to

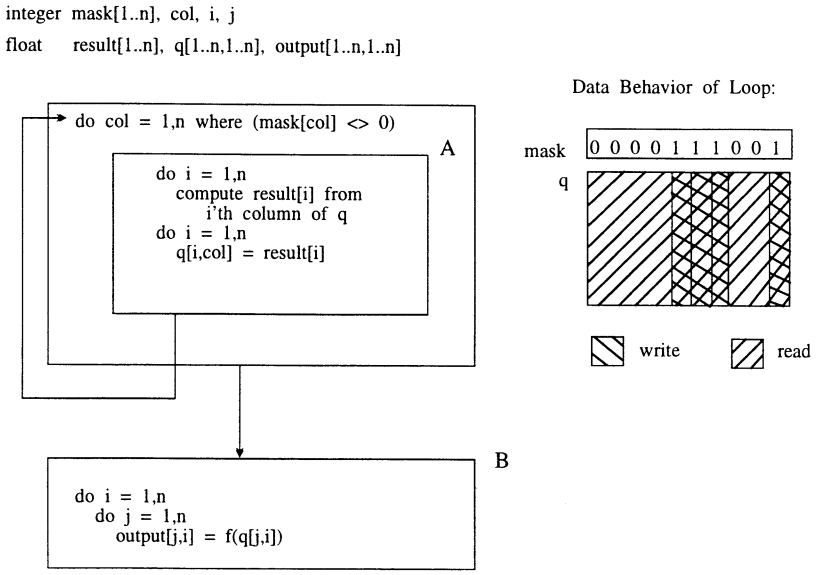


Figure 4.8: Code Before Split

process each column varies too widely.

Our approach is to transform the code to expose further concurrency; there are three sources that our strategy can reveal. The first of these is to divide B into three pieces, B_I, B_D, and B_M, where B_I processes the columns of q which are not touched by any of the instances of A and B_D processes the rest. B_M merges the results into a single output array. In Figure 4.9, we show the effect of applying this transformation.

The notation is equivalent to

```
do ...  where <expr>        do ...
   loop body                 if (<expr>)
                               loop body
```

For clarity, we show an explicit merge of the two output arrays in B_M. In practice, merging can often be handled implicitly by the runtime system during data communication.

The second transformation we can apply is to pipeline one iteration of the col loop with subsequent iterations. We show the code with this further optimization in Figure 4.10. The body of the loop has been converted into three computations A_D, A_I, and A_M. A_D represents the code that is dependent on the previous iteration of the loop; the runtime system waits for the previous iteration to complete before scheduling A_D. A_I, on the other hand, is independent of the previous iteration and can be scheduled concurrently. By weakening the synchronization constraint between iterations, we are able to pipeline them.

A_I computes the result vector for all but the missing column of q that comes from the previous iteration; A_D computes the one missing element into the variable prev_val. A_M takes the almost complete vector and the missing value and combines them into a single vector. Again, this kind of merge can often be done implicitly. Note that we use the form do var=<range> and <range> to denote a discontinuous sequence of values, rather than

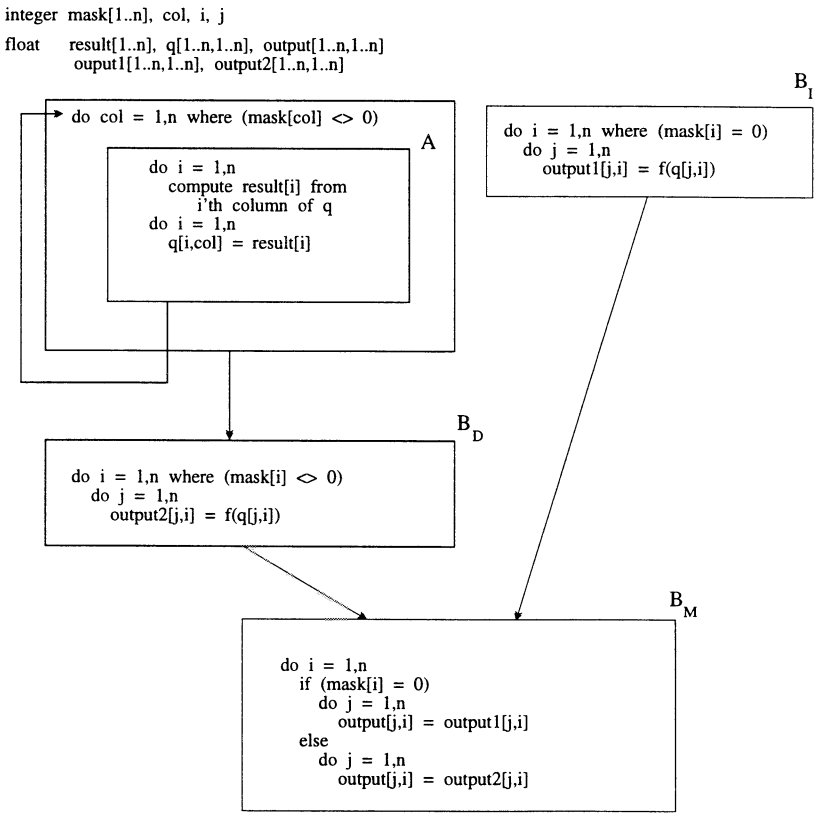


Figure 4.9: Code After Split

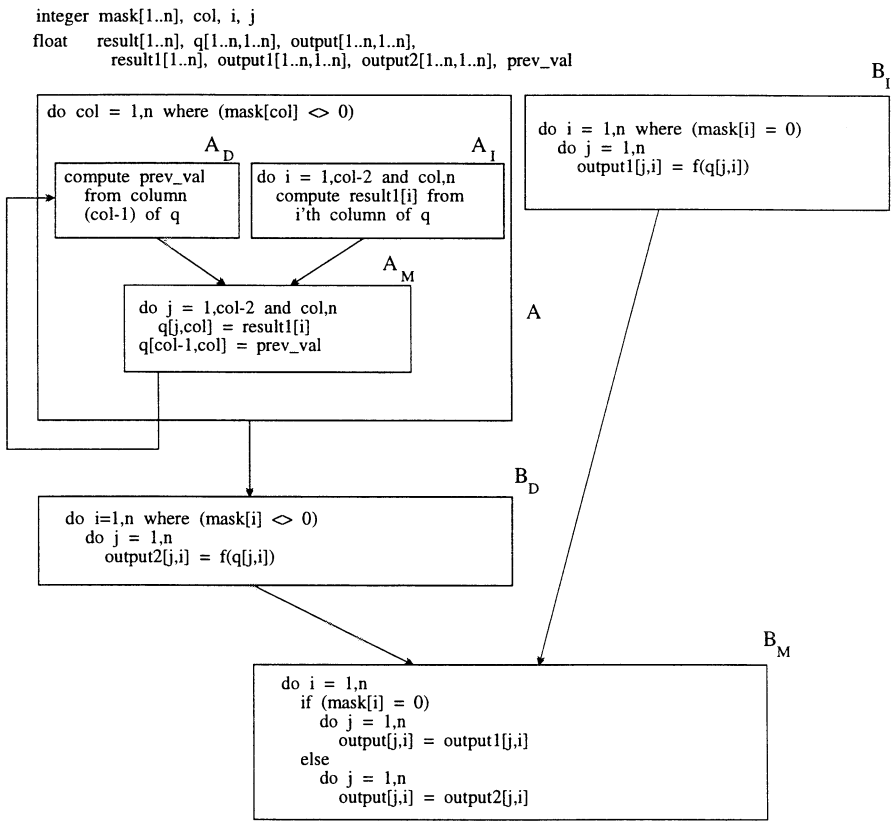


Figure 4.10: Code After Split and Pipeline

duplicating the entire loop for both ranges.

The third transformation which we could perform is to pipeline iterations of A with corresponding iterations of B_D, exposing a further source of concurrency.

