A Dynamic Scheduling Method for Irregular Parallel Programs

Steven Lucco*

Computer Science Division, 571 Evans Hall
UC Berkeley, Berkeley CA, 94720

Abstract

This paper develops a methodology for compiling and executing *irregular* parallel programs. Such programs implement parallel operations whose size and work distribution depend on input data. We show a fundamental relationship between three quantities that characterize an irregular parallel computation: the total available parallelism, the optimal grain size, and the statistical variance of execution times for individual tasks. This relationship yields a dynamic scheduling algorithm that substantially reduces the overhead of executing irregular parallel operations.

We incorporated this algorithm into an extended Fortran compiler. The compiler accepts as input a subset of Fortran D which includes blocked and cyclic decompositions and perfect alignment; it outputs Fortran 77 augmented with calls to library routines written in C. For irregular parallel operations, the compiled code gathers information about available parallelism and task execution time variance and uses this information to schedule the operation. On distributed memory architectures, the compiler encodes information about data access patterns for the runtime scheduling system so that it can preserve communication locality.

We evaluated these compilation 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 dynamic techniques described here 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 algorithm.

*Supported in part by an IBM Fellowship. Email address: lucco@cs.Berkeley.EDU. Research sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0200...$1.50

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, 25], static scheduling [8, 10, 24], linear transformation of iteration spaces to improve data locality or expose parallelism [31, 33, 34], and improved compiler technology to support the first three activities [7, 9]. In this paper we focus on a particularly intractable class of parallel programs for which static techniques such as these are not sufficient to generate highly efficient code. 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 [5], time-step subdivision [1, 20], tree traversal [12], and monte-carlo methods[30], yield irregular parallel programs. To execute such pro-

grams efficiently, we generate code that assigns tasks to processors dynamically. We require that a dynamic scheduling policy have two properties: adaptivity and statelessness. The latter property means that current scheduling decisions do not use information about past decisions; this implies that the scheduling policy will function correctly under the addition and subtraction of processors, and the addition of new tasks to the set of tasks being scheduled. 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. No previously proposed dynamic scheduling policies have both these properties; however, we believe both are necessary for efficient execution of irregular problems. In this paper, we develop an adaptive, stateless dynamic scheduling algorithm and describe how we incorporated the algorithm into a compiler for shared and distributed memory architectures.

The remainder of the paper is organized as follows. Section 2 gives some examples of irregular programs and illustrates how different types of execution time distributions can arise. Section 3 presents our basic compilation framework and describes how we modified it to incorporate dynamic scheduling. Section 4 illustrates some important properties of previously proposed scheduling algorithms. In section 5.1, we prove a new property of multiprocessor scheduling and use this property to derive a relationship between optimal grain size, available parallelism and task execution time variance. In section 5.2, we develop our probabilistic dynamic scheduling algorithm and extend it for execution on distributed memory machines. Finally, section 6 presents efficiency results for a set of application programs.

2 Irregular Programs

First we give some examples which illustrate how irregular work distributions can arise in programs. We define a parallel operation as a set of N independent tasks, such as the iterates of a DOALL loop. In an *irregular* parallel operation, task execution times vary significantly and unpredictably. Consider the irregular Fortran loop shown in figure 1.

The numbers in brackets indicate execution costs for sequential sections of the loop. The conditional statement is labeled with the probabilities of the true and false branches respectively. Even given this information it is impossible (except for very large N) to compute a schedule for this loop statically that is efficient in the average case.

Suppose that we randomly assign tasks to processors. Since the execution costs of the tasks are themselves random, we can just assign successive contiguous blocks of N/p tasks to each processor (this is equivalent to using the owner computes rule [14] for a blocked decomposition). Figure 2 shows the efficiency of this method for executing our loop example, given different values of N/p. No static method can beat the average case behavior of random assignment.

```
      DOALL 10 I=1,N
        IF (C{0.9,0.1}) THEN
          {200}
        ELSE
          {60000}
        ENDIF
10    CONTINUE
```

