Space-Efficient Implementation of Nested Parallelism

Girija J. Narlikar
narlikar@cs.cmu.edu

Guy E. Blelloch
blelloch@cs.cmu.edu

CMU School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15232

Abstract

Many of today's high level parallel languages support dynamic, fine-grained parallelism. These languages allow the user to expose all the parallelism in the program, which is typically of a much higher degree than the number of processors. Hence an efficient scheduling algorithm is required to assign computations to processors at runtime. Besides having low overheads and good load balancing, it is important for the scheduling algorithm to minimize the space usage of the parallel program. This paper presents a scheduling algorithm that is provably space-efficient and time-efficient for nested parallel languages. In addition to proving the space and time bounds of the parallel schedule generated by the algorithm, we demonstrate that it is efficient in practice. We have implemented a runtime system that uses our algorithm to schedule parallel threads. The results of executing parallel programs on this system show that our scheduling algorithm significantly reduces memory usage compared to previous techniques, without compromising performance.

Keywords: Space efficiency, dynamic scheduling, nested parallelism, multithreading, language implementation.

1 Introduction

Many of today's high level parallel programming languages provide constructs to express dynamic, fine-grained parallelism. Such languages include data-parallel languages such as NESL [2] and HPF [21], as well as control-parallel languages such as ID [1], Cilk [5], CC++ [12], Sisal [17], Proteus [28], and C with light-weight thread libraries [25, 30]. These languages allow the user to expose all the parallelism in the program, which is typically of a much higher degree than the number of processors. The language implementation is responsible for scheduling this parallelism onto the processors. If the scheduling is done at runtime, then the performance of the high-level code relies heavily on the scheduling algorithm, which should have low scheduling overheads and good load balancing.

This paper appears in the Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), June 18-21, 1997, Las Vegas.

Several systems providing dynamic parallelism have been implemented with efficient runtime schedulers [7, 11, 13, 18, 19, 22, 23, 33, 34, 35], resulting in good parallel performance. However, in addition to good time performance, the memory requirements of the parallel computation must be taken into consideration. In an attempt to expose a sufficient degree of parallelism to keep all processors busy, schedulers often create many more parallel threads than necessary, leading to excessive memory usage [16, 32, 36]. Further, the order in which the threads are scheduled can greatly affect the total size of the live data at any instance during the parallel execution, and unless the threads are scheduled carefully, the parallel execution of a program may require much more memory than its serial execution. Since the price of the memory is a significant portion of the price of a parallel computer, and parallel computers are typically used to run big problem sizes, reducing memory usage is often as important as reducing running time. Many researchers have addressed this problem in the past. Early attempts to reduce the memory usage of parallel computations were based on heuristics that limited the parallelism [10, 16, 32, 36], and are not guaranteed to be space-efficient in general. These were followed by scheduling techniques that provide proven space bounds for parallel programs [6, 7, 8, 9]. If S_1 is the space required by the serial execution, these techniques generate schedules for a multithreaded computation on p processors that require no more than $p \cdot S_1$ space. These ideas are used in the implementation of the Cilk programming language [5].

A recent scheduling algorithm improved these space bounds from a multiplicative factor on the number of processors to an additive factor [3]. The algorithm generates a schedule that uses only $S_1 + O(p \cdot D)$ space, where D is the depth of the parallel computation (i.e., the longest sequence of dependencies or the critical path in the computation). This bound is asymptotically lower than the previous bound of $p \cdot S_1$ when $D < S_1$, which is true for parallel computations that have a sufficient degree of parallelism. For example, a simple algorithm to multiply two $n \times n$ matrices has depth $D = \Theta(\log n)$ and serial space $S_1 = \Theta(n^2)$, giving space bounds of $O(n^2 + p \log n)$ instead of $O(n^2 p)$ on previous systems[1]. The low space bound of $S_1 + O(p \cdot D)$ is achieved by ensuring that the parallel execution follows an order that is as close as possible to the serial execution. However, the algorithm has scheduling overheads that are too high for it to be practical. Since it is synchronous, threads need to be

[1] More recent work provides a stronger upper bound than $p \cdot S_1$ for space requirements of regular divide-and-conquer algorithms in Cilk [4].

rescheduled after every unit computation to guarantee the space bounds. Moreover, it ignores the issue of locality — a thread may be moved from processor to processor at every timestep.

This paper presents a variant on the scheduling algorithm proposed in [3] that overcomes the above mentioned problems. The paper then gives experimental results that demonstrate that the algorithm does achieve good performance both in terms of memory and time. The main goal in the design of the algorithm was to allow threads to execute nonpreemptively and asynchronously, allowing for better locality and lower scheduling overhead. This is achieved by allocating a pool of a constant K units of memory to each thread when it starts up, and allowing a thread to execute non-preemptively on the same processor until it runs out of memory from that pool or reaches a synchronization point. When a thread suspends, the processor accesses a new thread in a non-blocking manner from a work queue which is kept in the appropriate priority order (according to when it would have executed sequentially). The algorithm also delays large block allocations by effectively lowering their priority.

Although the nonpreemptive and asynchronous nature of our scheduling algorithm results in an execution order that differs from the order generated by the previous algorithm, we show that it retains the same space bound. In particular, it bounds the memory allocated at any time to $S_1 + 2KpD$ bytes, and bounds the space required by the scheduler itself to $(2pD + D + 5p + 1) \cdot c$, where c is the small, constant amount of memory required to store a thread[2]. Together these are within the $S_1 + O(p \cdot D)$ bounds. In addition to proving space-efficiency, as with [3] we bound the time required to execute the schedule for a parallel computation in terms of its work (total number of operations) and depth. For a parallel computation with D depth and W work, if the total space allocated is $O(W)$, then the generated schedule runs in $O(W/p + D)$ time, making it time-efficient [6]. We note that a bigger K leads to a lower running time since it reduces scheduling costs, but also results in a larger space bound. The K parameter therefore provides a trade-off between the running time and the memory requirement of a parallel computation. For the benchmarks used in our experiments, a value of $K = 1000$ bytes yields good performance in both space and time.

Our scheduling algorithm assumes a shared memory programming model and applies to languages providing nested parallelism. These include nested data-parallel languages, and control-parallel languages with a fork-join style of parallelism.