The form of pipelining described in these last two transformations is different from loop pipelining optimizations defined elsewhere. Previous approaches fall into two classes. The first is *software pipelining* [35], which seeks to reduce overhead by reorganizing the code. The transformed loop performs fewer iterations but has a larger body that handles more than one of the original loop's iterations at once. This strategy works well in improving performance of a regular loop nest. The second approach, suggested by Balasundaran and Kennedy [9], uses post and wait primitives to allow more than one loop iteration to execute concurrently. Since this strategy imposes a fixed synchronization discipline, it does not admit adaptive scheduling techniques.

4.2.2 The Runtime Algorithm

Consider a runtime scenario in which the transformed parallel operations A and B_I from Figure 4.10 are executing simultaneously. A begins executing first and has partially completed when B_I begins executing. The runtime system must decide how many processors to reallocate. If we change the example so that all processors must synchronize upon completion of A and B_I, an ideal processor allocation would minimize the expected finishing time of these two parallel operations.

The expected finishing time of a parallel operation is a function of the number of tasks that make up the operation (N), the number of processors cooperating to execute the operation (p), the variance in task execution times, and the overhead of scheduling [34, 38]. Thus, approximating the ideal processor allocation requires information available only at runtime.

For this reason, we extended adaptive algorithms developed for single irregular parallel operations to manage interactions among multiple, simultaneously executing parallel operations. There are three main extensions. First, we developed a method for improving the accuracy of estimated finishing times that works for a wide range of scheduling algorithms. Second, we applied this method to the runtime processor allocation problem, using an iterative algorithm to equalize finishing time estimates. Finally, we combined finishing time estimates with runtime communication cost estimates to choose communication granularity for pairs of pipelined parallel operations.

Now, we return to our runtime scenario in which the parallel operations A and B_I are executing concurrently. When B_I begins executing, the runtime system must reallocate some of the p processors executing A. To accomplish this, the runtime system uses the following iterative algorithm:

$epsilon = 5\%$
$p1 = p/2$, $p2 = p - p1$, $count = 0$
$eA =$ finish_estimate$(A, p1)$, $eB =$ finish_estimate$(B, p2)$
while $((count < max\_count)$ and $(|eA - eB| > epsilon))$
    if $(eA > eB)$
      $p1 = p1 + f(p2)$

$p2 = p - p1$
else
$p2 = p2 + f(p1)$
$p1 = p - p2$
$eA = \text{finish\_estimate}(A, p1)$
$eB = \text{finish\_estimate}(B, p2)$
$count = count + 1$

We limit the number of iterations to control the amount of overhead imposed. In practice, using a *max_count* of four has been sufficient.

In estimating finishing time, the runtime system uses the expression:

$$finish = setup + compute + lag + comm + sched \tag{4.5}$$

setup is the maximum of the time to contract the data required by A onto p_1 processors and the time to expand the data required by B_I onto p_2 processors. *compute* is the expected mean time to perform a portion of the computation: $N\mu/p'$ (where $p' = p_1$ for A and $p' = p_2$ for B_I). *lag* is the expected maximum finishing time to perform a portion of the computation; it is related to the distribution of task execution times for the computation (μ,σ) [38]. *comm* represents the communication overhead of executing the given parallel operation on p' processors.

To estimate *comm* at runtime, we use an algorithm like that suggested by Sarkar and Hennessy [53], which performs a weighted sum of dataflow graph edges that cross processor boundaries. Rather than perform this computation statically, the Delirium compiler generates code blocks that perform the estimate given runtime parameters such as N and p'.

Finally, we must estimate the scheduling overhead (*sched*). To do so, we need to predict, at runtime, the number of chunks that will be scheduled for the parallel operation (hence the number of epochs in the distributed algorithm given above). The method for predicting this parameter is discussed elsewhere [38]; it applies to TAPER and many other algorithms for generating chunk sizes [29, 49, 58].

By balancing the estimated finishing times of A and B_I, the runtime system uses the extra concurrency from B_I to compensate for A's irregular execution behavior.

Chapter 5

System Support

This chapter outlines the system support that we used to implement our adaptive scheduling techniques. We first describe Delirium, a coordination language that we used as a compiler intermediate form. Delirium includes constructs designed to simplify the expression of data parallelism. We used Delirium in two ways. For the Climate, AVM, and EMU examples, we hand-compiled the programs, creating two forms of output: a set of FORTRAN and C computational functions, and a Delirium "glue" program to coordinate the execution of these functions. For PSIRRFAN we produced the same type of output, but used an optimizing FORTRAN compiler that we designed and implemented for this purpose. We believe that the compilation of our other examples can also be automated and have described algorithms for doing so elsewhere [24]. Many of the transformations required have already been incorporated into our FORTRAN compiler. Some of the information used by our runtime system for the performance measurements given in this dissertation was gathered by compilation. For example, information about communication costs and patterns are gathered by the FORTRAN compiler and stored as *thunks* in the resulting program. The runtime system evaluates these thunks, using runtime parameters such as loop bounds, to include information about communication locality and cost in its load balancing decisions.

The second part of this chapter describes the design and implementation of our runtime scheduling system. The core functionality of this system is implemented using Tarmac [39], a library that supports replication and location-independent modification of distributed state.

5.1 The Compiler Intermediate Form

Our compiler intermediate form consists of a simple language called Delirium. Using Delirium, our FORTRAN compiler can express the division of a program into sub-computations; it can also express any concurrency that might be possible among sub-computations. Finally, to support the organization of communication on distributed memory multiprocessors Delirium expresses how data will flow between sub-computations.

Delirium is designed to concisely express the multi-dimensional dataflow that can exist in parallel computations. Multi-dimensional dataflow arises out of data parallel operations. For example, in our climate modeling application, the hydrodynamics of the atmo-

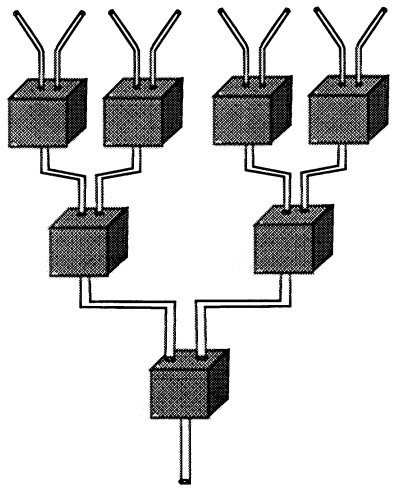


Figure 5.1: Coordination Structure for Mergesort.

sphere are computed using a 3-dimensional pattern of neighboring grid points. To compute the new value of quantities such as temperature and pressure at a particular point, the neighbors of that point are sampled.

Delirium's support for concise expression of data parallelism is based on *coordination structures*. A coordination structure is a (structured) collection of *coordination items*. Coordination items can be understood as individual ordering dependencies within a program [42]. Imagine the dataflow graph for the following expression:

```
let x = f(<expr1>)
in g(x)
```

In a normal strict functional language, `x` is a name that corresponds to the result of evaluating the application of `f` to `<expr1>`. A different way to understand `x`, however, is that it expresses an ordering dependency. To evaluate the application of `g`, one must first have evaluated the application of `f`. Think of `x` as a pipe that connects an application of `f` to an application of `g`, through which data can flow. Each such pipe is a coordination item.

A parallel computation can be expressed as a multi-stage pipeline of coordination structures. At each stage in the pipeline, a function is applied to the data flowing through each member in the collection of data pipes. Following a function application, the order of items (data pipes) within the coordination structure may be permuted to create the appropriate data organization for the next stage of the pipeline.

For example, a useful primitive for many parallel algorithms is binary reduction. A binary reduction takes N data items and applies some associative binary operation to successive pairs, yielding a group of $N/2$ results. The same operation is performed repeatedly until there is only one value left. Many algorithms are based on binary reduction, including, for example, merge sort. The pipeline for merge sort is shown in Figure 5.1 and

consists of $log(N)$ applications of the merge operator. If the original N values flow into the pipeline as a vector of pipes, the first step is to divide the pipes into $N/2$ pairs. Next, the program applies an instance of the merge operator to each of these pairs in parallel. One pipe flows out of each merge operator, so the cross section of the pipeline has been reduced to $N/2$ coordination items. The grouping and function application are done repeatedly, until at the end only a single pipe flows out of the pipeline and it contains the sorted list.