Figure 1: A Sample Irregular Loop

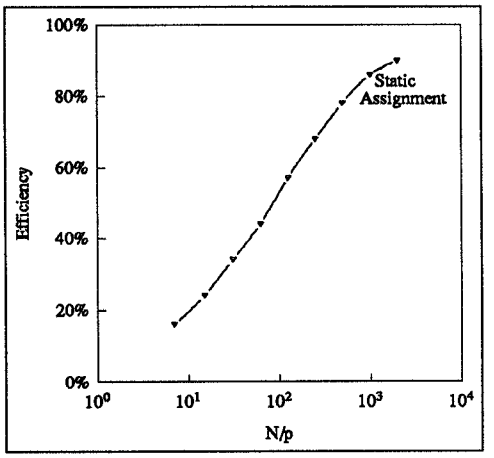

Figure 2: Efficiency of Static Assignment

This result is not limited to this example, but applies to any parallel operation with significant variance in task execution times. The expected finishing time of a chunk of N/p tasks is $N\mu/p$, where μ is the mean task cost. However, given no restriction on task cost distribution, the best bound on the finishing time of p such chunks is $N\mu/p+\sigma\sqrt{N/2}$ where σ is the standard deviation of task costs [11]. For an important class of distributions (including normal, uniform, and exponential), Kruskal [16] demonstrates the tighter bound $N\mu/p + \sigma\sqrt{2N \ln p/p}$. However, either bound leaves room for substantial inefficiency.

Since the expected inefficiency (the second term in these expressions) grows as the square root, while the expected completion time grows linearly, random assignment will be efficient for large N/p; however, we will demonstrate that it is only more efficient than dynamic methods when $\sigma \ll \mu$ (*i.e.* the tasks do not differ much in cost) or the scheduling overhead is much

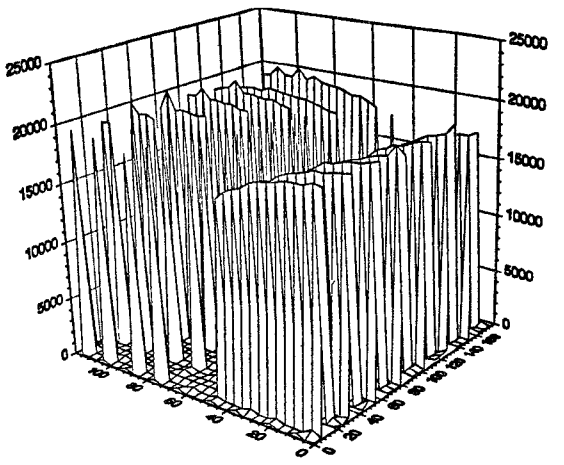

Figure 3: Execution Profile of Psirrfan Loop Nest. Z axis is microseconds; X and Y axes are iteration numbers.

greater than μ.

We abstracted our loop example from one of the programs in our benchmark set, an x-ray tomography application called Psirrfan. Psirrfan implements a new algorithm for reconstructing an image where tomographic data is only available for a limited range of angles [13]. Some of Psirrfan's most expensive operations are executed only when input data is missing. Figure 3 shows an execution profile for an important loop nest involved in implementing the image reconstruction.

In addition, Psirrfan contains another type of irregular operation. In this type of operation, the body of one loop computes the bounds for another. In Psirrfan, this gives rise to a normal work distribution; we have also seen instances where computed bounds give rise to uniform distributions.

Another application program whose measurements are reported in section 6 is the UCLA General Circulation Model [6]. Variants of this program are used widely to model trends in climate. We studied a section of this program, COMP3, that represents about 60% of the total execution time and is highly irregular. In this case, the irregularity arises during the simulation, rather than directly from the original input data. COMP3 contains several physics modules which are only invoked when certain atmospheric conditions exist. For example, in the cumulus clouds over Kenya, COMP3 models the thermal properties of water vapor; in the clear skies over the Sahara, it does not. To make matters worse, weather systems travel large distances across the Earth's surface, dragging along their special atmospheric conditions. As with Psirrfan, these computational irregularities are expressed as function calls and inner loops protected by conditionals.

The climate model has an additional important property. It updates its simulation state using a two dimensional 5x5 stencil (over latitude and longitude). This means that any scheduling algorithm must preserve the program's communication locality by mapping most adjacent grid elements to the same processor.

3 Context

This section gives a brief overview of our extended Fortran 77 compiler, and describes how we modified it to generate efficient code for irregular parallel programs, using the scheduling methods described in section 5.

The compiler recognizes the following two extensions for explicitly specifying parallelism: DOALL and PARALLEL SECTION. DOALL denotes that all iterates of a loop can execute simultaneously. PARALLEL SECTION identifies sequences of program statements that can execute independently. In addition, the compiler recognizes a subset of the Fortran D extensions for specifying data decomposition [14]. It handles BLOCKED and CYCLIC decompositions in one or two dimensions, and perfect alignment. The target architectures discussed in this paper are the Ncube-2 distributed memory multiprocessor (512 processors) and the Cray Y/MP shared memory vector multiprocessor (8 processors).

