**MIT Concurrent VLSI Architecture Memo 61**

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

# The Importance of Locality in Scheduling and Load Balancing for Multiprocessors

Stephen W. Keckler

February 4, 1994


## Abstract

This paper addresses the importance of locality when migrating tasks of a parallel program between processors for load balancing in a multiprocessor. Static and preprocessing task scheduling algorithms work well for certain applications, but irregular problems often require dynamic load balancing. Many heuristics have been developed for scheduling the proper number of iterations of a parallel loop on each processor. Other techniques dynamically balance load by relocating tasks from busy processors to idle ones. Most of these approaches, however, neglect the cost of migrating a task from the processor on which it was originally created or assigned. This paper proposes using data *affinity* to help determine which tasks should be migrated from a heavily loaded processor to an idle one. Affinity is a measure of cost of moving the task and can be generated by the programmer or compiler, or it can be computed at runtime. This paper is intended to fulfill the area examination requirement in the MIT Department of Electrical Engineering and Computer Science.

# 1 Introduction

The expectation of N-fold application speedup on N processors has been difficult to attain on large scale multiprocessors. Application speedup in multiprocessors, as compared to uniprocessors, is limited by four factors:

1. Inefficiencies in the parallel algorithm as compared to the best sequential algorithm.
2. Inherent sequentiality in sections of the parallel algorithm.
3. Overhead costs, such as communication delay and task management.
4. Idle time of processors due to load imbalance during parallel sections of the application.

The first two items are associated primarily with algorithm design and may be affected by parameters of the multiprocessor. For example, different algorithms may be chosen depending on the communication costs between processors. This paper focuses on load balancing and limiting its computation and communication overheads.

In order to effectively use the resources provided by a multiprocessor, an application must have enough tasks that can be computed in parallel, and those tasks must be distributed in such a way that all of the processors are busy computing useful work. Maximum speedup can be attained only if processor idle time is kept to a minimum. Task scheduling and load balancing techniques have been developed to try to keep all of the machine resources busy. These range from purely static techniques to purely dynamic. Any load balancing scheme, however, must minimize the overhead associated with managing the tasks.

Section 2 provides a survey of the classes of load balancing techniques, describing their strengths and weaknesses. Section 3 critiques the three papers assigned for this examination and summarizes the critical issues associated with effective load balancing and reducing its overhead. Section 4 proposes using *affinity* scheduling to exploit locality in load balancing. Section 5 summarizes the scheduling and load balancing problems addressed by the three case studies along with the suggestions made in this paper. Finally, Section 6 outlines future work and extensions to affinity scheduling.

# 2 Survey of Load Balancing Techniques

Based on Znati, *et. al.*'s [36] taxonomy of load balancing models, this sections discusses some of the more notable techniques for performing load balancing.

## 2.1 Static Scheduling

Static load balancing puts the burden of task placement on the compiler. This minimizes runtime overhead because no dynamic task control is required; tasks remain where they are initially assigned. An added advantage of this scheme is that the communication patterns of the application's execution graph can be statically mapped to the multiprocessor and optimized. Practical work in this area uses heuristics to get near-optimal performance, but optimal static scheduling of task graphs with precedence and communication edges is NP hard [3]. Many heuristics use variants of greedy critical path or list scheduling [14, 12]. Sarkar and Hennessy [31] use a greedy approximation algorithm which has a worst case performance of two times the optimal. In addition, they use profiling to determine task costs and frequencies when they are not known exactly at compile time. The weakness of static scheduling, of course, is that for many programs, the runtime data set is unknown, and the program's execution graph can not be accurately predicted. This is particularly true if different data sets result in vastly different dynamic execution graphs. Optimal static task scheduling also breaks down if the number of processors is not known at compile time, as is true for applications that are meant to be portable across multiple machine sizes without recompilation.

## 2.2 Preprocessing

Some applications have execution graphs that are not determined until runtime, but are reused multiple times. For example, a circuit simulator may represent the device connections as a sparse matrix, and solve the resulting system of equations, albeit with different coeffecients, at each time step. These types of applications can use a runtime preprocessing stage that dynamically partitions and schedules tasks. The overhead of preprocessing is amortized over the execution of the program. Saltz uses the *inspector/executor* model in which the inspector performs the task partitioning and data placement which is then used by the executor, terming this "run-time compilation" [30]. This research has been extended by attempting to dynamically detect when inspector results can be reused by the executor, and when the inspector must be invoked again [29]. Like static scheduling however, preprocessing techniques are useful only if a schedule can be determined without actually running the program. In addition, the schedule must be reused enough to amortize the cost of preprocessing. Since preprocessing and static scheduling techniques do not perform dynamic load balancing, they can suffer from processor idle time should the actual task latencies differ from those of the predicted schedule.

## 2.3 Dynamic Scheduling

Dynamic scheduling techniques are the most general and adaptable to dynamic task graphs and other programs that are not easily scheduled statically. Dynamic scheduling techniques are generally formulated using work queues. These may be represented as shared loop iteration variables, a true queue of tasks, or an implicit queue of tasks residing on a processor's runtime stack. These work queues can be *global* in which all processors access the same central queue, or *distributed* in which each processor has its own local queue. Both techniques must address the tradeoff between task size and load balance, in which large tasks reduce the task management overhead but can lead to greater load imbalance across processors.

### 2.3.1 Global Task Queue

A global task queue is a central data structure that each processor accesses in order to get work to perform. When accessing the queue, a processor may take a single task or a *chunk* of multiple tasks. After completing its work, the processor accesses the queue again to get more work. New tasks that are created are placed at the end of the global queue.

The best understood global queue strategies are the variances of self scheduling of parallel `DOALL` loops. In this execution model, each loop has a single global iteration variable that represents the work queue and can be accessed by all processors. The value of the iteration variable is determined at runtime by the size of the data set. A processor acquires work by accessing the counter and incrementing it by the chunk size, which is equal to the number of iterations it takes. Determining the optimal number of iterations, in terms of load balance, communication, and synchronization overhead, has been the subject of many research papers, including two that are a part of this exam.

Tang and Yew introduced *Self Scheduling*, a method in which each processor simply takes a single iteration at a time [33]. Self scheduling results in processor finishing at approximately the same time, but suffers from a scheduling overhead proportional to the number of loop iterations. Another variant of self scheduling allows each processor to take a chunk of $K$ iterations, where $K > 1$, to reduce some of the scheduling overhead.