It is important to note that the coordination structure of the application as a whole looks like a tree, even though the data that moves through the pipes is probably organized into an entirely different data structure (like a list). We believe that an algorithm is much clearer if the coordination structure is linguistically decoupled from the underlying data structure. A binary reduction always looks like a tree, regardless of whether the structure being operated on is a set, a list, a tree, or an array.

Section 5.1.3 describes an application with a complex coordination structure and shows a Delirium realization of this structure. Many classes of algorithms have good synchronous solutions which can be expressed as coordination structures. A few examples are: wavefront algorithms (including many types of dynamic programming), algorithms based on communication over trees (including the Delirium compiler [54]), and algorithms based on convolutions over grids (including Laplace's equation and successive over-relaxation [50]). A significant proportion of numerical scientific programs fall into one of these categories [5].

Because coordination structures directly support data parallel operations, they encourage a programming style that incorporates techniques found in SIMD programs. However, SIMD architectures impose synchronization requirements that narrow their application domain.

5.1.1 Delirium

Existing coordination languages, such as Linda [14] and Sloop [37], are *embedded*; they consist of a set of asynchronous coordination primitives that are accessed through statements scattered throughout a host language program.

Delirium is the first example of an *embedding* coordination language [41]. We call it an embedding language because a Delirium program specifies a framework for accomplishing a task in parallel; sequential sub-computations called *operators* are embedded within that framework. To guarantee that a Delirium program will execute deterministically, one need only ensure that operators do not maintain state across invocations.

We believe that embedding coordination languages such as Delirium offer significant advantages for the expression of parallelism. One can express all the glue necessary to coordinate a mid-sized application on a single page of Delirium. This organizing principle makes parallelization easier. Instead of scattering coordination throughout a program, creating a set of ill-defined sub-computations, the compiler front-end precisely defines sequential operators and embeds these operators within a coordination framework.

Each of these operators is callable directly from Delirium with the same syntax as a function invocation. Operators can be written in any language, including traditional imperative languages like C or FORTRAN. This allows the compiler to take advantage of existing libraries.

```
main()
  let board = empty-board()
  in show-solutions(do-it(board,1))

do-it(board,column)
  let h1 = try(board,column,1)
      h2 = try(board,column,2)
      h3 = try(board,column,3)
      h4 = try(board,column,4)
      h5 = try(board,column,5)
      h6 = try(board,column,6)
      h7 = try(board,column,7)
      h8 = try(board,column,8)
  in merge(h1,h2,h3,h4,h5,h6,h7,h8)

try(board,column,row)
  let new-board = add-queen(board,column,row)
  in if is-valid(new_board)
      then if is-equal(column,8)
            then new-board
            else do-it(new-board,incr(column))
      else 0
```

Figure 5.2: Eight-queens problem in Delirium.

Basic Delirium

At heart, Delirium is a straight-forward functional language which supports first class functions, recursion, iteration, let-bindings, and conditional expressions. All functions are evaluated strictly. There are no computation primitives in the language; all real work is accomplished within operators that are defined in a different language.

Figure 5.2 contains a sample Delirium program which solves the eight queens problem, expressing backtracking directly:

The code uses the operators `is-equal`, `is-valid`, `add-queen`, `show-solutions`, and `empty-board`. Because each of these does not involve much computation, the overhead for expressing all the backtracking in parallel is significant[1]. A simple solution to reducing overhead is to express only two levels of the recursion in Delirium, calling a recursive C operator to descend the rest of the search tree. The modified version redefines `try` to be:

```
try(board,column,row)
  let new-board = add-queen(board,column,row)
  in if is-valid-op(new-board) then
```

[1]On twelve processors, this version took three seconds versus five seconds for the sequential version.

```
    if is-equal(column,2) then
      do-it-op(new-board,incr(column))
    else do-it(new-board,incr(column))
  else 0
```

This version, run on eight Sequent processors, is seven times faster than a sequential C program.

Support for Coordination Structures

This section describes how one can build coordination structures using Delirium *transforms*. Each application of a transform creates a structure which connects two successive stages in a pipelined computation.

Names in Delirium refer to either *integers*, *functions*, *transforms* or n-dimensional *arrays* of untyped values. A transform is a rule which replaces a set of *input arrays* by a *result array*; the result array represents some permutation (with possible copying) of the values in the input arrays. Transforms have two parts, a *reshape statement* and a set of *wiring rules*. The wiring rules describe how the transform will permute its input arrays. The reshape statement determines the shape of the transform's result. It also requires that the shapes of the transform's input arrays conform to an *input pattern*. Application of a transform to an array that does not match the transform's input pattern results in a runtime error.

The following grammar describes the syntax of transforms:

transform ::= *heading reshape-stmt*
[*wiring-rules*] [*fill-constant*]

heading ::= *identifier* '(' *identifier-list* ')'
identifier-list ::= *identifier* ',' *identifier-list*
| *identifier*

reshape-stmt ::= *result-shape* '<-' *input-pattern*
input-pattern ::= *subscript-expr*
result-shape ::= *subscript-expr*

wiring-rules ::= 'with' *wiring-rule-list*
wiring-rule-list ::= *wiring-rule wiring-rule-list*
| *wiring-rule*
wiring-rule ::= *lsubscript-expr* '=' *rsubscript-expr*

fill-constant ::= 'fill' *number*

lsubscript-expr ::= *identifier* '[' *simple-expr-list* ']'
simple-expr-list ::= *simple-expr* ',' *simple-expr-list*
| *simple-expr*
simple-expr ::= *number* | *identifier*

rsubscript-expr ::= *subscript-expr* | *number*
subscript-expr ::= *identifier* '[' *index-expr-list* ']'
index-expr-list ::= *index-expr* ',' *index-expr-list*
| *index-expr*
index-expr ::= *expr*
expr ::= any Delirium expression (should yield an integer at runtime)

The operator '=' in a wiring rule is read "depends on". Each wiring rule specifies how a section of the transform's result depends on its input[2]. For example, the transform shown below groups into pairs adjacent elements of a one-dimensional input array:

```
adjacent(P)
  C[n,2]<-P[n]
  with
    C[i,j]=P[i+j]
```

The reshape statement, `C[n,2]<-P[n]`, contains the input pattern `P[n]`. This pattern matches any one-dimensional array, binding the name `P` to that array. The reshape statement declares that the result of the transform will be a two-dimensional array, `C`.

The single wiring rule of this transform maps successive (overlapping) pairs of adjacent elements from P into the corresponding columns of C. In general, the left-hand side (lhs) of the wiring rule specifies some elements of the transform's result. Array subscripts appearing on the lhs must be either *index variables* or constants. An index variable is an identifier which represents the entire range (with zero origin) of index values along a particular dimension of the result array. Index variables must be unique within a wiring rule and *consistent* within a transform's set of wiring rules. To be consistent, a given index variable must always refer to the same dimension of the result.

The right-hand side (rhs) of a wiring rule must be either a constant or a selection of some elements from one of the transform's input arrays. Array subscripts appearing on the rhs can contain arbitrary arithmetic expressions. Index variables bound on the lhs of a wiring rule can appear in its rhs. If an rhs subscript expression does not specify a valid index of the input array, the value of the entire rhs expression is determined by the *fill constant* of the transform. When defining transforms containing such rhs expressions, the programmer must explicitly specify a fill constant.

The semantics of a set of wiring rules are captured by the following algorithm:

for each wiring rule
  for each value in the range of each index variable
    compute the lhs expression
    if the indicated element of the result has
      already been specified

[2]For wiring rules in which the left and right subscript expressions refer to different arrays, '=' is semantically equivalent to an assignment. At present, this is the only kind of rule permitted in a transform.

then report error
else assign the value of the rhs expression
to this element

To understand how transforms work, imagine the dataflow graph of a computation to be a set of ribbon cables. These cables have a male and a female plug, both n-dimensional, and wires which connect each input of the female plug to one or more outputs of the male plug. The male plugs correspond to Delirium arrays. Female plugs correspond to the input patterns of transforms; they determine what shapes of input arrays can "plug into" the transform. The wiring rules of the transform constitute a schematic for creating a new male plug.