We have built a runtime system that uses our algorithm to schedule parallel threads on the SGI Power Challenge. To test its effectiveness in reducing memory usage, we have executed a number of parallel programs on it, and compared their space and time requirements with previous scheduling techniques. The experimental results show that, compared to previous techniques, our system significantly reduces the maximum amount of live data at any time during the execution of the programs. In addition, good single-processor performance and high parallel speedups indicate that the scheduling overheads in our system are low, that is, memory requirements can be effectively reduced without compromising performance.

[2]This is the memory required to store its state such as registers, not including the stack and heap data.

1.1 An example

The following pseudo-code illustrates the main ideas behind our scheduling algorithm, and how they result in lower memory usage compared to previous scheduling techniques.

```
In parallel for i = 1 to n
   Temporary B[n]
   In parallel for j = 1 to n
      F(B,i,j)
   Free B
```

This code has two levels of parallelism: the `i`-loop at the outer level and the `j`-loop at the inner level. In general, the number of iterations in each loop may not be known at compile time. Space for an array `B` is allocated at the start of each `i`-iteration, and is freed at the end of the iteration. Assuming that `F(B,i,j)` does not allocate any space, the serial execution requires $O(n)$ space, since the space for array `B` is reused for each `i`-iteration.

Now consider the parallel implementation of this function on p processors, where $p < n$. Previous scheduling systems [6, 8, 9, 14, 19, 23, 32, 36], which include both heuristic-based and provably space-efficient techniques, would schedule the outer level of parallelism first. This results in all the p processors executing one `i`-iteration each, and hence the total space allocated is $O(p \cdot n)$. Our scheduling algorithm also starts by scheduling the outer parallelism, but stalls big allocations of space. Moreover, it prioritizes operations by their serial execution order. As a result, the processors suspend the execution of their respective `i`-iterations before they allocate $O(n)$ space each, and execute `j`-iterations belonging to the first `i`-iteration instead. Thus, if each `i`-iteration has sufficient parallelism to keep the processors busy, our technique schedules iterations of a single `i`-loop at a time. This allows the parallel computation to run in just $O(n + p \cdot D)$ space, where D is the depth of the function `F`.

As a related example, consider n users of a parallel machine, each running parallel code. Each user program allocates a large block of space as it starts and deallocates the block when it finishes. In this case the outer parallelism is across the users and the inner parallelism is within each user's program. A scheduler that schedules the outer parallelism would schedule p user programs to run simultaneously, requiring a total memory equal to the sum over the memory requirements of p programs. On the other hand, our scheduling algorithm would schedule one program at a time (in their serial execution order), as long as there is sufficient parallelism within each program to keep the processors busy. In this case, the total memory required is just the maximum over the memory requirement of each user's program.

A potential problem with our algorithm is that since it often preferentially schedules inner parallelism (which is finer grained), it can cause larger scheduling overheads compared to algorithms that schedule outer parallelism. We overcome this problem by grouping the fine-grained iterations of innermost loops into chunks. Our experimental results demonstrate that this approach is sufficient to yield good performance in time and space (see Section 6). In the results reported in this paper we have blocked the iterations into chunks by hand, but in Section 7 we discuss some ongoing work on automatically chunking the iterations. In general there is a tradeoff between memory use and scheduling efficiency.

1.2 Outline of the paper

We begin by defining the multithreaded programming model that our scheduling algorithm implements in Section 2. Section 3 describes how any parallel computation in this model can be represented as a DAG—this representation is used in the proofs throughout the paper. Section 4 presents our online scheduling algorithm; the space and time bounds for schedules generated by it are stated in Section 5. The implementation of our runtime system and the results of executing parallel programs on it are described in Section 6. Finally, we summarize and describe future work in Section 7.

2 Model of parallelism

Our scheduling algorithm is applicable to languages that support nested parallelism, which include data-parallel languages (with nested parallel loops and nested parallel function calls), control-parallel languages (with fork-join constructs), and any mix of the two. The algorithm assumes a shared memory programming model, in which parallel programs can be described in terms of threads. Threads may be either *active* (executing or ready to execute) or *suspended*. The computation starts with one initial thread. On encountering a parallel loop (or fork), a thread forks one child thread for each iteration and suspends itself.[3] Each child thread may, in turn, fork more threads. We assume that the child threads do not communicate with each other. The last child thread to terminate reactivates the suspended parent thread. We call the last instruction of a thread its *synchronization point*. A thread may fork any number of child threads, and this number need not be known at compile time. We assume the program is deterministic and does not include speculative computation.

A thread is said to be *scheduled* when a processor begins executing its computations; a thread that suspends must be reactivated (made ready) before it is rescheduled. The *work* in a parallel computation is the total number of unit operations executed in it (that is, the time taken to execute it serially), and the *depth* is the length of the critical path (the time required to execute the computation on an infinite number of processors). The space requirement of an execution is the maximum of the total memory allocated across all processors at any time during the execution.

To maintain our space bounds, we impose an additional restriction on the threads. Once scheduled, a new or reactivated thread may perform a memory allocation from a global pool of memory only in its first step. The memory allocated becomes its private pool of memory, and may be subsequently used for a variety of purposes, such as, for dynamically allocated data or activation records. When the thread runs out of its private pool and reaches a computational step that needs to allocate more memory, the thread must suspend itself, to be rescheduled at a later time. The reason to suspend threads just before they allocate more space is to allow threads which have a higher priority (an earlier order in the serial execution) to get scheduled instead. The thread scheduling policy is transparent to the programmer.

[3] As discussed later, the implementation actually forks the threads lazily so that the space for a thread is only required when it is started.

3 Representing the computation as a DAG

To formally define the space required by a parallel computation and help us prove the time and space bounds we view the computation as a precedence graph, that is, a directed acyclic graph (DAG) in which each node corresponds to a unit of sequential computation (work) that takes one timestep to be executed. The edges of the DAG express the dependencies between the computations. We will denote the amount of memory allocated from the global pool in the computational step of a node v by $m(v)$. If the step performs a deallocation, $m(v)$ is negative[4].