Kruskal and Weiss [17] determine the optimal chunk size $K$ when task lengths vary randomly under different distributions with mean $\mu$ and standard deviation $\sigma$. $K$ is selected to limit the number of total accesses to the global iteration variable and cause all processors to finish at approximately the same time. Polychronopoulos and Kuck's *Guided Self Scheduling* (GSS) [28] allows $K$ to decrease, but assume that the task length is fixed. Hummel, *et. al.* [11] propose a refinement that provides better load balancing by dividing the iterations up into phases. Each chunk in a given phase contains the same number of tasks, but the chunk size in successive phases decreases. Tzen and Ni propose Trapezoid Self-Scheduling [34], in which chunk size decreases linearly. Lucco [21] uses a scheme similar to GSS, except that task sizes are modelled stochastically by $(\mu,\sigma)$. These techniques will be discussed in more detail in Section 3.

### 2.3.2 Distributed Task Queue

The advantage of a distributed task queue is that it is not a single shared resource for which all processors must compete. When a processor finishes with a task, it goes to its own local task queue to dequeue the next task. Tasks may be placed in a local task queue by the owning processor or by a remote processor. Some distributed schemes try to maintain and update at each processor information describing the load of the system, even on a processor by processor basis [24]. Other systems, such as the gradient model [20], keep no global state but maintain a gradient field in which tasks are passed downhill to more lightly loaded processors. Load balancing may be performed on local task queues by *offloading* or by *work stealing.*

**Offloading:** In an offloading (or bidding) scheme, processors alternate between two modes, executing and load balancing. During an execution phase, the processor removes tasks from its local task queue, executes them, and may create additional tasks, which are placed in the local task queue or in the task queue of another processor. A processor enters the load balancing phase if its task queue has too many or too few tasks, according to some upper and lower thresholds. If it has too few tasks, it requests work from other processors. If it has too many tasks, it sends work to other processors. Eager, *et. al.* [6] analyze the effectiveness of several adaptive load sharing techniques for distributed systems. These techniques differ in how much remote load information is required to determine where to place a new task. Kalé discusses two methods of load balancing that try to send work to lightly loaded processors within a local neighborhood [13]. These techniques are effective in the context of a distributed system in which tasks are long running compared to the load balancing overhead. However, in a parallel program with a high rate of dynamic task creation and relatively short running tasks, an otherwise busy processor that must perform load balancing may experience a large degradation in the rate at which it can complete useful work.

**Work Stealing:** In a work stealing system, processors fetch tasks from their own queues and execute them. If the queue is empty, the processor queries other processors, requesting some of their work. Thus only otherwise idle processors participate in any load balancing. Lazy task creation [23] uses work stealing. The local task queue is threaded through the execution stack and tasks are only added to the local task queue using the `future` construct of Mul-T [16]. WorkCrews [35] and optimizations to Qlisp [27] maintain local task queues and employ work stealing when processors become idle. All three of these schemes employ forms of lazy task creation to increase task granularity and reduce the overhead associated with task creation.

## 3 Case Studies

This section presents and analyzes the following three papers that address load balancing and scheduling of parallel programs:

1. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", by Polychronopoulos and Kuck [28].
2. "A Dynamic Scheduling Method for Irregular Parallel Programs", by Lucco [21].

3. "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs", by Mohr, Kranz, and Halstead [23].

The first two papers address the problem of scheduling iterations of parallel loops. The third paper examines a scheduling and load balancing method for parallel programs that create tasks dynamically.

## 3.1 Guided Self Scheduling

Polychronopoulos and Kuck describe a method for scheduling loop level parallelism that minimizes scheduling overhead and load imbalance. Loop level parallelism is expressed using the `DOALL` construct which indicates that all iterations of the loop are data independent and may be executed in parallel. The compiler performs optimizations, such as *loop coalescing* and *loop interchange*, to increase the amount of parallelism in nested loops, and converts all nested loop indices to a function of a single top level loop index. This single loop index serves as the global task queue, and processors access this variable to get iterations of the loop to execute.

Dynamically, processors execute the Guided Self Scheduling (GSS) algorithm to schedule loop iterations. A processor accesses the iteration variable to get the largest possible chunk of iterations that still guarantees that enough work will be left for other processors to execute. Thus the $i$th requester takes chunk size $K_i$ according to

$$K_i = \max(K_{min}, \lceil R_i/p \rceil) \tag{1}$$

where $R_i$ is the number of remaining iterations, $p$ is the number of processors, and $K_{min}$ is the minimum chunk size. This guarantees that the maximum difference of finishing times between any two processors is $K_{min} \times \mu$ where $\mu$ is the sequential computation time of an iteration. The scheduling overhead, which is equivalent to the number of accesses to the global iteration variable, is $O(p \ln N/p)$, where $N$ is the number of iterations. This compares favorably to Self Scheduling (SS) which requires $O(N)$ accesses to the global variable.

### 3.1.1 Critique

**Variable Length Tasks:** Guided self scheduling performs well on parallel loops with independent iterations whose execution times are all nearly equal. GSS is effective even when processors begin executing the loop at different times. However, the load balance can suffer greatly when iteration execution lengths have a large variance. For example, if the first $1/p$ iterations of the loop comprise $1/h$ of the work, where $h < p$, then the overall parallel speedup will be limited to $h$, since the processor getting these first iterations will be last one to finish. The simulation results in the paper comparing GSS to SS reflect this phenomenon. As a result of greater load imbalance, the relative advantage of GSS over SS decreases when task length variation is included in the simulations.

Several solutions have been proposed to reduce load imbalance when tasks have variable execution lengths. Lucco [21] computes the best task chunk size based on a measure of the task length variance. As discussed in more detail in the next section, when variance is large, chunk size is reduced. Hummel, *et. al* [11] divide the iterations into batches of $p$ equal sized chunks. This guarantees that all processors initially get the same number of iterations. Like GSS, the chunk size in each batch is reduced during execution of the loop. Both of these techniques improve load balance by decreasing the chunk size, and incur higher synchronization overhead as a result.