A computation is just a series of transformations, applied to the program's input "wires". For completeness, we should generalize the ribbon cable analogy so that, at any point, one can bifurcate a cable such that it has two male ends (since a program may copy an array).

The only thing that remains is to specify how actual work gets done. To do this, we introduce the primitive `map`, which applies a function to groups of elements in Delirium arrays. Depending on how it is applied, `map` groups array elements in different ways; this flexibility is necessary to support functions with multiple array arguments and multiple return values. The simplest way to use `map` is to apply a single argument, single return value function to one array. The result is an array of the same shape as the input array, where each element in the new array is computed by applying the function to the corresponding element of the original one.

If `map` applies a function that returns r results, the dimensionality of the output array is correspondingly increased. An $n \times m$ array, for example, would be transformed into an $n \times m \times r$ array. Element $(8, 10, 2)$ is the third return value of the function when it is applied to element $(8, 10)$ of the input array.

There are two ways to use `map` with multiple argument functions. The first is to apply an n argument function to n arguments. All the arguments must *conform*, meaning that they must either be arrays with the same shape, or scalars, which are automatically replicated (and coerced into the array type). For example, if a four argument single return value function is applied to two $m \times q$ arrays and two scalars, the result would be an $m \times q$ array where each element is computed by applying the function to the two scalars and the two corresponding array elements.

The other way to handle multiple inputs is to get them all from a single dependency array. In this case, the programmer applies an n-element function to an array whose lowest-numbered dimension is n. If the function has r return values, the `map` operation will change the size of the array's lowest-numbered dimension from n to r. For example, if a three argument, two return value function is applied to a 20×3 dependency array, the result array will be of shape 20×2. The two elements in row ten of the result array are the two values returned by the function when it is applied to the three elements in row ten of the original array.

To summarize, `map` can apply a function to either one or n dependency arrays. In the first case, the input arity of the function must match the lowest-numbered dimension of the input array. In the second case, the input arity must be n. The lowest-numbered

dimension of the output array will be the output arity of the function.

Note that the reshaping effect of `map` can be described as a transformation. The full semantics of `map` cannot be expressed in a transform, however, because transforms can only build coordination structures. They are not able to apply functions to data flowing through a coordination structure. With this restriction, Delirium linguistically enforces a decoupling between coordination and computation. Normal transforms create patterns of dataflow. The `map` transform applies a function across a pattern.

To demonstrate how `map` is used in practice, here is Delirium code that applies a function of two arguments to each pair of adjacent elements in a vector. The first step is to transform the vector into a two-dimensional array where each two-element column contains adjacent elements from the vector. Once the intermediate array is constructed, a simple `map` operation performs the computation:

```
adjacent(group,P)
  C[n,group]<-P[n]
  with
    C[i,j]=P[i+j]

f(a,b)
  op(a,b)

f-on-adjacent(vector)
  map(f,adjacent(2,vector))
```

This example generalizes the definition of adjacent to group by an arbitrary number (rather than just by pairs). In the example, `vector` is a one-dimensional array, `f` and `f-on-adjacent` are functions, and `adjacent` is a transform. The identifier `op` refers to a functional operator defined in a computational language (see the above description of basic Delirium). Note that `adjacent` makes use of an integer argument `group`. If a program passes an array in this argument position, it will generate a runtime type error.

5.1.2 Multi-Stage Transforms

While the current transform mechanism is sufficient for the specification of any coordination structure, it is oriented toward building these structures one pipeline stage at a time. Coordination structures built this way are easy to understand and manipulate because they can be represented as relationships among n-dimensional arrays. To represent a general coordination structure, we need a much more flexible data structure, such as a directed graph. Some coordination structures, such as wavefront computations, can be embedded in arrays. For such structures, it is useful in practice to support both views. Transforms expecting arrays should be able to take array-embedded coordination structures as arguments. In other contexts, programmers need to be able to express a direct modification to the array-embedded graph.

We are developing *multi-stage transforms* which provide this flexibility. Multi-stage transforms include a notion of recurrence. Recurrence relations are resolved into

```
hydro-stencil(P)
  C[n,m,5]<-P[n,m]
  with
    C[i,j,0]=P[i-1,j-1]
    C[i,j,1]=P[i-1,j+1]
    C[i,j,2]=P[i,j]
    C[i,j,3]=P[i+1,j-1]
    C[i,j,4]=P[i+1,j+1]

hydro-update(current-grid)
let
  new-grid=map(hydro-point,hydro-stencil(current-grid))
in new-grid
```

Figure 5.3: Hydrodynamics update from climate model expressed in Delirium.

coordination structures that can be manipulated explicitly or implicitly (as array-embedded graphs) by transforms.

Most current Delirium applications use a different strategy for expressing recurrences. Prior to the development of multi-stage transforms, we provided a recurrence notation similar to Crystal [16] which programmers could use as an alternative to writing recurrences as iterations over transforms. Crystal is a system that converts computations expressed as recurrence equations into a systolic architecture. At an intermediate step in this conversion, the Crystal compiler derives a set of linear combinations that express the communication inherent in the given recurrence. There is a straightforward mapping from such sets to Delirium transforms.

5.1.3 A Delirium Example

This section presents a part of the climate modeling application expressed in Delirium. It represents the hydrodynamics computation, the part of the climate model that computes how air will flow during a simulation timestep. The Delirium compiler uses this code to determine which processors will need to exchange data. The dataflow expressed in this example has been simplified; the actual code uses a more complex, three-dimensional pattern of neighboring points.

The code shown in Figure 5.3 a classic 5-point stencil, a pattern of how data is read from neighboring points to update a particular point. This stencil is an analogue of the one actually used by our climate modeling benchmark. The Delirium transform `hydro-stencil` expresses all the dependencies among grid-points in the computation. It transforms an `n` by `m` grid of data points into an `n` by `m` by 5 matrix, where the last dimension reflects a grouping of all the neighboring points needed to update the value at point `(n,m)` in the original grid.

The Delirium function `hydro-update` performs an update on the grid by first grouping the data by applying the `hydro-stencil` transform, and then mapping the function `hydro-point` across the grouped data. `hydro-point` takes its 5 arguments and combines them to yield an updated grid-point value.

5.1.4 Related Work

Functional Languages

One can achieve the organizing effects of coordination structures using higher-order functions. For example, one could write a function that groups pairs of adjacent elements in a vector:

λP . λf .
construct array A
with range $i = 0$ to $length(P) - 1$
such that $A[i] = f(P[i], P[i+1])$

To realize the same degree of parallelism as the equivalent Delirium transform, this higher-order function must be implemented in a functional language which has *lazy aggregate construction* and strict function applications. By lazy aggregate construction, we mean that an array can be used before all its elements are computed. Combining this property with strict function application, a functional language could realize the multi-dimensional, pipelined communication pattern created by Delirium transforms.

Strict Functional Languages

Most dataflow languages, including VAL [43] and SISAL [44], are evaluated strictly. These languages have strict aggregate construction, and so they can not implement Delirium transforms. The language ParALFL [27], though not a dataflow language, also has strict arrays; however, one could implement coordination structures inefficiently in ParALFL using its lazy lists.

Lazy Functional Languages

Some dataflow languages, such as Id [6, 45], have lazy aggregate construction as well as lazy function application. To realize the parallelism of Delirium transforms, a coordination structure built in Id would require strict evaluation of function applications within lazy arrays. This un-intuitive evaluation strategy could be arranged by a compiler that was designed to recognize coordination structures as a stylized idiom. In contrast, Delirium adds the transform mechanism to an otherwise strict language; using this mechanism, programmers directly express the desired evaluation behavior.