Compilation proceeds in four pipelined phases. In phase one, parallel operations are transformed to improve data locality and to reduce synchronization overhead. The compiler uses transformations including loop interchange [4] and loop skewing [34] to expose coarse grained parallelism [32]. In addition, phase one performs tiling transformations to improve the data locality of loop nests [31].

When compiling for distributed memory machines, phase one also generates code to partition arrays among processors according to programmer-specified decompositions. Following the convention of the Rice Fortran D compiler [14], our compiler uses the *owner computes* rule to make an initial assignment of loop iterations to processors. Under this rule, if an array element is the left hand side of an assignment statement, the processor holding that array element computes the right hand side of the statement.

Our runtime scheduling system uses the distributed algorithm described in section 5.2 to transfer ownership of array segments and their associated computations. This algorithm operates under the assumption that data has been laid out to maximize communication locality. When re-assigning computations, the algorithm avoids excessive data scattering by maintaining a minimum grain size, K_{min}. K_{min} determines the minimum number of iterations that can be transferred as a block from one processor to another. This

minimum grain size constrains the scheduler not to exceed the bandwidth requirements of the machine. On machines like the Ncube, which have a large message startup cost, the minimum grain size must also be large enough to keep the average number of messages per processor reasonably low.

Phase two of the compiler generates calls to the runtime communication library. It uses the dataflow algorithms described in [14] to compute *send* and *receive* sets for each parallel operation, and then uses this information to generate communication code. To support transfer of ownership, all messages follow a forwarding protocol [19] to find their actual destinations.

In phase three, the compiler generates *code blocks* for each parallel operation. The code blocks are parameterized by grain size. Phase four generates *scheduling templates* for each parallel operation. There are four types of scheduling templates: *sequential*, *static*, *profile*, and *sample*. The first of these executes the operation sequentially. The second uses the owner computes rule on distributed memory machines, and allocates tasks to processors evenly on shared memory machines.

The latter two use the dynamic scheduling algorithms described below. They differ in how they gather information about the distribution of task execution times. Profile templates use information gathered from previous runs of the program to estimate execution time mean and variance [27]. Sample templates begin with a rough estimate of these two quantities and refine the estimates through runtime sampling of task execution times. On the Ncube and Cray Y/MP, the code performs sampling by reading a memory-mapped microsecond clock. The scheduling algorithm preserves the output of sampled tasks, so that no repeated task execution is necessary.

4 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 [8] or combinatorial exploration of possible processor assignments [24]. 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 [28]. 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 in [27] to estimate execution time variance. Our scheduling algorithm is different in that it uses runtime information about execution time variance to determine the grain size at which a parallel operation is scheduled.

A somewhat different approach to scheduling is *load balancing*[26]. 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.

Yew and Tang have proposed a dynamic scheduling method called self-scheduling [29]. Self-scheduling follows the *first available* rule in assigning tasks to processors. Whenever a processor finishes executing a task, it 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 $N(\mu + h)/p$. For $\mu = h$, the contribution of scheduling overhead doubles the expected finishing time.

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 [16] for determining the optimal value of K, given N, p, σ, and μ. 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.

We would like to find a strategy that combines the low runtime overhead of large chunk sizes with the even finishing times of self-scheduling. In the next section, we investigate the 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) [23], 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 $\sigma = 0$, but does not address the problem of variable task execution times.

5 Tapering Methods

In this section, we present our method for selecting a chunk size K_i, given the number of remaining tasks R_i, number of processors p, and distribution(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 tapering method 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. Define $finish_{i,j}$ to be the time at which processor j has finished with any chunks numbered less than or equal to i. 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 for an example).