As another solution to the uneven task length problem, I suggest using task redistribution. Thus, if a processor finds the work queue empty, it queries other processors for unfinished iterations. This can be implemented in a parallel loop by keeping the local iteration counter used by each processor in memory

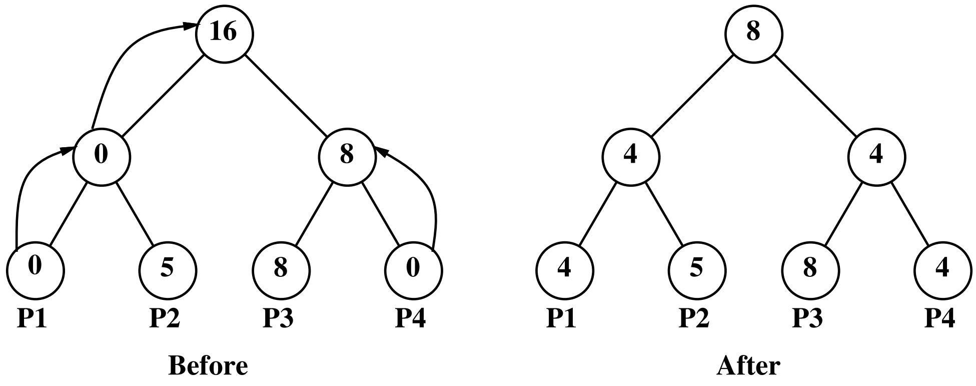


Figure 1: The distributed tasks queue before and after `P1` and `P4` request work. The number of iterations in a task queue is shown inside each internal node.

---

and synchronizing access to it. An idle processor can steal work by incrementing a busy processor's local iteration counter, similar to how iterations are acquired from the global iteration counter. Dividing the chunk into fixed size tasks that may be stolen, instead of single iterations, would help to reduce the overhead of synchronizing on the local iteration counter.

**Synchronization Overhead:** GSS does reduce the number of synchronization operations required to access the shared iteration variable ($O(p \ln N/p)$ for GSS instead of $O(N)$ for SS). However, the counter remains a central resource for all processors to access. In a small scale shared-memory multiprocessor or vector processor, the communication and synchronization overheads are likely to be small. However, in a large scale distributed memory machine, communication costs are higher and contention for the global variable can be quite large, especially at the end of the loop where $p$ processors each gets $K_{min}$ iterations. For fixed $N \gg p$, the synchronization overhead of GSS grows linearly with $p$; thus GSS does not scale well. This phenomenon is not reflected in the GSS simulation results, as the authors have fixed the overhead to a constant independent of $N$ and $p$.

I suggest using a distributed global work queue with GSS to provide better scalability. A tree is embedded in the processor network where the internal nodes of the tree hold *local* ranges of iterations representing a piece of the global task queue. If an idle processor asks its parent for work and none is found, the request traverses up the tree until a non-empty task queue is located. A portion of the task queue is taken and delivered back down the tree. Pieces of the task queue are left in the empty queues at the internal nodes of the tree. Subsequent requests for work from other processors in the subtree need not travel to the root since requests may be satisfied by internal nodes between the requester and root. Work requests can be combined at internal tree nodes. Using the GSS algorithm, the size of the chunk taken from any node is $R_i/b$ where $R_i$ is the number of remaining iterations in the work queue and $b$ is the branching factor of the tree. In order to prevent too fine a subdivision of tasks the minimum chunk size acquired at each internal node should depend on its height. Thus, at the root the minimum chunk size is $K_{min}p/b$, while the minimum chunk size at the parent of a leaf is $K_{min}$. Figure 1 shows the task queues before and after `P1` and `P4` get additional work. Like GSS, enough work for all the processors in the subtree is left at the internal nodes.

This hierarchical distributed work queue reduces the contention at each node to the constant branching factor $b$. As long as task lengths are similar, processors within each subtree are well balanced and this algorithm should result in load balance similar to GSS, while providing more scalable synchronization behavior. One drawback to this scheme, however, is the additional latency required for traversing multiple levels of the tree. Also, there will be more *tails* in which processors get only $K_{min}$ iterations. This can be alleviated somewhat by using thresholds on the length of internal tree node queues to determine when to search higher in the tree for work.

**Data Locality:** In a distributed memory machine, if iterations are executed on the processors that initially have much of the data they need, then communication overhead and latency will be reduced, resulting in better execution time. Furthermore, if multiple iterations access the same data, communication requirements can be reduced by executing them on the same processor. Locality is a difficult issue to address, as it must be weighed against load balancing [22]. If the data is striped across the processors, a block scheduling of tasks to processors can minimize communication. However, the schedule produced by GSS will ignore this structure of the data and require more communication. GSS does produce a schedule in which it is extremely likely that consecutive loop iterations are placed on the same processor, which can lead to data reuse between iterations. This property can be maintained in the tail of the schedule by increasing the minimum number of iterations assigned in a chunk. SS does not see this benefit as successive iterations are assigned to different processors. Of course, the chunk size in SS can be increased as well.

## 3.2 Irregular Parallel Programs

Lucco extends Guided Self Scheduling to perform well on irregular problems. He defines irregular problems to be `DOALL` loops whose iterations vary significantly in execution time. Like Kruskal and Weiss [17], task lengths are modeled stochastically with ($\mu$, $\sigma$), but chunk size $K$ is allowed to decrease over time. His analysis derives the following equation for chunk size $K_i$ given to the $i$th requester:

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

The parameters of the above equation are:

- $K_{min}$ = the minimum number of tasks in a chunk. $K_{min}$ must be large enough to sufficiently minimize the per chunk scheduling overhead and the communication required between processors with adjacent chunks.
- $R_i$ = the remaining number of tasks
- $p$ = the number of processors
- $T_i = \frac{R_i}{p} + K_{min}/2$
- $v_\alpha = \alpha\sigma/\mu$
- $\alpha$ = an empirically determined coefficient

This equation reduces to equation (1) of GSS when $\sigma$ and $K_{min}$ are zero. The task length distribution parameters ($\mu$, $\sigma$) can be determined by profiling or through on-line estimation. Lucco introduces TAPER, the scheduling algorithm based on equation (2). Analytical results are presented comparing TAPER, GSS, and Self Scheduling (SS) for different task length distributions; experimental results are used to validate the models.

This paper addresses several of the problems of GSS. To improve load balancing when the variance in task lengths is large, the chunk size based on equation (2) is reduced (as task length variation increases, so does $\sigma$, and hence $v_\alpha$). This allows the algorithm to adapt to run length variation, maximizing load balance while still maintaining the minimal number of synchronization operations. Lucco compares the performance of TAPER to GSS and SS using his analytical model and various task length distributions. When $N/p$ is small, load balancing is more important than overhead reduction. Thus SS performs better than GSS, as does TAPER since the chunk size is smaller. As $N/p$ increases, reducing the chunk scheduling overhead becomes more critical. The early chunk sizes for TAPER increase to near those of GSS, thus reducing the task scheduling overhead. Because GSS has a larger chunk size in the tail of the loop, resulting in greater imbalance at the end, TAPER still performs better. SS performs poorly because of the small fixed chunk size.

### 3.2.1 Critique

TAPER addresses the issues of task length variation and correlated task lengths, and even has an implementation for a distributed memory machine that tries to maximize locality. However, there are some deficiencies relating to implementing TAPER on large scale parallel machines.