Array Comprehensions

Anderson and Hudak discuss the construction of "lazy arrays within a strict context" using a functional syntax called *array comprehensions* [5]. One can use such array comprehensions to implement coordination structures. However, array comprehensions can conveniently express only those coordination structures which can be embedded in arrays.

Like multi-stage transforms, array comprehensions can express recurrences. We will provide a detailed comparison between array comprehensions and multi-stage transforms in a future report. We have found that the complexity of compiling the Delirium recurrence notation into iterations over transforms is similar to the complexity of compiling Haskell [28] array comprehensions for sequential machines.

5.1.5 Aggregate Primitives

Early Aggregate Languages

Delirium transformations have been heavily influenced by APL [32], which introduced the idea of a pipeline of functional transformations that modify an aggregate structure. This idea was significantly elaborated by FP [7], which provides a rich set of functional operators for creating new transformations. APL does not have first class functions, and so can't express coordination structures. FP's functional operators are similar to Delirium transforms, but operate on a data object that must model memory as well as coordination, and are thus difficult to implement efficiently. Water's series expressions [61] also create pipelined computations; however, like FP and APL, they provide a fixed set of operators for modifying dataflow through the pipeline. In contrast, Delirium transforms are a general mechanism for creating dataflow modification operators.

5.1.6 Summary of Delirium

We have proposed a compiler intermediate form which supports the creation and manipulation of multi-dimensional dataflow graphs. We have used this intermediate form explicitly to construct concise and efficient implementations for several applications. We have also used it to express the communication and loop interaction information necessary to efficiently schedule the four main benchmark applications of this dissertation.

5.2 Runtime System Implementation

5.2.1 Tarmac

Our runtime system is designed to adaptively move sub-computations among processors. To keep track of these sub-computations, and the data objects that they require, we built a distributed shared memory toolkit called *Tarmac*. Tarmac is a *language system substrate* on which systems for distributed parallel programming can be built. The basic unit of state in Tarmac can be viewed as both 1) a block of memory that can be directly accessed by machine instructions, and 2) a logical entity with a globally unique name that may

be efficiently located, copied and moved. To support higher-level synchronization models, the movement of a memory unit may optionally enable computations.

Tarmac is more flexible than models such as distributed virtual memory, shared tuple space, or distributed objects. It avoids the limitations of fixed page size, fixed data placement policy, and type-system or language dependence. This flexibility allows Tarmac to support a wide range of parallel programming models efficiently

Tarmac is designed to support adaptive parallel programming on a network of computers (uniprocessors, multiprocessors, or both). A variety of possible models exist for this type of programming, each with its own advantages: functional, object-oriented, and dataflow languages, parallelizing compilers for sequential languages such as FORTRAN, parallel database systems, and so on. Our main goal was to identify the functionality common to these models, and implement it in a single system layer, thereby facilitating the implementation (and interaction) of different models. While these models differ in many important ways, they all provide some form of "state" that is shared and communicated among the parts of the computation. The resulting system, Tarmac, is a toolkit for building systems that incorporate shared state.

Examples of systems for distributed parallel programming include Amber [15], Linda [22], and Ivy [36]. These systems represent different models, but they have two important and desirable properties in common: 1) global shared state is encapsulated and explicitly managed by the underlying system (language, runtime system, and operating system); 2) the system handles the movement of parts of the shared state between computational nodes. These properties eliminate the need for the programmer to design and program message-passing protocols for maintaining and replicating shared state.

The above systems, however, have properties that limit their range of applicability. Shared virtual memory systems such as Ivy have page granularity. Entire pages must be moved between nodes even if only single bytes are referenced, and some global access patterns may cause thrashing. Implementations of the Linda system have embedded policies for tuple transmission and placement.

The goal of Tarmac is to provide a facility for managing shared state with the beneficial features of existing systems (encapsulation of small and large state units, mobility, global reference and location, and replication), but without language dependence and system-enforced data placement policies. One of Tarmac's central contributions is that it supports a dual view of state units. Clients can view state both as "black boxes" that can be named and moved, and as virtual memory segments that can be manipulated by statements in arbitrary programming languages.

With this goal in mind, we designed a model of shared state that we call *mobile memory*. In this model, clients can create, access, move and copy arbitrary-size *memory units*. Tarmac provides only mobile memory; any notion of "type", as well as policies for data location, movement, and concurrent access, are left up to the client. Tarmac's level of abstraction is high enough to hide details of communication and allocation (thus simplifying language system development) yet low enough so that few restrictions are made on these systems.

Mobile Memory

Tarmac is designed around a *mobile memory* abstraction. Mobile memory is a model of the communication and storage resources available in a distributed computing network. The model involves the following types of entities:

- *memory unit* (MU): a region of memory.
- *habitat*: a set of memory units (generally an address space)
- *label*: a tag, attached to MUs, with client-determined interpretation

Tarmac uses unique identifiers (UIDs) to identify all of these entities. UIDs have an immutable part, used to establish object identity, and a location hint part. UIDs are always passed by reference to Tarmac, so that Tarmac can update their hint part whenever they are used.

A *memory unit* (MU) is a region of memory with a definite size. MUs can contain code or data; there is no Tarmac-defined notion of type. Tarmac provides the following interface for creating MUs:

```
UID-list = create_memory_units(number_of_units, size);
make_immutable(UID);
```

Labels A *label* is a client-defined tag for MUs.

```
label_UID = create_label();
bind_label(label_UID, UID);
```

One can associate zero or more labels with a particular MU. Applications can use labels to represent MU types or other information. Copies of the same application on separate hosts must agree, by some higher level mechanism, on the interpretation of label UIDs. The location hint part of a label's UID refers to the location of an arbitrary MU that has been bound to that label, so that clients can access MUs associatively through labels.

Habitats represent the physical location of a MU. A habitat may be a virtual address space on a particular host, or a file system (only address space habitats are supported in the current design). Processes running in a habitat can also interact with the local operating system to create non-shared data structures. Every MU has a single *current location* which is a habitat. MUs can be moved or copied from one habitat to another, using

```
move(UID, target_UID);
copy(UID, target_UID);
virtual_move(UID, target_UID);
virtual_copy(UID, target_UID);
```

An MU A can be moved to any other MU B, after which A*'s* current location becomes B*'s* habitat. If the MU has been designated *immutable*, Tarmac may maintain a copy of the MU in both the source and destination habitats, for more efficient access. *Copy* creates a new MU and then behaves like move. UIDs act as capabilities to MUs; any MU A which knows the UID for an MU B can specify B as a target for a move.

MUs located in the same habitat can reference each other directly, using memory addresses. The memory address of an MU is made available when it is first moved to a habitat (see Section 2.3). If an MU containing direct references to other MUs is moved, it is the responsibility of the language system to patch up its references.

Tarmac supports an *alignment* facility, similar in intent to Emerald's notion of attached objects [12]. If A and B are aligned, and A moves to a new habitat, Tarmac will move B to join A. Furthermore, the relative addresses of A and B will remain the same. Tarmac tries to place aligned MUs in the same physical page, increasing the efficiency of moves involving large numbers of small, aligned MUs. The interface for alignment specification is

```
align(UID1, UID2);
```

As an example of the use of alignment, if a user-level computation creates a binary tree in which all nodes are aligned with their parent, a move of any node will cause the subtree rooted at that node to move.

Events MU motions can trigger computations in the source or destination habitats. For example, an MU move could correspond to an RPC request. Tarmac clients may indicate which MU moves are to trigger computations by registering *events*.

```
event_UID = add_event(target_UID, label_UID, ...);
remove_event(target_UID, label_UID, ...);
```

An event is an ordered tuple of one or more UIDs. The first field in the tuple names a move target T. The remaining fields are labels. For an MU move to generate an event E, the move target T must match the first field in E and each of the remaining fields of E must be a member of the moved MU's set of label UIDs. The first field of an event can be the special label '*', which matches any move target.

Events are transferred to the Tarmac client either through a queue in the habitat or through a software interrupt. In either case, an event descriptor is passed to the client. Event descriptors contain the event_UID of the event, the virtual address of the MU in the source and destination habitats, and the UIDs of both habitats.