5.1 The Fill Lemma

First, we introduce a lemma that illustrates a key property of the first available rule. The goal of this 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,

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

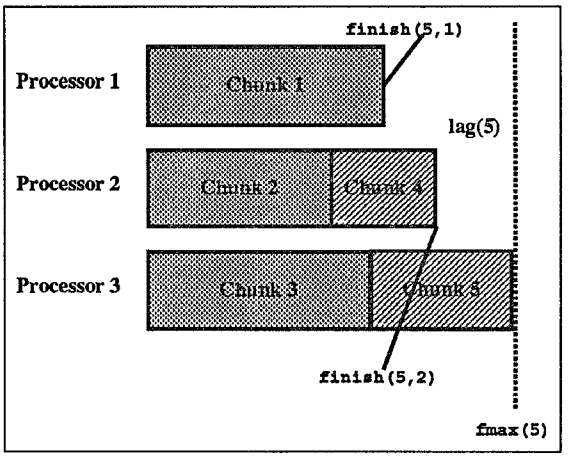

Figure 4: A Scheduling Scenario

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 $i+1$, $\sum_j(fmax_{i+1} - finish_{i,j}) \leq (p-1)cost_{i+1}$□.

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.*

5.2 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 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.

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 $\Pr[Z_i > b] = \epsilon$. 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), we have

$$\epsilon < \sigma^2(((p-1)^2 - 1)K_i + R_i)/(b - (pK_i - R_i)\mu)^2 \quad (1)$$

The partial derivative of (1)'s right hand side with respect to K_i is always positive. Thus, if we set the right hand side of (1) equal to ϵ 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) this approach is rapid.

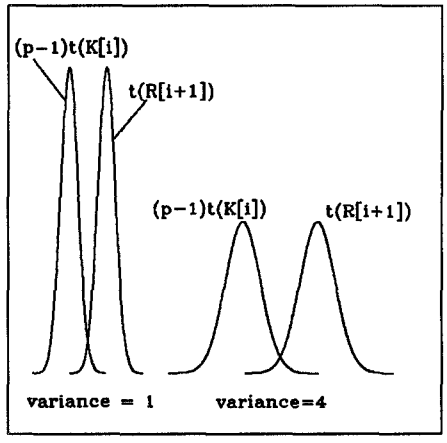

Figure 5: Illustration of K_i selection.

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

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

Let $T_i = R_i/p$, and $v_\alpha = \alpha\sigma/\mu$. Then (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, then solving for K_i yields

$$K_i = \left\lceil T_i + \frac{v_\alpha^2}{2} - v_\alpha\sqrt{2T_i + v_\alpha^2/4} \right\rceil \quad (3)$$

Figure 5 shows how equation (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 (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 (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, we would like to derive a single value of α

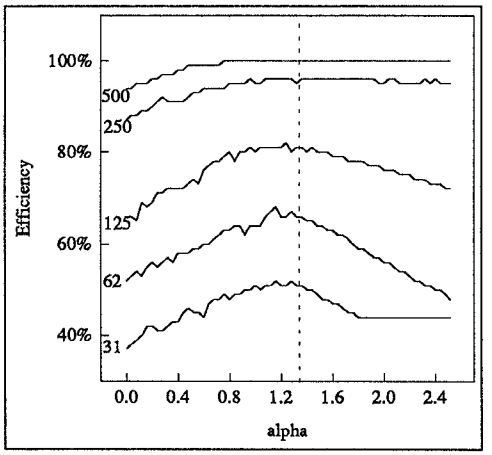

Figure 6: Effect of α. Lines labeled with N/p.

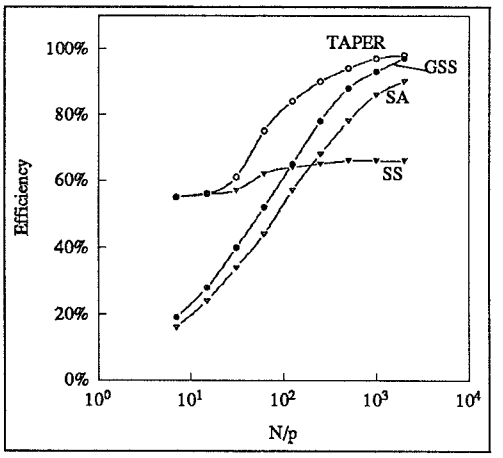

Figure 7: Binomial Distribution Pr[60000]=0.1, Pr[200]=0.9

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 6 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 6 also indicates that a scheduling method using (3) can withstand considerable inaccuracy in the value of $\frac{\sigma}{\mu}$, since v_α in (3) is just the scaled coefficient of variation $\alpha\frac{\sigma}{\mu}$. For example, we found that for all of the benchmark applications discussed in section 6, simply using (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 (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 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 section 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 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 $K_{min} = \min(K_{band}, K_{sched}, N/p)$.