For the nested parallelism model described in Section 2, the dynamically unfolding DAG has a *series-parallel* structure. A series-parallel DAG [3] can be defined inductively: the DAG G_0 consisting of a single node (which is both its source and sink) and no edges, is series-parallel. If $G_1, \ldots, G_n$, $n \geq 1$ are series-parallel, then the DAG obtained by adding to $G_1 \cup \ldots \cup G_n$ a new source node u, with edges from u to the source nodes of $G_1, \ldots, G_n$, and a new sink node v, with edges from the sink nodes of $G_1, \ldots, G_n$ to v is series-parallel.

Since the program is deterministic and does not include speculative computation, the DAG is independent of the implementation. The total number of nodes in the DAG corresponds to the total work of the computation, and the longest path in the DAG corresponds to the depth. A thread that performs w work is represented as a sequence of w nodes in the DAG. When a thread forks child threads, edges from its current node to the initial nodes of the child threads are revealed. Similar dependency edges are revealed at the synchronization point. Since we do not restrict the number of threads that can be forked, a node may have an arbitrary in-degree and out-degree. For example, Figure 1 shows a small portion of a DAG. Only the first node of a new or reactivated thread may allocate space; a thread must suspend itself on reaching a subsequent node that performs an allocation. The points where threads are suspended and subsequently reactivated and rescheduled are marked as dashed lines.

Definitions. In a DAG $G = (V, E)$, for every edge $(u, v) \in E$, we call u a *parent* of v, and v a *child* of u. A *serial schedule* of the DAG is a topologically sorted sequence $s_1 = v_1, v_2, \ldots, v_W$ of its nodes in which each node appears exactly once, and a node can appear only after all its parent nodes have appeared, since the edges are dependency edges. A serial schedule for a DAG G represents a serial execution of the computation represented by G. Thus, a node v appears in the serial schedule at position i if the computation of v is executed at timestep i. Similarly, the parallel execution of the computation on p processors can be represented by a *p-schedule* $s_p = V_1, V_2, \ldots, V_\tau$, where V_i is the set of nodes executed at timestep i. Since each processor can execute at most one node in each timestep, each set V_i contains at most p nodes. As in the serial case, this schedule must obey the dependency edges, that is, a node may appear in a set V_i only if all its parent nodes appear in previous sets. We say a node v is *ready* at timestep i, if all the parents of v have been executed, but v has not been executed.

For a serial execution represented by the serial schedule $v_1, v_2, \ldots, v_W$, the total space requirement is defined as $S_1 = n + max_{j=1,\ldots,W} \left(\sum_{i=1}^{j} m(v_i) \right)$, where n is the

[4] We assume that a single computational step does not perform both an allocation and a deallocation.

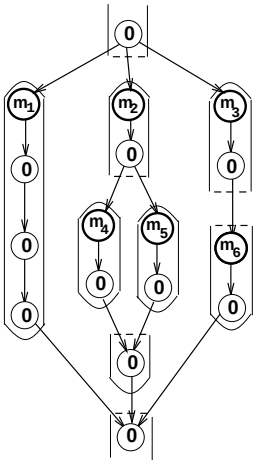

Figure 1: A portion of a program DAG and the threads that perform its computation. Each node corresponds to a unit computational step, and is labeled with the amount of memory allocated in that step. The edges express dependencies between the computational steps. The threads are shown as outlines around the nodes. Dashed lines mark points in a thread where it gets suspended and subsequently reactivated and rescheduled. Only the initial node of each new or reactivated thread, which is shown in bold, may have a positive label, that is, may allocate space from the global pool to be used as the local pool.

space required to store the input. Similarly, the space required by a parallel execution represented as $V_1, V_2, \ldots, V_\tau$ is $S_p = n + max_{j=1,\ldots,\tau} \left(\sum_{i=1}^{j} Space(V_i)\right)$, where $Space(V_i) = \sum_{v \in V_i} m(v)$.

1DF-schedule: In most languages, a serial execution can be represented by a *depth-first* schedule (1DF-schedule) of the computation DAG. The first step of a 1DF-schedule executes the root node; at every subsequent step, the leftmost ready child of the most recently executed node with a ready child is executed. The order in which the nodes are executed determines their 1DF-*numbers*. For example, Figure 2 shows the 1DF-numbering of a simple program DAG.

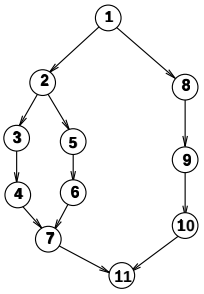

Figure 2: 1DF-schedule of a program DAG: each node is labeled with the order in which it is executed. Note that a node cannot be executed until all the nodes that have edges into it have been executed. Most systems execute nodes of a serial computation in this order.

4 The scheduling algorithm

The key idea behind our scheduling algorithm is to schedule threads in an order that is as close to a 1DF ordering (serial ordering) of their nodes as possible. Since we must schedule multiple threads simultaneously to keep all the processors busy, threads get scheduled earlier than they would be in the serial execution. These are the threads that cause the parallel execution to require more memory; however, we can limit the number of such threads by following an order of execution dictated by the 1DF-numbers.

We first describe the data structures used in the system, followed by a description of the algorithm. We assume for now that every time a thread is scheduled, the number of bytes it allocates from the global pool is at most a constant K, and that it suspends when it reaches a computational step requiring more memory. This assumption is valid if no single computational step in a thread needs to allocate more than K memory; we will explain how to handle allocations larger than K later in this section.

Runtime Data Structures. The central data structure is the *ready-queue*, which is a priority queue containing ready threads (threads that are ready to be executed), suspended threads, and stubs that act as place-holders for threads that are currently being executed. Threads in the ready-queue are stored from, say, left to right, in increasing order of the 1DF-numbers of their *leading* nodes (the first nodes to be executed when the thread is next scheduled). The lower the 1DF-number of the leading node of a thread, the higher is the thread's priority. Maintaining threads in this order requires the ready-queue to support operations such as inserting or deleting from the middle of the queue. Implementing fast, concurrent operations on such a queue is difficult; instead, we introduce two additional queues called the *in-queue* and the *out-queue*. These queues are simple FIFOs that support efficient concurrent inserts and deletes. They act as input and output buffers to store threads that are to be inserted or that have been removed, respectively, from the ready-queue (see Figure 3). Thus processors can perform fast, non-blocking accesses to these queues. In this paper we use a serial algorithm to move threads between the buffers and the ready-queue. Elsewhere we discuss how this can be done in parallel [29].