Language systems can base decisions to suspend and resume processes on these asynchronous events. The event mechanism is sufficient to support any currently popular technique for synchronization.

For example, object-oriented systems can use events and MU motion to implement the invocation of operations on objects. A simple system would define an *operation* object type as an appropriately labeled MU. Moving an operation object O to an MU T would denote invocation of the operation represented by O on the object represented by T. The state of an operation object could include information such as the type of the target object and an operation number for that type. The language's runtime system would define an event (*, *operation*), where *operation* is an agreed upon label for MUs representing operations. Occurrences of this event could then trigger the local invocation of the represented operation (plain RPC would work similarly). We have to implemented a version of the object-oriented language Sloop [37] using this general strategy.

5.2.2 Tarmac Implementation

The Tarmac implementation is divided between a *Tarmac server* (one per host) and a runtime library present in each habitat. The server can reside in the operating system kernel or run as a user process. To implement *virtual move* and *virtual copy* the server requires notification whenever a page fault occurs in a local habitat.

Habitats are generally implemented as virtual address spaces loaded by a local mechanism. Through the runtime library, the application registers with its local Tarmac server and can begin interacting with the global mobile memory network. For efficiency, the runtime library handles MU allocation without contacting the Tarmac server. Each copy of the library maintains the mapping between UIDs and virtual addresses for its habitat. Library code can request multiple UIDs from the Tarmac server, so that it does not have to make such a request for every MU allocated.

The Tarmac library matches incoming MUs against a set of currently active event types. The library uses either a software interrupt or a special queue to transmit events to processes running in the habitat.

Tarmac maintains information that allows it to recognize when multiple aligned MUs exist in a contiguous region of memory (part or all of a physical page). When it recognizes this situation, it can transfer the region directly to the destination host. Otherwise it packs the aligned objects into a new page and sends that page. The receiving server always tries to allocate contiguous memory for the aligned MUs.

Tarmac assumes that given a UID, a language system (or Tarmac itself) may need to locate an MU's host from among thousands of hosts on a network. When an MU M moves from habitat *H1* to habitat *H2*, Tarmac holds a forwarding address for M on *H1's* host. When the system attempts to move another MU to M at its old location, the kernel forwards the move to *M's* new location, and updates the host initiating the errant move. The details of the forwarding algorithm, such as when to update backward hosts along a chain of forwarding addresses, are essentially identical to those of the Sloop [37] forwarding algorithm.

5.2.3 Related Work

In this section we contrast Tarmac with other systems that support manipulation of distributed shared state. We evaluate these other system in terms of Tarmac's goals: those of providing a language-system substrate for parallel distributed computation in a variety of programming models. The limitations we point out in these systems are relative to this goal, and are not intended as criticisms of the systems in general.

The key property of Tarmac is that it allows a memory unit to be viewed as both 1) an abstract entity that can be named, moved, copied, *etc.*, and 2) an array of bytes that can be directly manipulated by statements of an arbitrary programming language. Other systems do not afford this flexibility.

Distributed Virtual Memory Distributed virtual memory provides the abstraction of a consistent virtual address space shared by processes running on separate hosts. A protocol similar to the cache consistency protocols used in shared-memory multiprocessors governs

page movement and replication. Examples of systems providing distributed virtual memory include Ivy [36].

The distributed virtual memory model has several possible drawbacks as a general substrate. First, because the resolution of page-table mapping is that of a fixed-size page (typically 8KB to 32KB on current machines) the granularity of state operations (movement and replication) is fixed. For programs that put many small data items on a single page, this granularity may be too large, causing increased contention and excessive data movement. Second, the model's concurrency control mechanism (single-writer, multiple-reader) dictates the data movement policy. Clients have no direct control over data movement, and it may be difficult to prevent "data thrashing" (situations in which a shared data structure is moved frequently between clients, each of which accesses it only briefly). Finally, distributed virtual memory is not scalable. The system cannot keep track of pointers within a page to other pages, so when a non-resident page is accessed the request must, in some cases, be broadcast.

Shared Tuple Space Linda [22] is a system based on the abstraction of tuple space shared among multiple concurrent processes. Its goals are similar to those of Tarmac: to provide a multiple-language substrate for parallel computing. The Linda model has the following limitations. First, the client has no direct control over tuple placement or communication pattern. Each Linda kernel implementation dictates a particular policy, determined by when and where *in()* and *out()* messages are sent. A language system substrate should not dictate policy, since a fixed policy cannot work well for all possible applications.

Second, the Linda model requires all shared state to be encoding into tuples. This imposes extra work on some applications. Finally, the model does not scale well in all cases. Linda kernels must resort in the worst case to either broadcast or centralization.

Master/Slave Systems In the master/slave model, a single *master* process generates and accepts asynchronous function calls that are performed in parallel on a set of *slave* processors. There is no communication between the slaves. Marionette [57] is an example of such a system. Marionette provides shared global state that is read/write by master and read-only to slaves. Because the master processor is a bottleneck, the master/slave model has a limited range of values of the (grain-size, number of slaves) pair in which it performs well.

Object-Oriented Systems Many system have used the object model for parallel distributed computation. Some of these systems, such as Matchmaker, [33] and the Apollo Network Computing Architecture (NCA) [46], are intended as a structuring mechanism for permanent storage and client/server interactions, They provide an interface description language to specify both halves of what is essentially an RPC connection.

We focus our attention, therefore, on systems that support small objects, efficient access to both local and remote objects, and object mobility. Examples include Emerald [12], Amber [15] and Sloop [37]. These systems suffer from several drawbacks relative to the goals of Tarmac:

Parallel object models have difficulty accommodating typical numerical data structures such as arrays. Is one expresses the array as a single object, than all operations on the array must pass through a single processor, which is likely to become a bottleneck in communication or computation. If each of the array elements is a separate object, the latency of individual operations increases. In addition, most object-oriented systems are coupled to a fixed type system, and usually to a single programming language.

The Tarmac location hint forwarding scheme is similar to the Hermes [56] location independent invocation mechanism, with two important exceptions. First, the Hermes system fits object location into an operation invocation mechanism with a fixed request/reply protocol. This precludes higher level systems from expressing higher level requests, such as a search for a whole group of objects or for any member from a group. Tarmac supports such requests. For example, one can use a label_UID to specify the target of a move. The moved MU will end up in the habitat of some object labeled with the label_UID.

Second, the Hermes system requires the program to explicitly specify, for each object T, all other objects to which T holds references. Hermes uses this interobject reference information to maintain a cache of location hints at each host. The cache contains a hint for each object for which a local object holds a reference. Hermes, however, does not make full use of the interobject reference information. References to objects may be either UIDs or virtual addresses. The latter occurs because objects can share an address space. However, when an object T moves from address space $A1$ to address space $A2$, objects in $A1$ holding virtual address references to T will have to replace these references with UIDs (the reverse for reference holders in $A2$). Hermes does not do this reference patching, because its designers wanted to make it a language independent facility. At the same time, it is dependent on higher level language support in that it requires interobject reference information for its location independent invocation mechanism.

Tarmac avoids this dichotomy by keeping a location hint with each UID. This approach makes UIDs larger and causes memory units to contain possibly redundant location hint information. However, it releases language systems from the need to specify anything about a memory unit's internal structure. This makes memory unit creation more efficient, and decouples Tarmac from the type systems of its clients. Further, using this approach does not increase network traffic because each Tarmac server maintains a cache of hint updates.

Chapter 6

Summary

In this chapter, we summarize why we believe our work will be widely applicable. We then outline the main contributions of the dissertation. Finally, we discuss some directions for future work.

6.1 Applicability

Most of the success that we were able to achieve with our four benchmark programs hinges on a single property that those programs share: temporal work distribution locality. For the work reported in Chapter 3 of this dissertation to be widely applicable, there will have to be a large class of parallel applications that exhibit this locality property.

We have direct experience with about a dozen parallel applications. Some programs, especially those that use Monte-Carlo methods[60] do not exhibit this property. However, we have observed this property in most of the applications we have examined.