When $K_{min} = N/p$, we can't improve on a static schedule for the parallel operation. However, we can modify equation (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 will be at least $K_{min}p\mu/2$. Hence, there is no reason to require $Z_i \leq 0$ and therefore b (the bound on acceptable lag_n) $= 0$ in the derivation above. If we set $b = K_{min}p\mu/2$, then equation (3) still holds, provided we set $T_i = \frac{R_i}{p} + K_{min}/2$. Our final expression for computing K_i becomes

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

Combining our techniques for selecting α and K_{min} with equation (4) we have an algorithm for dynamic scheduling. We call the algorithm TAPER. Figures 7 through 9 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,

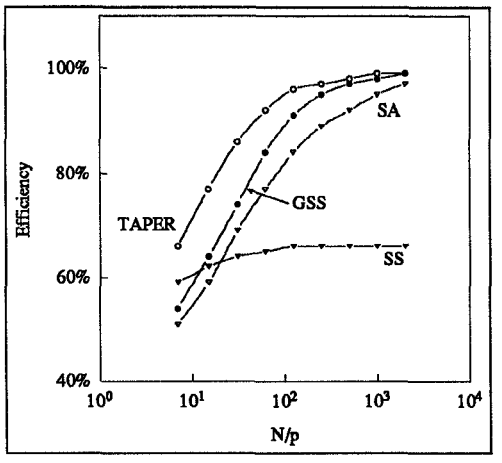

Figure 8: Uniform Distribution on [0,10]. Overhead = 0.5μ.

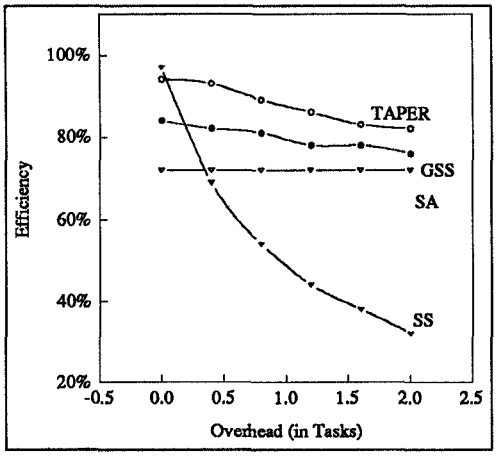

Figure 9: Performance on Pr[10] = 0.9, Pr[1] = 0.1 for different overheads.

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} chunk is assigned. Let $D_i = \frac{N}{p} - \frac{s_i}{\mu}$. This 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 10 illustrates how EVENSTART maintains a small unevenness in chunk finish times throughout a parallel operation.

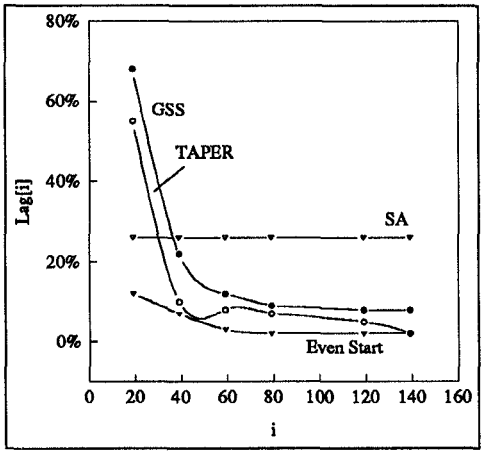

Figure 10: As chunks are scheduled, lag_i decreases.

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 [15]. 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.

Nonindependent Task Costs The above discussion assumes that task execution times are independent random variables. If execution times are correlated, then we have two choices. First, we can randomize the iteration space. That is we can randomly permute the tasks of a parallel operation so that we can use the independence assumption. This is possible for some programs like Psirrfan, where there is little communication locality. For the COMP3 benchmark, communication requirements make this transformation counterproductive.

Second, we can do additional sampling of the task costs to build a *cost function*. A cost function estimates task execution times as a function of iteration number. When determining K_i we begin with the estimate $K_i = \lceil R_i/p \rceil$. We then use the cost function to determine the distribution over the particular set of K_i tasks selected. Given the distribution estimate (μ,σ), we refine the value for K_i and obtain a new distribution estimate. To find the correct value for K_i we must scale the value we obtain from (4) by $s = \mu_g/\mu_c$. In this expression, μ_g is the global mean and μ_c is the mean for the tasks in the current chunk.