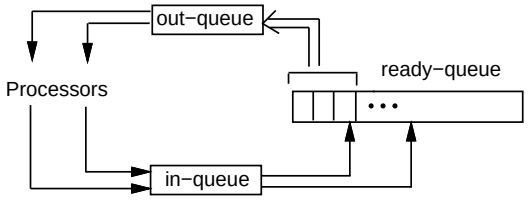

Figure 3: The movement of threads between the processors and the scheduling queues. The in-queue and out-queue are FIFOs, whereas the ready-queue allows insertions and deletions of intermediate nodes. Threads in the ready-queue are always stored in increasing order of the 1DF-numbers of their leading nodes (the first nodes to be executed when the thread gets scheduled). The in- and out-queues contain at most $3p$ threads combined.

Algorithm description. The pseudocode for the scheduling algorithm is given in Figure 4. The processors normally act as *workers*, when they take threads off the out-queue,

```
begin worker
  while (there exist threads in the system)
    τ := remove-thread(out-queue);
    if (τ is a scheduling thread) then scheduler()
    else
      execute the computation associated with τ;
      if (τ terminates) or (τ suspends)
      then insert-thread(τ, in-queue);
end worker

begin scheduler
  acquire scheduler-lock;
  insert a scheduling thread into the out-queue;
  T := remove-all-threads(in-queue);
  for each thread τ in T
    insert τ into the ready-queue in its original position;
    if τ has terminated
      if τ is the last among its siblings to synchronize,
        reactivate τ's parent;
      delete τ from the ready-queue;
  select the leftmost p ready threads from the ready-queue:
    if there are less than p ready threads, select them all;
    fork child threads in place if needed;
  insert these selected threads into the out-queue;
  release scheduler-lock;
end scheduler
```

Figure 4: The asynchronous scheduling algorithm. When the scheduler forks child threads, it inserts them into the ready-queue in the positions of their parents. This maintains the invariant that the threads in the ready queue are always in the order of increasing 1DF-numbers of their leading nodes. Therefore, at every scheduling step, the p ready threads whose leading nodes have the smallest 1DF-numbers, are moved to the out-queue. Child threads are forked only when they are to be added to the out-queue, that is, when they are among the leftmost p ready threads in the ready-queue.

execute them until they suspend or terminate, and then return them to the in-queue. A new or reactivated thread that is picked off the out-queue may allocate space from a global pool in its first computational step; it must suspend itself before any subsequent step that requires more space. A thread that performs a fork must also suspend itself; it is reawakened when the last of its forked child threads terminates. A child thread terminates upon reaching the synchronization point.

In addition to acting as workers, the processors take turns in acting as the *scheduler*. For this purpose, we introduce special *scheduling threads* into the system. Whenever the thread taken off the out-queue by a processor turns out to be a scheduling thread, it assumes the role of the scheduling processor and executes one scheduling step. Only one processor can be executing a scheduling step at a time. The algorithm begins with a scheduling thread and the first (root) thread of the program on the out-queue.

A processor that executes a scheduling step starts by putting a new scheduling thread on the out-queue. Next, it moves threads from the in-queue to the ready-queue. Each thread has a pointer to a stub that marks its original position relative to the other threads in the ready-queue; it is inserted back in that position. The scheduler then compacts the ready queue by removing threads that have terminated. These are child threads that have reached their synchronization point and the root thread at the end of the entire computation. If a thread is the last among its siblings to reach its synchronization point, its suspended parent thread is reactivated. All threads that were suspended due to a memory allocation are immediately reactivated on being returned to the ready-queue. If a thread performs a fork, its child threads are inserted immediately before it, and the forking thread suspends. The child threads are placed in the ready-queue in order of the 1DF-numbers of their leading nodes. Finally, the scheduler moves the *leftmost* p ready threads from the ready-queue to the out-queue, leaving behind stubs to mark their positions in the ready-queue. If the ready-queue has less than p ready threads, the scheduler moves them all to the out-queue. The scheduling thread then completes, and the processor resumes the task of a worker.

This scheduling algorithm ensures that the total number of threads in the in-queue and out-queue is at most $3p$ (see Lemma 7). Further, to limit the number of threads in the ready-queue, we *lazily* create the child threads of a forking thread: a child thread is not explicitly created until it is to be moved to the out-queue, that is, when it is among the leftmost p threads represented in the ready-queue. Until then, the parent thread implicitly represents the child thread. A single parent may represent several child threads. This optimization ensures that a thread does not have an entry in the ready-queue until it has been scheduled at least once before, or is in (or about to be inserted into) the out-queue. Note that if a thread T_i is ready to fork child threads, all its child threads will be forked (created) and scheduled before any other threads in the ready-queue to the right of T_i can be scheduled.

Handling large allocations of space. We had assumed earlier in this section that every time a thread is scheduled, it allocates at most K bytes for its use from a global pool of memory. This does not allow a unit computational step within a thread to allocate more than K bytes. We now show how such allocations are handled, similar to the technique suggested in [3]. The key idea is to delay the big allocations, so that if threads with lower 1DF numbers become ready, they will be executed instead. Consider a thread with a node that allocates m units of space in the original DAG, and $m > K$. We transform the DAG by inserting a fork of m/K parallel threads before the memory allocation (see Figure 5). These new child threads perform a unit of work (a no-op), but do not allocate any space; they simply consist of dummy nodes. However, we treat the dummy nodes as if they allocate space, and the original thread is suspended at the fork. It is reactivated when all the dummy threads have been executed, and may now proceed with the allocation of m space. This transformation of the DAG increases its depth by at most a constant factor. If S_a is the total space allocated in the program (not counting the deallocations), the number of nodes in the transformed DAG is at most $W + S_a/K$. This transformation takes place at runtime, and our scheduling algorithm generates a schedule for this transformed DAG. This ensures that the space requirement of the generated schedule does not exceed our space bounds, as proved in Appendix A.

We state the following lemma about the order of the nodes in the ready-queue maintained by our algorithm.