Among scientific programs, there is a powerful reason to expect that parallel loops within these programs will exhibit work distribution locality. Many scientific programs simulate the physical world using *explicit methods*. Explicit methods are so called because each relevant physical quantity is explicitly updated for every spatial grid point in the simulated space, and for every timestep of the simulation. Updates at a particular point in space are computed to be some function of neighboring points in space.

To retain physical stability, explicit methods can only transfer physical quantities locally in any given timestep. For example, suppose you hold a lighted match at one end of a metal plate. Heat radiates outward from the match, progressing a finite distance with each simulation timestep. Now, we further suppose that, where the plate reaches a temperature greater than some constant K, we wish to simulate the air turbulence above the plate. A parallel program simulating this scenario will be irregular. Grid points at temperature greater than K will require more computation than the other grid points. If the program uses an explicit method, it must also exhibit work distribution locality, *in order to remain numerically stable.* If it were possible for a large proportion of the grid points to jump from temperatures less than K to temperatures greater than K in only one timestep, then the program's simulation timestep would be too large.

More concretely, consider the extreme case. In one timestep, heat can not travel

from one edge of the metal plate to the other edge, because explicit methods only consider the interactions among neighboring points in space. The program will have to execute a number of timesteps that is proportional to the size of the metal plate. Hence, when the computational cost of a grid point is determined by the physical quantities it represents there is a good chance that, as in our heat diffusion example, the work distribution of the program will evolve slowly relative to the duration of the program.

6.2 Contributions

The most important contribution of this dissertation is its identification and exploitation of work distribution locality properties. Previous work on irregular parallel program scheduling unearthed the following dilemma: compilers can not predict work distribution accurately enough to schedule programs efficiently; however, runtime load balancing solutions, while more accurate, incur prohibitive overhead. This dissertation shows how to avoid this dilemma whenever irregular loops within parallel programs have *work distribution locality*, that is, when a loop retains a similar distribution of individual iteration execution times from one *execution instance* to the next. An execution instance is simply an execution of the entire loop, possibly in parallel.

Where this common case arises, we exploit it through *work distribution caching*: guessing the work distribution of a loop execution instance based on earlier measurements. We also exploit work distribution locality through *deferred load balancing*: reducing communication overhead and thrashing potential of load balancing algorithms by applying them across multiple execution instances of a loop.

Chapter 3 described work distribution caching and showed that it could be applied to *any* dynamic scheduling method. Chapter 3 then described the dynamic scheduling method used by our prototype scheduling system: deferred load balancing. In Chapter 4, we described two other techniques that play important supporting roles in achieving efficient execution of irregular parallel programs. First, we described *probabilistic grain size selection*, a method for determining how much work each processor should perform before synchronizing with other processors. Second, we presented techniques for orchestrating the interactions among concurrently executing parallel loops. Finally, we outlined the Delirium intermediate form and the Tarmac distributed shared memory toolkit, two key systems we developed to support adaptive parallel programming.

Figures 6.1 through 6.4 summarize the performance results of this dissertation. Each figure compares dynamic load balancing, augmented with the techniques described in Chapter 4, with the best previously proposed scheduling method. The results demonstrate that, for these applications, the techniques described in this dissertation achieve near-optimal efficiency on large numbers of processors. In addition, they perform significantly better, on these problems, than any previously proposed static or dynamic scheduling method.

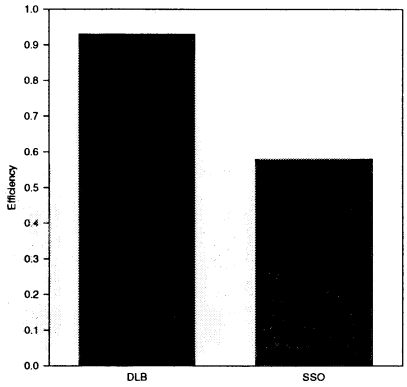


Figure 6.1: Performance of Deferred Load Balancing on climate model, compared to best previous approach.

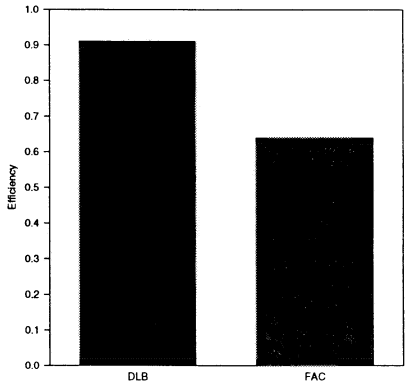


Figure 6.2: Performance of Deferred Load Balancing on AMR, compared to best previous approach.

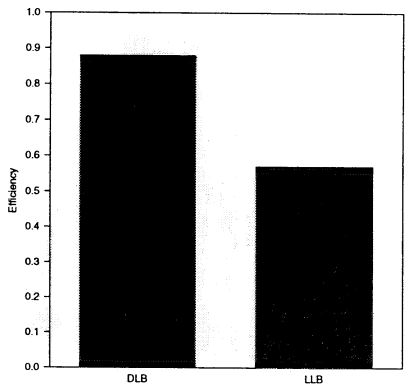


Figure 6.3: Performance of Deferred Load Balancing on EMU, compared to best previous approach.

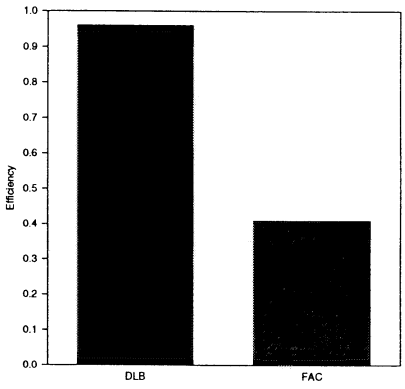


Figure 6.4: Performance of Deferred Load Balancing on Psirrfan, compared to best previous approach.

6.3 Future Directions

To expand upon this work, we are currently pursuing several avenues of research. First, we are applying the Tarmac system to the support of object-oriented parallel languages. Second, we are investigating the phenomenon of work distribution coherence. We are measuring several dozen parallel applications to determine how frequently successive execution instances of parallel loops within these applications exhibit closely similar work distributions. Finally, we are investigating how well the scheduling methods developed in this dissertation can be applied to parallel applications that have irregular communication patterns as well as irregular distributions of loop iteration execution times.

Bibliography

[1] Bryan Ackland, Steven Lucco, Tom London, and Eric DeBenedictis. "CEMU: A Parallel Circuit Simulator,". In *Proceedings of the International Conference on Computer Design*, October 1986.

[2] Frances E. Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. "An Overview of the PTRAN Analysis System for Multiprocessing,". *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[3] Randy Allen, Donn Baumgartner, Ken Kennedy, and Allen Porterfield. "PTOOL: A Semi-Automatic Parallel Programming Assistant,". In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 164–170, St. Charles, Illinois, August 1986. IEEE Computer Society Press, Washington, D.C.

[4] Ann Almgren. *A Fast Adaptive Vortex Method Using Local Corrections*. PhD thesis, Center for Pure and Applied Mathematics, UC/Berkeley, 1991.

[5] Steven Anderson and Paul Hudak. "Compilation of Haskell Array Comprehensions for Scientific Computing,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 137–149, White Plains, New York, June 1990. ACM Press, New York.

[6] Arvind and Kim P. Gostelow. "An Asynchronous Programming Language and Computing Machine,". Technical Report TR114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[7] J. Backus. "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs,". *Communications of the ACM*, 21(8), August 1978.

[8] Scott B. Baden. "Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors,". *SIAM Journal of Scientific and Statistical Computing*, 12(1):145–157, January 1991.

[9] Vasanth Balasundaram. "A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor,". *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[10] A. Barak and A. Shiloh. "A Distributed Load Balancing Policy for a Multicomputer,". *Software Practice and Experience*, page 901, September 1985.

[11] M. J. Berger and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics,". *Journal of Computational Physics*, 82(1):64–84, May 1989.