We have found the cost function technique to be extremely effective because most parallel operations have considerable *distribution coherence*. That is, successive executions of the same parallel operation (such as a parallel inner loop) will have almost the same cost function. Distribution coherence is a temporal locality property. Just as caches take advantage of memory access locality, we can take advantage of distribution coherence by cacheing our assignments of tasks to processors. On distributed memory machines, this cacheing takes the form of transferring blocks of array elements between processors. On shared memory machines, each processor stores its own task assignments.

Distributed Memory On distributed memory machines, we can no longer model the effect of scheduling a chunk of tasks with a fixed overhead h. 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 need to preserve communication locality by maintaining a minimum chunk size, K_{min}. We solve this problem by beginning with some original data decomposition and assigning tasks to processors according to the owner computes rule. As we gain information about the work distribution, we refine the data decomposition.

In the distributed algorithm the p processors are logically connected as a binary tree with p leaves. Some of the processors act as both leaves and internal nodes of the tree. We modify TAPER so that it chooses chunk sizes in *epochs* of p chunks. Adding this constraint to the derivation above yields the scheduling rule $K_j = \max(K_{min}, \lceil \frac{2}{3}T_j - v_\alpha\sqrt{3T_j + v_\alpha^2/4} \rceil)$ (where K_j is the number of tasks in the p chunks of the j^{th} 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.

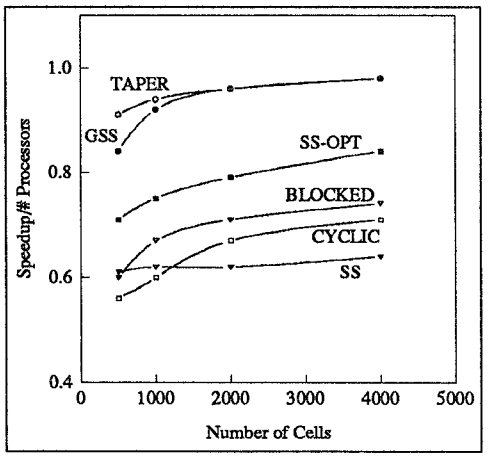

Figure 11: AMR on Ncube

Processors compete for the p chunks of each epoch. If processor a can get two tokens of value i to the root before processor b can send one token of value i, then the root will re-assign processor b's chunk of size K_i to processor a. Processor b is then forced to re-interpret the chunk it is currently executing as belonging to some later epoch (and thus containing fewer tasks). If most of the actual task cost is on a few processors, this scheme will degenerate into the centralized TAPER 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.

6 Benchmarks

This section gives performance results for several real applications that contain irregular parallel operations. For each application we give a brief description of its computational characteristics. The appendix gives graphs comparing the performance of GSS, SS, SS with optimal single chunk size (SS-OPT), static assignment (SA or BLOCKED and CYCLIC decompositions), and TAPER on each application. Efficiencies are reported against optimized sequential code. Cray results are for 8 processors, Ncube results are for 512 processors. To be fair to SS and GSS, we modified these methods to use the distributed algorithm given in section 5.2.

Our first application, AMR, is an adaptive vortex method for computing fluid flow [5]. The method uses a finer grid size wherever vortices are present. Its computational structure is similar to our climate example, but it has more floating point operations per memory

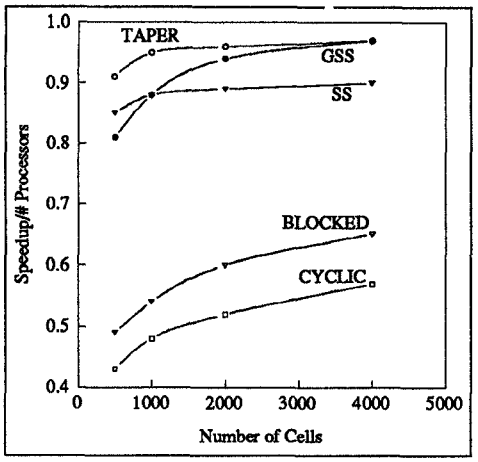

Figure 12: AMR on Cray Y/MP

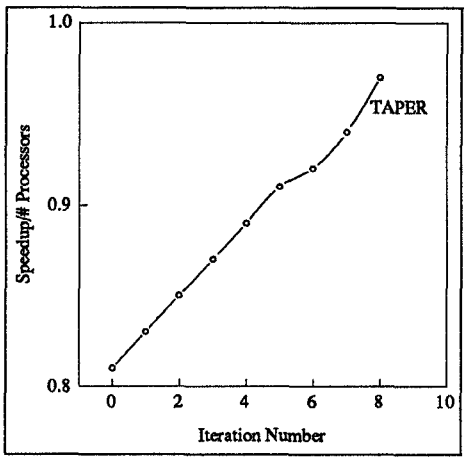

Figure 13: First few iterations of AMR inner loop on Ncube.

access. This example illustrates the power of distribution coherence. For the first few seconds of the short computation, TAPER builds the cost function. Then it reaches a point of maximum efficiency where the cost function needs only incremental improvement (see figure 13).