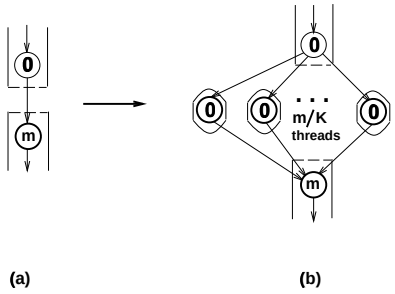

Figure 5: A transformation of the DAG to handle a large allocation of space at a node without violating the space bound. Each node is labeled with the amount of memory its computational step allocates. When a thread needs to allocate m space ($m > K$), we insert m/K parallel child threads before the allocation. Each child thread consists of a dummy node that does not allocate any space. After these child threads complete execution, the original thread performs the allocation, and continues with its execution.

Lemma 1 *The asynchronous scheduling algorithm in Figure 4 always maintains the threads in the ready-queue in an increasing order of the* 1DF*-numbers of their leading nodes.*

Proof: This can be proved by induction. The proof is similar to the one presented in [3], and we present the outline here. When the execution begins, the ready-queue contains just the root thread, and therefore it is ordered by the 1DF-numbers. Assume that at the start of some scheduling step, the threads in the ready-queue are in increasing order of the 1DF-numbers of their leading nodes. For a thread that forks, inserting its child threads before it in the order of their 1DF-numbers maintains the ordering by 1DF-numbers. A thread that suspends due to a memory allocation is returned to its original position. Its new leading node has the same 1DF-number as its previous leading node, *relative* to leading nodes of other threads. Deleting threads from the ready-queue does not affect their ordering. Therefore the ordering of threads in the ready-queue by 1DF-numbers is preserved after every operation performed by the scheduler. ∎

Lemma 1 implies that when the scheduler moves the leftmost p threads from the ready-queue to the out-queue, their leading nodes are the nodes with the lowest 1DF-numbers. We will use this fact to prove the space bounds of the schedule generated by our scheduling algorithm.

5 Theoretical results

5.1 Space bound

We state the following theorem on the space required to execute the schedule generated by our scheduling algorithm. The bound includes the space required for the three scheduling queues. The proof is presented in Appendix A.

Theorem 2 *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by our scheduling algorithm using $S_1 + O(p \cdot D)$ space (including scheduler space).* ∎

5.2 Time bound

The proof for the following theorem on the time bound is also presented in Appendix A.

Theorem 3 *Consider a computation of depth D and work W that allocates a total of S_a space. Our scheduling algorithm executes this computation on p processors in $O(W/p + S_a/p + D)$ timesteps.* ∎

Our time bounds do not include the cost of the scheduler. We have described a serial scheduler in this paper; however, the scheduler can be parallelized as described in [29], and its time and space costs can be included to execute a schedule in $O(W/p + S_a/p + D \log p)$ time and $S_1 + O(pD \log p)$ space. Such a parallel implementation is important when the number of processors is large, to ensure the scalability of the scheduler. In this paper, we restrict our discussion to the serial scheduler, and concentrate on its implementation and experimental results on a moderate number of processors.

6 Experimental results

We have built a runtime system that uses our algorithm to schedule parallel threads, and have run several experiments to analyze both the time and the memory required by parallel computations. In this section we briefly describe the implementation of the system and the benchmarks used to evaluate its performance, followed by the experimental results.

6.1 Implementation

The runtime system has been implemented on a 16-processor SGI Power Challenge, which has a shared memory architecture with processors and memory connected via a fast shared-bus interconnect. Since the number of processors on this architecture is not very large, a serial implementation of the scheduler as described in Section 4 does not create a bottleneck in our runtime system. The ready-queue is implemented as a simple, singly-linked list. The in-queue and out-queue, which are accessed by the scheduler and the workers, are required to support concurrent enqueue and dequeue operations. They are implemented using variants of lock-free algorithms based on atomic fetch-and-Φ primitives [27].

The parallel programs executed using this system have been explicitly hand-coded in the continuation-passing style, similar to the code generated by the Cilk preprocessor[5] [5]. Each continuation points to a C function representing the next computation of a thread, and a structure containing all its arguments. These continuations are created dynamically and moved between the queues. A worker processor takes a continuation off the out-queue, and simply applies the function pointed to by the continuation, to its arguments. The high-level program is broken into such functions at points where it executes a parallel fork, a recursive call, or a memory allocation.

For nested parallel loops, we group iterations of the innermost loop into equal-sized chunks, provided it does not contain calls to any recursive functions[6]. Scheduling a chunk

[5] We expect a preprocessor-generated version on our system to have similar efficiency as the straightforward hand-coded version.

[6] It should be possible to automate such coarsening with compiler support.

at a time improves performance by reducing scheduling overheads and providing good locality, especially for fine-grained iterations.

6.2 Benchmark programs

We implemented five parallel programs on our runtime system. We briefly describe the implementation of these programs, along with the problem sizes we used in our experiments.

• *Blocked recursive matrix multiply (Rec MM).* This program multiplies two dense $n \times n$ matrices using a simple recursive divide-and-conquer method, and performs $O(n^3)$ work. The recursion stops when the blocks are down to the size of 64×64, after which the standard row-column matrix multiply is executed serially. This algorithm significantly outperforms the row-column matrix multiply for large matrices (e.g., by a factor of over 4 for 1024×1024 matrices) because its use of blocks results in better cache locality. At each step, the eight recursive calls are made in parallel. Each recursive call needs to allocate temporary storage, which is deallocated before returning from the call. The results reported are for the multiplication of two 1024×1024 matrices of double-precision floats.

• *Strassen's matrix multiply (Str MM).* The DAG for this algorithm is very similar to that of the blocked recursive matrix multiply, but performs only $O(n^{2.807})$ work and makes seven recursive calls at each step [37]. Once again, a simple serial matrix multiply is used at the leaves of the recursion tree. The sizes of matrices multiplied were the same as for the previous program.