**Synchronization Overhead:** The algorithm for distributed memory machines assumes an initial task placement to processors based on the data distribution, using the owner-computes rule. A combining tree is embedded in the processors for communication and broadcast. In the $j$th *epoch* each processor first sends its identifier to the root and then gets a chunk of $K_j$ iterations from the local queue. When the root has received $p$ tokens, it increments an epoch counter and broadcasts the new value to all of the processors, indicating that the next epoch has started. $K_j$ decreases with successive epochs. The root detects when processors fall behind and reassigns some of their work in between epochs. Using this algorithm, TAPER does confront issues of locality and work redistribution for load balance. Unfortunately, the discussion of this distributed algorithm is incomplete. The author implies that a root processor must maintain a table indicating the epochs in which processors have finished executing chunks. Also, it appears that task reassignment may only be performed by the root processor. The centralized processor status table and task redistribution strategy can result in a bottleneck for large systems that become imbalanced.

Several improvements to Lucco's distributed algorithm can be made. First, the centralized epoch counter and processor status table could be maintained within the tree. The epoch counter and processor status table at the internal nodes of the tree would indicate if a processor in the subtree has fallen behind. Internal nodes can reassign tasks within their subtrees, relieving the root of much of task reassignment responsibilities. Second, tasks should not be redistributed unless processors become idle. Tasks could be transferred to and from the some processor if it receives additional work during one epoch and then lags behind during another. A more efficient solution is to have idle processors search for work in a hierarchical fashion. When looking for more work within an epoch, a processor need only search to the least common ancestor of itself and a lagging processor. If the global resources are distributed throughout the machine, load balancing and task redistribution become more scalable with increasing numbers of processors.

**Locality:** Locality is not addressed in the shared-memory implementation of TAPER. The distributed version assumes that tasks are originally placed near their data, and it tries to maintain this mapping throughout execution. However, all tasks are treated as equally expensive to migrate, even though some might execute more quickly than others on the original processor. In addition, the cost of task migration is ignored when determining how many tasks to migrate. When transferring a set of tasks, the number of tasks could be reduced to account for the time required to transfer. Thus if $k$ iterations can be executed during the time required to transfer a chunk of size $K_i$, then transferring approximately $K_i - k$ iterations would be beneficial. This is particularly true if the transfer can occur without disturbing the computation at the lagging processor. A communication coprocessor, whether in a cache coherent shared memory machine [1, 19], or in a message passing system [25] allows data to be transferred without interrupting processor execution and would be useful here.

**Additional Overhead:** TAPER calculates a more complex function than GSS when determining how many iterations belong to chunk $K_i$ and incurs additional overhead in dynamically sampling the task length parameters ($\mu$, $\sigma$). However, these overheads are relatively small compared to the communication and synchronization overheads seen by both for centralized task distribution.

**Dynamic Task Creation:** Like GSS, TAPER is designed to handle loop parallelism. Neither GSS nor TAPER addresses dynamic task creation (adding tasks to a global or distributed task queue) or functional parallelism. In fact, one of the reasons that these techniques work well is that tasks are represented simply by lower and upper loop bounds. Thus, there is no overhead associated with handling frames that might hold an arbitrary task. These loop scheduling techniques are well suited for the (usually) data parallel jobs for which they were designed, but control parallelism has additional complications in limiting overhead in task management, scheduling, and load balancing.

## 3.3 Lazy Task Creation

Mohr, Kranz, and Halstead describe lazy task creation as a method of increasing grain size of parallel programs, and how it may be used to balance load across processors. Lazy task creation relies on the `future` construct of Mul-T [16] in which the child computation contained within the future may execute in parallel with the parent. If the parent needs to use the result from the child before the child has finished, the parent blocks. Unlike eager task creation in which tasks are created for every future call, lazy tasks are created only if additional parallelism is required to keep processors busy. In the implementation of lazy tasks, when a `future` function is called, a pointer to the parent's stack frame is placed at the head of the local lazy task queue, and a new frame is allocated for the child. Processors that become idle steal work from busy processors by taking parent continuations from the end of the lazy task queue. If the child finishes and its parent has not been stolen, it returns as if it were a sequential procedure call. If the continuation has been stolen, a forwarding pointer is found and the result is sent to the processor that now owns the parent. Lazy tasks are added to and removed from the head of the queue by the local processor. Continuations are stolen from the tail of the queue, corresponding to activations at the bottom of the stack. Activations lower in the stack are those tasks higher in the dynamic call tree and are expected to generate more work for the processor stealing them than tasks at the leaves. The paper compares lazy task creation to eager task creation and load-based inlining. Load-based inlining turns parallel tasks into sequential calls at task creation time if system load is high. Qlisp provides load-based inlining and mechanisms for the programmer to dynamically check the system load [8, 9]. On a synthetic benchmark, load-based inlining eliminated 90–95% of the eager tasks, while lazy task creation eliminated 99%, resulting in improved performance. Similar results were seen on several benchmark programs run on an Encore Multimax and the Alewife simulator [1].

### 3.3.1 Critique

Unlike the scheduling algorithms described in the first two papers of the case study, lazy task creation is used to manage control parallelism and dynamic task creation. The authors propose solutions to problems not found in `DOALL` loops including those of resource management and dynamic task migration for load balancing. By creating tasks only when additional parallelism is required, the overhead of task management is reduced. Similar ideas for limiting parallelism in dataflow programs has been explored in [4]. Lazy task creation provides an efficient solution to task management for a particular class of programs. However, there are some residual sources of overhead and some issues of scalability and locality that need to be addressed.

**Additional Overhead:** As the authors point out, lazy task creation places some additional latency on `future` calls that are not evaluated in parallel, since the stack frame must be linked and unlinked from the lazy task queue. Unexercised futures would have been faster if `future` had not been used. This overhead is difficult to eliminate if a large degree of parallelism is to be exposed. On the Encore, future and stack checking places a 40–120% overhead on a set of sample Mul-T programs. If the compiler could identify where to change from `future` calls to regular procedure calls (for example in a tree search) much of this overhead could be eliminated. However, because of the dynamic nature of the execution graph, this can be difficult to do. The authors note that this overhead is not incurred on Alewife since hardware tag checking and trapping eliminates the need to check in software. In either case, the larger overhead of eager task creation has been eliminated.

Another source of overhead comes from the stack frame implementation. Linked stack frames are more expensive to manipulate than a linear stack, but they make frame management and storage reclamation much easier when tasks are stolen. This tradeoff is more flexible in the current implementation of Mul-T on Alewife, as stack segments are used [15]. The stack segment length is fixed at several frames so that the overhead of manipulating individual stack frames is not incurred on every procedure call.