We have already introduced the features of COMP3, a section of the UCLA General Circulation Model [6]. Note that, for small data sizes, no method can execute COMP3 efficiently because of the climate model's communication requirements. We have also introduced Psirrfan, an x-ray tomography application. For Psirrfan, we include a comparison with a representative local load balancing algorithm [26].

Our final application, EMU, is a timing simulator that is part of the MULGA circuit design system [1, 20]. Unlike the applications seen thus far, EMU has an exponential work distribution. EMU divides a circuit into regions; elements of a particular region are connected by pass transistors. For each region, EMU

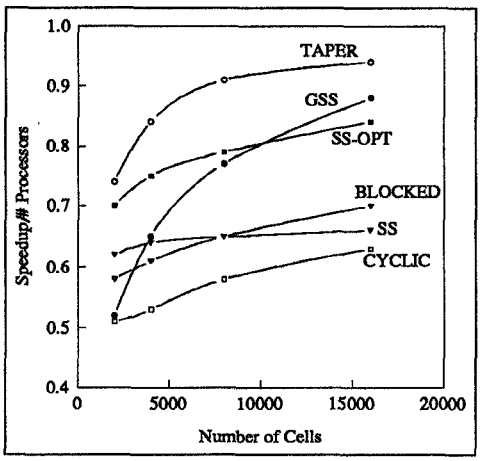

Figure 14: COMP3 on Ncube

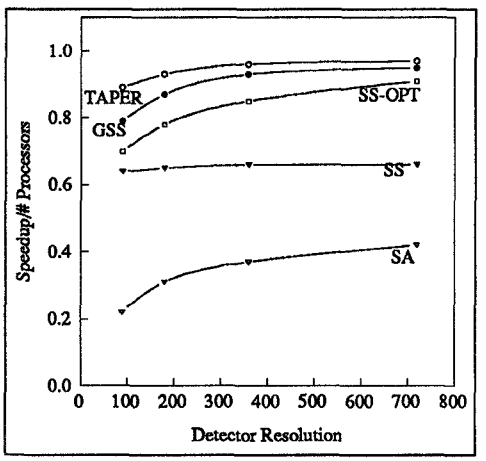

Figure 15: Psirrfan on Ncube

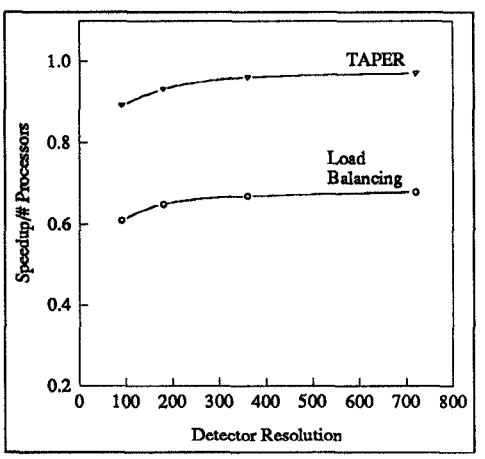

Figure 16: Psirrfan Load Balancing Comparison

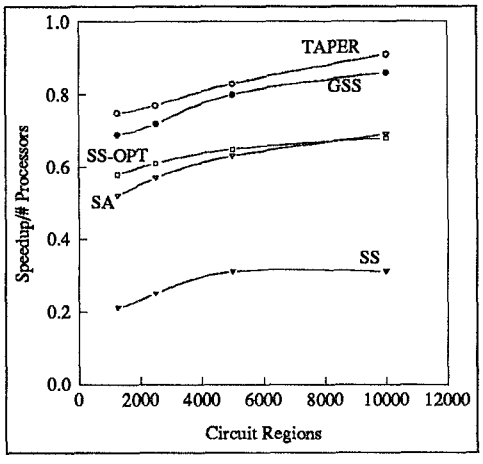

Figure 17: EMU on Ncube

uses a backward Euler integration to update voltage values. If the numerical method diverges, EMU subdivides the timestep and re-integrates.