• *Fast multipole method (FMM).* This is an n-body algorithm that calculates the forces between n bodies in $O(n)$ work [20]. We have implemented the most time-consuming phases of the algorithm, which are a bottom-up traversal of the octree followed by a top-down traversal. In the top-down traversal, for each level of the octree, the forces on the cells in that level due to their neighboring cells are calculated in parallel. For each cell, the forces over all its neighbors are also calculated in parallel, for which temporary storage needs to be allocated. This storage is freed when the forces over the neighbors have been added to get the resulting force on that cell. With two levels of parallelism, the structure of this code looks very similar to the pseudocode described in Section 1. We executed the FMM on a uniform octree with 4 levels (8^3 leaves), using 5 multipole terms for force calculation.

• *Sparse matrix-vector multiplication (Sparse MV).* This multiplies an $m \times n$ sparse matrix with a $n \times 1$ dense vector. The dot product of each row of the matrix with the vector is calculated to get the corresponding element of the resulting vector. There are two levels of parallelism: over each row of the matrix, and over the elements of each row multiplied with the corresponding elements of the vector to calculate the dot product. For our experiments, we used $m = 20$ and $n = 1500000$, and 30% of the elements were non-zeroes. (Using a large value of n provides sufficient parallelism within a row, but using large values of m leads to a very large size of the input matrix, making the amount of dynamic memory allocated in the program negligible in comparison.)

• *ID3.* This algorithm by Quinlan [31] builds a decision tree from a set of training examples in a top-down manner, using a recursive divide-and-conquer strategy. At the root node, the attribute that best classifies the training data is picked, and recursive calls are made to build subtrees, with each subtree using only the training examples with a particular value of that attribute. Each recursive call is made in parallel, and the computation of picking the best attribute at a node, which involves counting the number of examples in each class for different values for each attribute, is also parallelized. Temporary space is allocated to store the subset of training examples used to build each subtree, and is freed once the subtree is built. We built a tree from 4 million test examples, each with 4 multi-valued attributes.

6.3 Time performance

Figure 6 shows the speedups for the above programs for up to 16 processors. The speedup for each program is with respect to its efficient serial C version, which does not use our runtime system. Since the serial C program runs faster than our runtime system on a single processor, the speedup shown for one processor is less than 1. However, for all the programs, it is close to 1, implying that the overheads in our system are low. The timings on our system include the delay introduced before large allocations, in the form of m/K dummy nodes for an allocation of m bytes. We have used a value of $K = 1000$ bytes in our experiments. Figure 6 also shows the speedups for the same programs running on an existing space-efficient system Cilk. To make a fair comparison, we have chunked innermost iterations of the Cilk programs in the same manner as we did for our programs. The timings show that the performance on our system is comparable with that on Cilk. The memory-intensive programs such as sparse matrix-vector multiply do not scale well on either system beyond 12 processors; their performance is probably affected by bus contention as the number of processors increases.

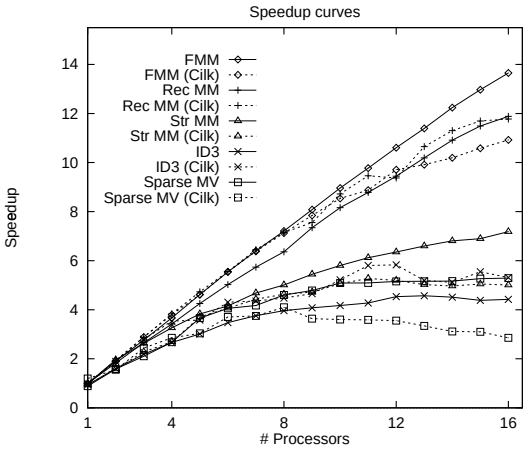

Figure 6: The speedups achieved on up to 16 R10000 processors of a Power Challenge machine, using a value of K=1000 bytes. The speedup on p processors is the time taken for the serial C version of the program divided by the time for our runtime system to run it on p processors. For each application, the solid line represents the speedup using our system, while the dashed line represents the speedup using the Cilk system. All programs were compiled using `gcc -O2 -mips2`.

Figure 7 shows the breakdown of the running time for

one of the programs, blocked recursive matrix multiplication. The results show that the percentage of time spent waiting for threads to appear in the out-queue increases as the number of processors increases (since we use a serial scheduler). A parallel implementation of the scheduler, such as one described in [29], will be more efficient for a larger number of processors.

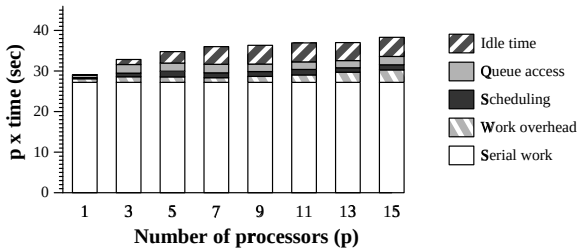

Figure 7: The total processor time (the running time multiplied by the number of processors p) for blocked recursive matrix multiplication. "Serial work" is the time taken by a single processor executing the equivalent serial C program. For ideal speedups, all the other components would be zero. The other components are overheads of the parallel execution and the runtime system. "Idle time" is the total time spent waiting for threads to appear in the out-queue; "queue access" is the total time spent by the worker processors inserting threads into the in-queue and removing them from the out-queue. "Scheduling" is the total time spent as the scheduler, and "work overhead" includes overheads of creating continuations, building structures to hold arguments, executing dummy nodes, and (de)allocating memory from a shared pool of memory, as well as the effects of cache misses and bus contention.

6.4 Space performance

Figure 8 shows the memory usage for each application. Here we compare three implementations for each program—one in Cilk and the other two using our scheduling system. Of the two implementations on our system, one uses dummy nodes to delay large allocations, while the other does not. For the programs we have implemented, the version without the delay results in approximately the same space requirements as would result from scheduling the outermost level of parallelism. For example, in Strassen's matrix multiplication, our algorithm without the delay would allocate temporary space required for p branches at the top level of the recursion tree before reverting to the execution of the subtree under the first branch. On the other hand, scheduling the outer parallelism would allocate space for the p branches at the top level, with each processor executing a subtree serially. Hence we use our algorithm without the delay to estimate the memory requirements of previous techniques like [14, 23], which schedule the outer parallelism with higher priority. Cilk uses less memory than this estimate due to its use of randomization: an idle processor steals the topmost thread (representing the outermost parallelism) from the private queue of a randomly picked processor; this thread may not represent the outermost parallelism in the entire computation. Previous techniques like [8, 9, 19] use a strategy similar to that of Cilk. The results show that when big allocations are delayed with dummy nodes, our algorithm results in a significantly lower memory usage, particularly as the number of processors increases. We have not compared it to naive scheduling techniques, such as breadth-first schedules, which have much higher memory requirements.