**Parallel Loop Iterations:** Lazy task creation does not perform well on iterative loops. For example, if the programmer writes a loop with a `future` around the body, the continuation for the rest of the loop may migrate around the machine leaving only a single iteration, or a small chunk of iterations if the loop is unrolled, running on any one node. In this event, load balancing is sequentialized since only a single continuation is available to be stolen. As the authors discuss, loops may be restructured by the programmer or the compiler in a divide and conquer fashion. A parallel loop construct such as `DOALL` would indicate when the compiler is allowed to restructure a loop. However, executing divide and conquer can be inefficient, since no loop iterations are executed until the leaves of the computation tree are reached. Instead, we would like to minimize this overhead by executing parallel loops in a self scheduled manner, perhaps like those methods described by the first two case studies. A means of spawning tasks on remote processors could be used here to efficiently initiate parallel execution of the loop.

One way to fit self scheduling into the framework of lazy task scheduling is to replicate continuations, rather than steal them. When a parallel loop is reached, the processor entering the loop begins executing iterations directly. When an idle processor requests work, the parent continuation is replicated, instead of being stolen, and the loop bounds for the iterations remaining on the processor and for those iterations migrating to the idle processor are adjusted. At the new node a different set of iterations begins executing, and any remaining idle processors can now steal iterations from either busy processor. The local iteration counters are stored in memory on each processor and access to them is synchronized. Multiple iterations could be chunked together to reduce the overhead of accessing the local loop counter.

**Scalability:** The issue of scalability is not addressed in lazy task creation. Since only small systems were used to run experiments (16 nodes on both the Encore and the Alewife), contention and traffic that might be seen in large scale systems was not encountered. Nussbaum proposes a scalable method of performing runtime load balancing with which lazy task creation fits well [26]. He embeds an X-Tree in the multiprocessor network and has idle processors search for work hierarchically. Work requests are combined at interior nodes of the tree and the work is divided across multiple requesters as it is delivered back to the leaves of the tree. He discovered that on small machines, almost any balancing scheme that avoid hot-spots performs acceptably, but on large machines ($p > 256$), hierarchical methods show substantial performance gains.

**Locality:** Finally, as with the loop scheduling methods of the first two case studies, lazy task creation does not address data locality. When an idle processor steals work from a loaded processor, it takes the first task from the lazy task queue, ignoring other tasks in the queue that might be less expensive to migrate. The implementation of lazy task creation is likely to be acceptable since the tasks at the head of the lazy task queue are the oldest continuations, and in a tree search are nodes high up in the call graph. These tasks are less likely to have data in common with a leaf computation than two adjacent leaves. However, if there is a wide variance of task migration costs, performance could be enhanced by stealing the cheap tasks.

### 3.4 Summary of Case Studies

The first two case studies address self scheduling of parallel loop iterations to minimize synchronization cost and load imbalance. Neither of these techniques is scalable, or is applicable to control parallelism and dynamic task creation. The third case study describes an efficient method for management of dynamically created tasks, but the solution is incompatible with loop execution. The issue that all three case cases share is locality: all of them ignore it. In the critiques of the case studies, this paper has suggested improvements to help overcome their individual weaknesses. Section 4 focuses on using locality in the load balancing process.

As a result of examining these cases, the following issues are identified as being crucial to scheduling and load balancing in a large scale parallel computer. They focus on keeping processors busy doing useful work and on reducing the overhead of performing scheduling and load balancing.

**Initial task placement:** Initial task placement is critical to reducing the overhead associated with load balancing. Ideally tasks are run where they are created or at least queued where they are to run. If all tasks are placed optimally, then no load balancing is required. However, ideal task placement depends on several factors, including the location of the data a task will use and the availability of processor resources. If the compiler can determine the computation pattern and schedule the work accordingly, runtime overhead will be greatly reduced. Dynamic task placement can result in large overheads if too much time is spent analyzing and updating records of the state of the system used to determine the best location for a task.

**Task creation:** Since task creation overhead is non-zero, only tasks that must execute in parallel should be created. Execution time will be reduced if tasks that are eventually executed serially do not incur the cost of task creation. Similarly, if too many tasks are created, the memory resources of the machine may be consumed entirely. Thus, only enough tasks to keep all of the processors busy should be created.

**Task migration:** Task migration for load balancing is necessary when the task creation and placement strategy is not able to spread work out evenly across the processors. Since any time spent performing task migration is time not spent doing useful work, only idle processors should initiate task migration. Those processors executing the application must be allowed to continue with as few interruptions as possible. Ideally, task migration is done without interrupting computation on a heavily loaded processor.

**Scalability:** Finally, a load balancing algorithm must be scalable, meaning that contention and synchronization should grow sublinearly with the number of processors. This has two implications. First, a distributed task queue must be used. A centralized task queue quickly becomes a bottleneck for scheduling tasks in large scale multiprocessors. Second, searching for work should be performed hierarchically as described in [26]. A tree-based algorithm eliminates work distribution hot spots as requests for work are combined going up the tree and work is split among idle processor groups on the way back down.

# 4 Locality and Load Balancing

Locality and load balancing are two competing goals in a parallel program. However, both are necessary to achieve high performance. For example, if tasks are spread evenly across the processors without regard for their data requirements, time and network bandwidth will be consumed in order to deliver the data that each task needs. Furthermore, if two tasks needing the same data are placed on different processors, data may be moved or copied many times. If, on the other hand, tasks are executed on those processors containing most of their data requirements, or if data is shared among many tasks, some processors may become overburdened while other processors sit idle. The goal, then, is to use locality information while performing task placement and load balancing to minimize the overhead of data transfer, while still keeping all processors busy.

This section proposes using task affinities when deciding which tasks to migrate during load balancing. The sources of tasks affinities are presented along with a simple task management strategy using affinities. Methods for statically and dynamically detecting task affinities are discussed. Finally, an analytical model of a simple load balancing problem is presented to illustrate the role of affinity scheduling.

## 4.1 Affinity Scheduling

The cost of moving a task from a source to a destination processor consists of the time to transmit the state of the task and the time to deliver all of the data required by the task that already resides on the source. This cost is a measure of the task's *affinity* for the source processor. If the task migration cost is large compared to the execution time of the task, the task has high affinity for the processor. If the migration cost is small, the task has low affinity. High task affinity can arise from several sources:

- Task state is local
  - The task has previously been active on the processor and is waiting to be reactivated.
- Task data is local
  - The task and its data is initially colocated on the processor.
  - Data required by the task is generated or retrieved by other tasks on the processor.