7 Summary

In this paper, we introduced a methodology for compiling and executing irregular parallel programs. We developed a new dynamic scheduling algorithm for such programs and extended the algorithm to work on distributed memory machines. We also extended the algorithm to build *cost functions* for nonindependent task cost distributions. This modification was especially successful because of *distribution coherence*. If a parallel operation has distribution coherence, successive execution instances of the operation will have nearly identical work distributions. We modeled the effect of communication locality through a minimum grain size K_{min}; this method was sufficient for three real applications involving stencil communication. Finally, we developed a relationship between total available parallelism, optimal grain size, and execution time variance. We applied this relationship to the problem of choosing grain sizes for parallel operations.

We plan to continue this work by incorporating the compiler and runtime techniques described above into the Rice Fortran D compiler [14]. We are currently incorporating these techniques into a coordination language system [21, 22] and an object-oriented parallel language [18] that uses runtime techniques to optimize communication patterns [17]. We are also working on compilation techniques for statically discovering cost functions and for proving distribution coherence at compile-time. Finally, we plan to make available, in conjunction with the Advanced Computing Research Facility at Argonne National Labs, a benchmark suite of irregular programs.

References

[1] B. Ackland, S. Lucco, T. London, and E. DeBenedictis. "CEMU: A Parallel Circuit Simulator,". In *Proceedings of the International Conference on Computer Design*, October 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. "An Overview of the PTRAN Analysis System for Multiprocessing,". *Journal of Parallel and Distributed Computing*, 5:617–640, October 1988.

[3] J. R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. "PTOOL: A Semi-Automatic Parallel Programming Assistant,". In *International Conference on Parallel Processing*, pages 164–170, 1986.

[4] J. R. Allen and K. Kennedy. "Automatic Loop Interchange,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, June 1984.

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

[6] A. Arakawa and V. R. Lamb. "Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model,". *Methods in Computational Physics*, 17:173–265, 1977.

[7] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. "Interprocedural Constant Propagation,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 152–161, Palo Alto, 1986.

[8] R. Cytron. "Limited Processor Scheduling of Doacross Loops,". In *Proceedings ICPP*, pages 226–234, 1987.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and K. Zadeck. "An Efficient Method of Computing Static Single Assignment Form,". In *ACM Conference on Principles of Programming Languages*, pages 25–35, 1989.

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

[11] E. Gumbel. "The Maxima of the Mean of the Largest Value of the Range,". *Annals of Mathematics and Statistics*, 25, 1954.

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

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

[14] S. Hiranandani, K. Kennedy, and C.-W. Tseng. "Compiler Support for Machine-Independent Parallel Programming in Fortran D,". Technical Report TR91-149, Rice University, March 1991.

[15] S. F. Hummel, E. Schonberg, and L. E. Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops,". Technical Report 74172, IBM Research Division, 1991.

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

[17] S. Lucco. "A Heuristic Linda Kernel for Hypercube Multiprocessors,". In *Proceedings of the Second Conference on Hypercube Multiprocessors*, September 1987.

[18] S. Lucco. "Parallel Programming in a Virtual Object Space,". In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[19] S. Lucco and D. Anderson. "Tarmac: A Language System Substrate Based on Mobile Memory,". In *International Conference on Distributed Computing Systems*, 1990.

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

[21] S. Lucco and O. Sharp. "Delirium: An Embedding Coordination Language,". In *Proceedings of Supercomputing '90*, pages 515–524, November 1990.

[22] S. Lucco and O. Sharp. "Parallel Programming With Coordination Structures,". In *ACM Conference on the Principles of Programming Languages*, January 1991.

[23] C. Polychronopoulis and D. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,". *IEEE Transactions on Computers*, C-36(12), December 1987.

[24] C. Polychronopoulos and U. 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.

[25] C. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors,". In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 39–48, 1989.

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

[27] V. Sarkar. "Determining Average Program Execution Times and their Variance,". In *SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[28] V. Sarkar and J. Hennessey. "Partitioning Parallel Programs for Macro Dataflow,". In *ACM Conference on Lisp and Functional Programming*, pages 202–211, Cambridge, Mass., 1986.

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

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

[31] M. E. Wolf and M. S. Lam. "A Data Locality Optimizing Algorithm,". In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[32] M. E. Wolf and M. 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.

[33] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[34] M. Wolfe. "More Iteration Space Tiling,". In *Proceedings of Supercomputing*, pages 655–664, 1989.