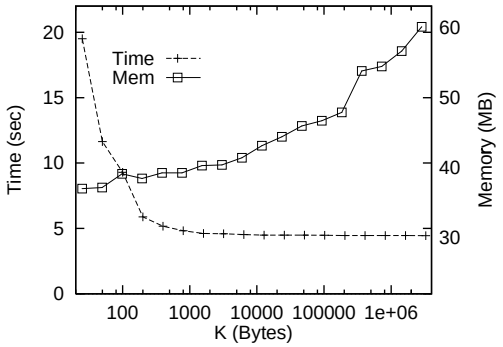

Figure 9: The variation of running time and memory usage with K (in bytes) for multiplying two 1024×1024 matrices using blocked recursive matrix multiplication on 8 processors. For very small K, many dummy nodes are inserted, which results in a high running time. For very large K, very few dummy nodes are inserted, causing only a small delay before a big memory allocation, resulting in high memory usage. K=500-2000 bytes results in both good performance and low memory usage.

As mentioned above, the number of dummy nodes introduced depends on the number of bytes K that a thread is allowed to allocate before being suspended. Hence there is a trade-off between memory usage and running time. For example, Figure 9 shows how the running time and memory usage for blocked recursive matrix multiplication are affected by K. For all the programs we implemented, the trade-off curves looked similar; however, they may vary for other parallel programs. A default value of $K = 1000$ bytes resulted in a good balance between space and time performance for all our programs, although in practice it might be useful to allow users to tune the parameter for their needs.

7 Summary and discussion

We have presented an asynchronous scheduling algorithm for languages that support nested parallelism, and have shown that it is space-efficient and time-efficient in both theory and practice. The space bound presented in this paper is significantly lower than space bounds on existing systems for programs with a sufficient amount of parallelism ($D \gg S_1$). Most parallel programs, including all programs in NC [15], fall in this category. We have built a low-overhead runtime system that schedules parallel threads using our algorithm. The results demonstrate that our approach is more effective in reducing space usage than previous scheduling techniques, and at the same time yields good parallel performance. A more detailed specification of the theoretical framework on which the scheduling algorithm is based, along with a description of an efficient parallelized scheduler, can be found in [29].

We are currently considering methods to further improve the scheduling algorithm, particularly to provide better support for fine-grained computations. At present, fine-grained iterations of innermost loops are statically grouped

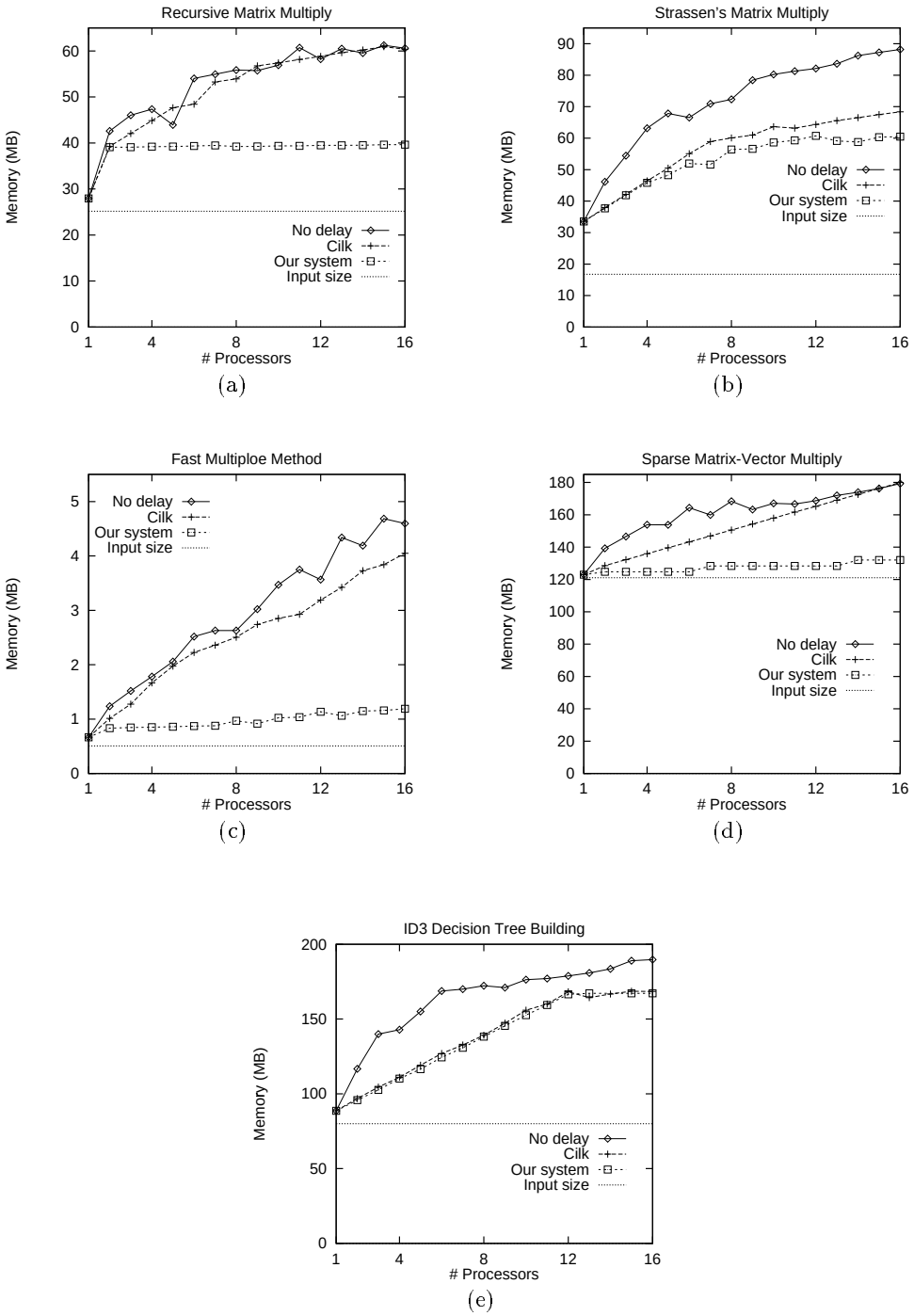

Figure 8: The memory requirements of the parallel programs. For $p = 1$ the memory usage shown is for the serial C version. We compare the memory usage of each program when the big memory allocations are delayed by inserting dummy threads (using $K = 1000$), with when they are allowed to proceed without any delay, as well as with the memory usage on Cilk. The version without the delay on our system (labeled "No delay") is an estimate of the memory usage resulting from previous scheduling techniques. These results show that delaying big allocations significantly changes the order of execution of the threads, and results in much lower memory usage, especially as the number of processors increases.