A task's affinity can be for data in the cache, and in a distributed memory machine, for the contents of the processor's local memory. Because of the non-zero cost of migrating a task, higher performance might be achieved by sacrificing some load balancing, as it may be more efficient to execute a task directly on the source instead.

Squillante [32] proposed having the operating system consider a task's cache affinity when scheduling and migrating processes. His algorithm uses the time since the task's last activation as a metric of affinity. Since he is scheduling large tasks in the operating system, he can afford to spend more time calculating and monitoring affinities. In this paper we address parallel programs in which tasks are lighter weight and are scheduled by the application, rather than the operating system, to reduce overhead.

Consequently, task affinity must be determined quickly at the time of task creation or suspension. The compiler may be able to help here either by assigning a static affinity value or by providing a simple method of computing it at runtime. A task's affinity is based on an estimate of the amount of resident task state and where the data it requires is located. A task might have strong affinity for the processor on which it is created. A set of tasks on the same processor that access the same data all might have strong affinities even if the data is not immediately located on the processor. When selecting tasks to migrate for load balancing, tasks with low processor affinity will be preferred to those with high affinity.

The simplest scheme uses binary affinities and two task queues on each processor: a *here* queue and a *there* queue. When a task is created, it is placed in one of the queues. If it is believed to have great affinity for the processor on which it was created, it is placed on the *here* queue. If the cost to move it elsewhere is anticipated to be less than some threshold, it is placed on the *there* queue. When a processor finishes with a task, it dequeues the next task from its *here* queue. When the *here* queue becomes empty, it executes tasks from its *there* queue. When a processor becomes idle, it steals work first from other processors *there* queues, and if necessary, from their *here* queues. Thus a processor prefers to steal work that has less affinity for the processor on which it resides. However, load balancing is preferred over locality, as tasks with high affinity will be stolen if necessary. At the previously idle processor, the tasks with high affinity are executed first, as they are most costly to migrate and should not be moved again.

## 4.2 Determining Task Affinities

In order for affinity scheduling to be effective, an inexpensive means of determining task affinity is required. Ideally, information regarding task affinity can be supplied by the compiler, providing some semantic information that would otherwise unavailable or too expensive to compute at runtime. Data access analysis has already been used to optimize cache performance in uniprocessors [18] and to partition parallel loops in multiprocessors [2]. Similar analysis may be used to help determine which loop iterations share the greatest

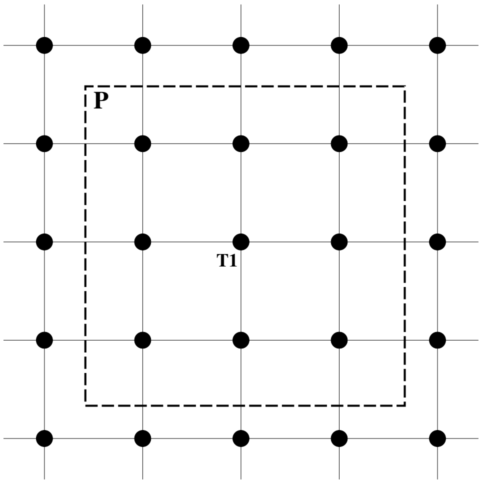


Figure 2: Task `T1` has the highest affinity for physical processor P. The tasks on the edges have lower affinity, while the tasks in the corners have the lowest.

amount of data, both in the cache and in local memory. The tasks with less reuse would be less expensive to migrate to another node should the load become imbalanced.

Take for example the task graph of Figure 2. The vertices of the graph are tasks, while the edges connect tasks which access a significant amount of the same data. The data is initially spread across the processors and the tasks are initially assigned to minimize data transfer. The dotted box circumscribes a set of tasks that are assigned to processor `P`. Task `T1` has the highest affinity to processor `P` since all of the data it requires is local. The affinities for a task on the edge is lower, since only 75% of its data is local, while a task in the corner has the lowest affinity, since only 50% of its data is local. A low affinity task will require less overall data to be transferred across the communication network if it is migrated to another processor. In a binary affinity scheme, the corner tasks would be placed in the *there* queue, while the other tasks would be placed in the *here* queue.

For dynamic parallelism, the job of determining task affinities is more difficult to perform at compile time. Heuristics may be used here to help dynamically determine a task's affinity when it is created. In many programs, a task has affinity to the processor on which it is created. If that task is migrated, it no longer has affinity to the processor on which it resides. Thus tasks that are migrated once are more likely to be migrated again, since they have not had the opportunity to gather data at their new home. In tree search problems, basing the affinity on the height of the tree can be effective. Because tasks toward the leaves have been more recently active, more of their data is likely to reside in the cache, and data reuse between siblings is likely. Lazy task creation already supports this with the double-ended lazy task queue. Determining a task's affinity is not unlike the problem of selecting RPC, data migration, or computation migration as discussed in [10].

## 4.3 Analytical Model

Consider a simple model in which all tasks begin on a single processor in a $p$ processor system. If executed on that processor, each task runs in time $t$. Tasks that have affinity for this processor incur a cost $h_m$ if migrated to another processor. Tasks without affinity incur a cost $h_n$ when executed remotely. The overheads $h_m$ and $h_n$ represent migration costs including moving task state and data. Table 1 summarizes the notation used in this section.

$N$ = number of tasks
$p$ = number of processors
$t$ = task execution time
$T_d$ = time to execute tasks on destination processor
$T_s$ = time to execute tasks on source processor
$T$ = time to complete all tasks
$m$ = number of expensive tasks stolen per destination
$h_m = c_m t$: overhead of stealing expensive task
$n$ = number of cheap tasks stolen per destination
$h_n = c_n t$: overhead of stealing cheap task
$r = m/n$: ratio of expensive to cheap tasks stolen

Table 1: Notation used in analytical model.