[12] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald,". *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[13] David Callahan, Keith Cooper, Ken Kennedy, and Linda Torczon. "Interprocedural Constant Propagation,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 152–161, Palo Alto, California, June 1986. ACM Press, New York.

[14] N. Carriero, D. Gelernter, and J. Leichter. "Distributed Data Structures in Linda,". In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, Florida, January 1986. ACM Press, New York.

[15] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. "The Amber System: Parallel Programming on a Network of Multiprocessors,". In *Proceedings of the Twelfth Symposium on Operating Systems Principles (SOSP)*, pages 147–158, Litchfield Park, Arizona, December 1989. ACM Press, New York.

[16] Marina Chen. "A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI,". In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 131–139, St. Petersburg Beach, Florida, January 1986. ACM Press, New York.

[17] George Cybenko. "Dynamic Load Balancing for Distributed Memory Multiprocessors,". *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[18] Ron Cytron. "Limited Processor Scheduling of Doacross Loops,". In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 226–234, University Park, Pennsylvania, August 1987. Pennsylvania State University Press.

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. "An Efficient Method of Computing Static Single Assignment Form,". In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 25–35, Austin, Texas, January 1989. ACM Press, New York.

[20] Zhixi Fang, Pen-Chung Yew, Peiyi Tang, and Chuan-Qi Zhu. "Dynamic Processor Self-Scheduling for General Parallel Nested Loops,". In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 1–10, University Park, Pennsylvania, August 1987. Pennsylvania State University Press.

[21] Geoffrey C. Fox. "Domain Decomposition in Distributed and Shared Memory Environments,". In E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, editors, *Proceedings of the First International Conference on Supercomputing*, pages 1042–1073, Athens, Greece, June 1987. Springer Verlag, Berlin, Germany.

[22] David Gelernter. "Parallel Programming in Linda,". In *Proceedings of the International Conference on Parallel Processing*, pages 255–263, August 1985.

[23] Milind Girkar and Constantine Polychronopoulos. "Partitioning Programs for Parallel Execution,". *1988 International Conference on Supercomputing (ICS)*, pages 217–229, July 1988.

[24] Susan L. Graham, Steven Lucco, and Oliver J. Sharp. "Orchestrating Interactions Among Parallel Computations,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 100–111, Albuquerque, New Mexico, June 1993. ACM Press, New York.

[25] O. Hanson and A. Mayer. "Heuristic Search as Evidential Reasoning,". In *Proceedings of the Fifth Workshop on Uncertainty in AI*, August 1989.

[26] Kaarlo Heiskanen. *Tomography with Limited Data in Fan Beam Geometry.* PhD thesis, UC/Berkeley, 1990.

[27] Paul Hudak and Lauren Smith. "Para-functional Programming: A Paradigm for Programming Multiprocessor Systems,". In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 243–254, St. Petersburg Beach, Florida, January 1986. ACM Press, New York.

[28] Paul Hudak and P. Wadler. "Report on the Programming Language Haskell,". Technical Report YALU/DCS/RR666, Computer Science Department, Yale University, November 1988.

[29] Susan Hummel, Edith Schonberg, and Lawrence Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops,". In *Proceedings of Supercomputing '91*, pages 610–619, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press, Los Alamitos, California.

[30] Susan Hummel, Edith Schonberg, and Lawrence Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops,". *Communications of the ACM*, 35(8):90–101, August 1992.

[31] M. Ashraf Iqbal, Joel H. Saltz, and Shahid H. Bokhari. "A Comparative Analysis of Static and Dynamic Load Balancing Strategies,". In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 1040–1047, St. Charles, Illinois, August 1986. IEEE Computer Society Press, Washington, D.C.

[32] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, New York, New York, 1962.

[33] Michael B. Jones and Richard F. Rashid. "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems,". Technical Report CMU-CS-87-150, Carnegie-Mellon University, September 1986.

[34] C. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors,". *IEEE Transactions on Software Engineering*, SE-11, October 1985.

[35] Monica S. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, Atlanta, Georgia, June 1988. ACM Press, New York.

[36] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems,". In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[37] Steven Lucco. "Parallel Programming in a Virtual Object Space,". In *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, October 1987. ACM Press, New York.

[38] Steven Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 200–211, San Francisco, California, June 1992. ACM Press, New York.

[39] Steven Lucco and David Anderson. "Tarmac: A Language System Substrate Based on Mobile Memory,". In *Proceedings of the Tenth International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, California, 1990.

[40] Steven Lucco and Kathleen Nichols. "A Performance Analysis of Two Parallel Programming Methodologies in the Context of MOS Timing Simulation,". In *Digest of Papers: IEEE Compcon*, pages 205–210, 1987.

[41] Steven Lucco and Oliver Sharp. "Delirium: An Embedding Coordination Language,". In *Proceedings of Supercomputing '90*, pages 515–524, New York, New York, November 1990. IEEE Computer Society Press, Los Alamitos, California.

[42] Steven Lucco and Oliver Sharp. "Parallel Programming With Coordination Structures,". In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, Orlando, Florida, January 1991. ACM Press, New York.

[43] James R. McGraw. "The VAL Language: Description and Analysis,". *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[44] James R. McGraw. "SISAL: Streams and Iteration in a Single Assignment Language,". Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

[45] Rishiyur S. Nikhil. "ID Reference Manual, version 88.0,". Technical Report 284, Laboratory for Computer Science, Massachussets Institute of Technology, 1988.

[46] M. Ferhan Pekergrin. "Parallel Computing Optimization in the Apollo Domain Network,". *IEEE Transactions on Software Engineering*, 18(4):296–303, April 1992.

[47] Constantine D. Polychronopoulos and Utpal Banerjee. "Speedup Bounds and Processor Allocation for Parallel Programs on a Multiprocessor,". *Proceedings of the 1986 International Conference on Parallel Processing*, pages 961–968, August 1986.

[48] Constantine D. Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors,". In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume II, pages 39–48, University Park, Pennsylvania, August 1989. Pennsylvania State University Press.

[49] Constantine D. Polychronopoulos and David J. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,". *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[50] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computation.* Cambridge University Press, Cambridge, England, 1988.

[51] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines,". In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[52] Vivek Sarkar. "Determining Average Program Execution Times and Their Variance,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 298–312, Portland, Oregon, June 1989. ACM Press, New York.

[53] Vivek Sarkar and John Hennessey. "Partitioning Parallel Programs for Macro Dataflow,". In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 202–211, Cambridge, Massachussetts, August 1986. ACM Press, New York.

[54] Oliver Sharp. "Pythia: A Parallel Compiler for Delirium,". Master's thesis, Computer Science Division, University of California, Berkeley, 1990.

[55] Jaswinder Pal Singh, C. Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. "Load Balancing and Data Locality in Hierarchical N-body Methods,". Technical Report CSL-TR-92-505, Computer Science Department, Stanford University, January 1992.

[56] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing.* Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[57] Mark Sullivan and David Anderson. "Marionette: a System for Parallel Distributed Programming using a Master/Slave Model,". In *Proceedings of the Ninth International Conference on Distributed Computing Systems.* IEEE Computer Society Press, Los Alamitos, California, June 1989.

[58] Peiyi Tang and Pen-Chung Yew. "Dynamic Processor Self-Scheduling for General Parallel Nested Loops,". *IEEE Transactions on Computers*, C-39(7):919–929, July 1990.

[59] Peiyi Tang and Pen-Chung Yew. "Processor Self-Scheduling for Multiple Nested Parallel Loops,". *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[60] S. Ulam and N. Metropolis. "The Monte Carlo Method,". *Journal of the American Statistics Association*, 44:335ff, 1949.

[61] Richard Waters. "Appendix A: Series,". In Guy L. Steele Jr., editor, *Common LISP: The Language, 2nd edition.* Digital Press, Bedford, Massachussetts, 1990.

[62] Michael E. Wolf and Monica S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism,". *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[63] Michael J. Wolfe. "More Iteration Space Tiling,". In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989. ACM Press, New York.

[64] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachussetts, October 1989. Based on the author's Ph.D. thesis at the University of Illinois at Urbana-Champaign, 1982.