Ignoring contention, after stealing tasks from the source processor, the time $T_d$ to execute the migrated tasks on each of the $p-1$ destination processors is:

$$T_d = m(t + h_m) + n(t + h_n) \tag{3}$$

where $m$ is the number of tasks with affinity migrated to each processor, and $n$ is the number of tasks without affinity migrated to each processor. The time $T_s$ to execute the remaining tasks on the source processor is:

$$T_s = Nt - (m + n)(p - 1)t \tag{4}$$

where $N$ is the total number of tasks. If the optimal number of tasks $k = (m+n)(p-1)$ are migrated, then $T = T_s = T_d$ and the following equation expressing the total execution time can be derived:

$$T = \frac{Nt(rt + t + rh_m + h_n)}{prt + pt + rh_m + h_n} \tag{5}$$

where $r = m/n$, the ratio of expensive to cheap tasks migrated. The optimal execution time of $N$ tasks on $p$ processors is $Nt/p$ if overheads are ignored. If the overheads of task migration are $h_m = c_m t$ and $h_n = c_n t$, the following equation expresses the slowdown of parallel task execution from optimal as a function $r$, $p$, $c_m$, and $c_n$.

$$\frac{T}{Nt/p} = \frac{prc_m + pc_n + pr + p}{rc_m + c_n + pr + p} \tag{6}$$

If $c_m = c_n = 0$, then $\frac{T}{Nt/p} = 1$, meaning that ideal speedup is achieved. Also

$$\lim_{c_m, c_n \to \infty} \frac{T}{Nt/p} = p \tag{7}$$

Thus, as the task migration costs become large no speedup (slowdown $p$) results, since all tasks are executed on the source processor. As $r$ increases, more of the tasks migrated have affinity to the source and slowdown increases. Figure 3 shows a graph of the slowdown, equation (6), as a function of $r$. The number of processors $p$ is set to 1024 for a fairly large parallel machine. The task migration overhead for tasks without affinity $c_n$ is fixed at 0.05 (5% of task execution time), while the cost for migrating tasks with affinity $c_m$ ranges from 0.1 to 10. The graph shows that if tasks only have slight affinity for processors, then there is little performance degradation, regardless of the which tasks are migrated. However, if there is a sharp penalty for migrating tasks with affinity, then the slowdown can become significant. For $p \gg c_m$, the slowdown

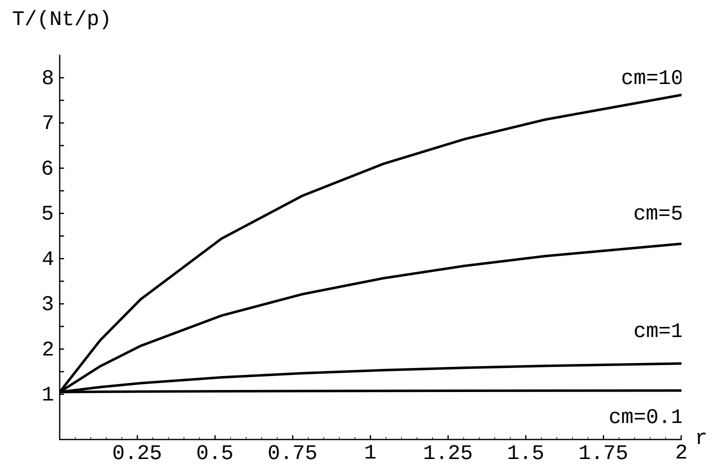


Figure 3: The slowdown ratio $\frac{T}{Nt/p}$ is shown as a function of $r$, the ratio of stolen tasks with affinity to those without, and $c_m$, the overhead (as a percentage of task execution time) of executing a task with affinity on a remote processor. The overhead of executing a task without affinity on a remote processor is fixed at 5% of task execution time ($c_n = 0.05$). The number of processors $p$ is set to 1024.

approaches $c_m + 1$, with increasing $r$. Also, just migrating a few tasks with affinity ($0 < r < 1$) results in a significant performance penalty.

#### 4.3.1 Chunk Size Stolen

The overhead of work stealing is reduced when an idle processor steals multiple tasks. This also helps load balancing since subsequent idle processors need not contend for a single source of tasks; tasks get distributed by processors participating in load balancing. A common heuristic is to steal half of the work from a processor. However, it is difficult to determine what constitutes half of the work, even if task execution times are predictable. The time to execute a migrated task is a function of its execution time on its original home and the amount of data that needs to migrate with it. If an estimate of the task execution time is available, the cost to migrate each task can be used to determine the optimal amount of work to be migrated such that the total execution time is minimized. The analysis in the previous subsection assumed the optimal number of tasks were stolen, based on $r$ and the cost parameters of the tasks with and without affinity. The optimal number of tasks stolen, $k$, can be expressed as:

$$k = \frac{(p-1)(r+1)Nt}{prt + pt + rh_m + h_n} \tag{8}$$

Thus, as the task migration overheads increase, the number of tasks migrated becomes smaller.

#### 4.3.2 Extensions to the Model

This model is extremely simple, assuming predictable execution times and affinities. A more robust model can use probabilistic estimates of task affinities and costs, as well as the ability to predict a task's affinity. In addition, a more realistic initial task placement, or even dynamic task creation could be included. However, this simple model is easy reason about and does display the importance of the migration costs and the ratio of expensive to cheap tasks migrated.

# 5 Conclusions

This paper described several load balancing techniques, ranging from purely static to dynamic strategies. Two of the case studies of Section 3 propose to optimize parallel loops by by parcelling out chunks of iterations to processors, decreasing the chunk size to minimize the completion times between the first and last processors and to minimize the synchronization overhead. Guided Self Scheduling [28] assumes a fixed task execution length while Lucco [21] models task lengths stochastically to help guide the chunk size. Both approaches use a centralized task queue or data structure which limit their scalability. This paper has suggested improvements to distribute these structures throughout the machine in order to improve scalability. Neither algorithm allows dynamic task creation or task relocation.

The third case study presents a mechanism for dynamic load balancing of control parallelism, while minimizing task creation overhead by creating tasks lazily [23]. Lazy task creation can be scaled to large numbers of processors if a hierarchical method of searching for work is used. Lazy task creation does not execute parallel loops well. This paper suggests replicating a loop continuation rather than stealing it from a busy processor so that loop iterations can be distributed more easily throughout the machine. Each of the three case studies ignore the issue of locality in the assignment of tasks to processors, as well as in determining which tasks to steal from a heavily loaded processor.

This paper proposes using a task's affinity for a processor to determine which tasks to migrate during load balancing. The affinity of a task provides an indicator of the cost to execute the task on a remote node. Two tasks queues per processor, a *here* queue and a *there* queue, are used to separate queued tasks that have high affinity for the local processor from those that do not. Performance can be improved if work is stolen first from the *there* queue. An analytical model of task migration costs with affinity was presented. This model demonstrates that if task migration costs are low, then affinity is not useful. However, if the cost to migrate a task with affinity is near or larger than the task's execution time, then selecting cheap tasks becomes much more important.

In order for affinity scheduling to work well, accurate affinity assertions must be made. The affinity parameter is is a way for the compiler to communicate semantic information regarding a task to the runtime scheduling mechanisms. Reducing the overhead involved with scheduling is particularly important in fine-grained parallel programs. Ideally the programmer would be isolated from this additional complexity, but his input could be extremely valuable. Clearly these ideas must be expanded and worked out in detail before the utility of affinity scheduling can be judged fully.

# 6 Future Work

Affinity scheduling can be extended by using a continuum of affinities instead of just *here* or *there*. Task affinities might be updated when a task is migrated. Fiske [7] is studying how to schedule tasks based on adjustable task priority. Task affinity could be used as a scheduling criteria in such a system. Dynamic task creation and task affinity may provide another tool for the analysis and execution of `DOACROSS` loops [5], which have some form of loop carried dependence between iterations and are difficult to parallelize. The computation of a particular array element may trigger task creation for subsequent iterations. A task can be queued locally or sent to another processor, while the task's affinity may be calculated from the loop index and the mapping of the data.

Affinity scheduling may be incorporated into already existing scheduling and load balancing schemes. Two task queues can be used with lazy task creation for managing dynamic parallelism. Both queues are threaded through the processor's stack. When a future call is made and the parent has high affinity, then the call behaves as a normal lazy future. However, if the parent has low affinity, then the parent's activation can be linked into the *there* queue, where it becomes a more likely candidate for migration. When tasks are stolen, additional pointer manipulation is required to patch the activation frames back together, as tasks may be stolen out of the middle of the stack. The parent's affinity may be assigned by the programmer (via a `future-here` or `future-there`) or the compiler. The caveat to this, as well as all complex scheduling schemes, is that the benefits of more precise information must be weighed against the overhead in maintaining it.

## Acknowledgements

My appreciation goes to those people who have contributed directly to this work. Stuart Fiske provided some references on scheduling, as well as feedback on an early draft of this paper. Kathy Knobe provided expertise and suggestions regarding loop analysis. I also thank Ulana Legedza, Fred Chong, and Richard Lethin for their patience in reading drafts and for providing extremely useful suggestions of all sorts.

## References

[1] Agarwal, A., et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] Agarwal, A., Kranz, D., and Natarajan, V. Automatic partitioning of parallel loops for cache-coherent multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing* (1993), pp. 2–11.

[3] Coffman, E. *Computer and Job-Shop Scheduling Theory*. Wiley, New York, 1976.

[4] Culler, D. E. *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, Massachusetts Institute of Technology, June 1989.

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

[6] Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering SE-12*, 5 (May 1986), 662–675.

[7] Fiske, J. S. Thread scheduling mechanisms for multi-threaded parallel processors. PhD Thesis Proposal, Massachusetts Institute of Technology, July 1993.

[8] Gabriel, R. P., and McCarthy, J. Queue-based multi-processing lisp. In *1984 ACM Symposium on Lisp and Functional Programming* (August 1984), pp. 25–44.

[9] Goldman, R., Gabriel, R. P., and Sexton, C. Qlisp: An interim report. In *Proceedings of US/Japan Workshop on Parallel Lisp* (June 1989), pp. 161–181.

[10] Hsieh, W. C., Wang, P., and Weihl, W. E. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May 1993), pp. 239–248.

[11] Hummel, S. F., Schoenberg, E., and Flynn, L. E. Factoring: A method for scheduling parallel loops. *Communications of the ACM 35*, 8 (August 1992), 90–101.

[12] Hwang, J.-J., Chow, Y.-C., and Anger, F. D. An analysis of multiprocessing speedup with emphasis on the effect of scheduling methods. In *8th International Conference on Distributed Computing Systems* (June 1988), pp. 242–248.

[13] Kalé, L. V. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing* (1988), pp. 8–12.

[14] Kasahara, H., and Narita, S. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers c-33*, 11 (November 1984), 1023–1029.

[15] Kranz, D. A., January 1994. Personal Communication.

[16] Kranz, D. A., Halstead, R. H., and Mohr, E. Mul-T: A high-performance Lisp. In *Sigplan '89 Symposium on Programming Language Design and Implementation* (June 1989), pp. 1–10.

[17] Kruskal, C. P., and Weiss, A. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering SE-11*, 10 (October 1985), 1001–1016.

[18] Lam, M. S., Rothberg, E. E., and Wolf, M. E. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* (April 1991), pp. 63–74.

[19] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S. The Stanford Dash multiprocessor. *IEEE Computer 25*, 3 (March 1992), 63–79.

[20] Lin, F. C. H., and Keller, R. M. The gradient model load balancing method. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987), 32–38.

[21] Lucco, S. A dynamic scheduling method for irregular parallel programs. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (July 1992), pp. 200–211.

[22] Markatos, E. P., and LeBlanc, T. J. Load balancing vs. locality management in shared-memory multiprocessors. Tech. Rep. 399, Computer Science Department, University of Rochester, October 1991.

[23] Mohr, E., Kranz, D., and Halstead, R. H. Lazy task creation: A technique for increasing the granularity of parallel programs. Tech. Rep. 449, MIT Laboratory for Computer Science, June 1991.

[24] Ni, L. M., Xu, C.-W., and Gendreau, T. B. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering SE-11*, 10 (October 1985), 1153–1161.

[25] Nikhil, R., Papadopoulos, G., and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 156–167.

[26] Nussbaum, D. *Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors.* PhD thesis, Massachusetts Institute of Technology, September 1993.

[27] Pehoushek, J. D., and Weening, J. S. Low-cost process creation and dynamic partitioning in Qlisp. In *Proceedings of US/Japan Workshop on Parallel Lisp* (June 1989), pp. 182–199.

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

[29] Ponnusamy, R., Saltz, J., and Choudhary, A. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93* (November 1993), pp. 361–370.

[30] Saltz, J., Berryman, H., and Wu, J. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience 3(6)* (December 1991), 573–592.

[31] Sarkar, V., and Hennessy, J. Compile-time partitioning and scheduling of parallel programs. In *SIGPLAN '86 Symposium on Compiler Construction* (1986), pp. 17–26.

[32] Squillante, M. S. *Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation.* PhD thesis, University of Washington, October 1990.

[33] Tang, P., and Yew, P.-C. Processor self-scheduling for multiple-nested parallel loops. In *Proceedings of the 1986 International Conference on Parallel Processing* (August 1986), pp. 528–535.

[34] Tzen, T. H., and Ni, L. M. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems 4*, 1 (January 1993), 87–98.

[35] Vandevoorde, M. T., and Roberts, E. S. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming 17*, 4 (1988), 347–366.

[36] Znati, T. F., Melhem, R. G., and Pruhs, K. R. Dilation based bidding schemes for dynamic load balancing on distributed processing systems. In *Sixth Distributed Memory Conference Proceedings* (1991), pp. 129–136.