into fixed-size chunks. A dynamic, decreasing-size chunking scheme such as [24, 26, 38] can be used instead. We are considering ways of automatically introducing such coarsening at runtime through the scheduling algorithm; for example, by allowing the execution order to differ to a limited extent from the order dictated by the 1DF-numbers. We also plan to reduce contention for the shared queue by introducing multiple queues; the processors will access separate queues at any given time. Finally, we plan to implement faster parallel calls to efficiently execute fine-grained computations.

Acknowledgements

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant DABT63-96-C-0071. Access to the SGI Power Challenge was provided by the the National Center for Supercomputing Applications (NCSA), whose staff was extremely helpful when dealing with all our requests

References

[1] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[3] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. Symposium on Parallel Algorithms and Architectures*, Santa Barbara, July 1995.

[4] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proc. Symp. on Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. Symposium on Principles and Practice of Parallel Programming*, pages 207–216, November 1995.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 362–371, May 1993.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pages 356–368, November 1994.

[8] F. W. Burton. Storage management in virtual tree machines. *IEEE Trans. on Computers*, 37(3):321–328, 1988.

[9] F. W. Burton and D. J. Simpson. Space efficient execution of deterministic parallel programs. Manuscript, December 1994.

[10] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Conference on Functional Programming Languages and Computer Architecture*, October 1981.

[11] R. Chandra, A. Gupta, and J. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[12] K. M. Chandy and C. Kesselman. Compositional c++: compositional parallel programming. In *Proc. 5th. Intl. Wkshp. on Languages and Compilers for Parallel Computing*, pages 124–144, New Haven, CT, August 1992.

[13] J. S. Chase, F. G. Amador, and E. D. Lazowska. The amber system: Parallel programming on a network of multiprocessors. In *Proc. Symposium on Operating Systems Principles*, December 1989.

[14] J. H. Chow and W. L. Harrison III. Switch-stacks: A scheme for microtasking nested parallel loops. In *Proc. Supercomputing*, New York, NY, November 1990.

[15] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[16] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proc. Intl. Symposium on Computer Architecture*, May 1988.

[17] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[18] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, Monterey, CA, November 1994.

[19] S. C. Goldstein, D. E. Culler, and K. E. Schauser. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Rochester, NY, May 1995.

[20] L. Greengard. *The rapid evaluation of potential fields in particle systems.* The MIT Press, 1987.

[21] High Performance Fortran Forum. *High Performance Fortran language specification*, May 1993.

[22] W. E. Hseih, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed memory parallel systems. In *Proc. Symposium on Principles and Practice of Parallel Programming*, San Francisco, California, May 1993.

[23] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelsim. *IBM Journal of Research and Development*, 35(5-6):743–65, 1991.

[24] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug 1992.

[25] IEEE. Threads extension for portable operating systems (draft 6), Feb 1985.

[26] C. D. Polychronopoulos; D.J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–39, Dec 1987.

[27] J. M. Mellor-Crummey. Concurrent queues: Practical Fetch-and-Φ algorithms. Technical Report 229, University of Rochester, November 1987.

[28] P. H. Mills, L. S. Nyland, J. F. Prins, J. H. Reif, and R. A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

[29] G. J. Narlikar and G. E. Blelloch. A framework for space and time efficient scheduling of parallelism. Technical Report CMU-CS-96-197, Computer Science Department, Carnegie Mellon University, 1996.

[30] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proc. USENIX COnference*, 1991.

[31] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[32] Jr. R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[33] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, June 1993.

[34] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[35] R.S.Nikhil. Cid: A parallel, shared-memory c for distributed memory machines. In *Proc. 7th. Ann. Wkshp. on Languages and Compilers for Parallel Computing*, pages 376–390, August 1994.

[36] C. A. Rugguero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In *Functional Programming Languages and Computer Architecture*, volume 174 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1987.

[37] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

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

A Proofs for space and time bounds

A.1 Space bound

In order to prove that the parallel execution requires $S_p = S_1 + O(p \cdot D)$ space, we will view each thread as a series of nodes in the program DAG, as described in Section 2. Every node in the graph represents a unit of work, which we assume takes one timestep to be executed. We will call the node representing the first computation of a new or reactivated thread its *heavy* node, and the other nodes *light* nodes. Thus, heavy nodes may allocate space, while light nodes allocate no space (but may deallocate space). When a thread is moved from the ready-queue to the out-queue by a scheduling processor, we will consider its leading heavy node to be *scheduled*. This is because once threads are put on the out-queue, they are taken out and executed by processors in the FIFO order. Note that a heavy node may get executed several timesteps after it becomes ready and after it is scheduled. However, a light node is executed in the timestep it becomes ready, since a processor executes consecutive light nodes nonpreemptively.

Let $s_p = V_1, \ldots, V_\tau$ be the parallel schedule of the DAG generated by our scheduling algorithm. Here V_i is the set of nodes that are executed at timestep i. Let s_1 be the 1DF-schedule for the same DAG. Consider an arbitrary prefix, σ_p, of s_p. Let σ_1 be the longest prefix of s_1 containing only nodes in σ_p, that is, σ_1 does not contain any nodes that are not part of σ_p. We call the nodes that are in σ_p but not in σ_1 the *premature* nodes, since they have been executed out of order. For example, Figure 10 shows a simple DAG with a parallel prefix σ_p and its corresponding serial prefix σ_1.

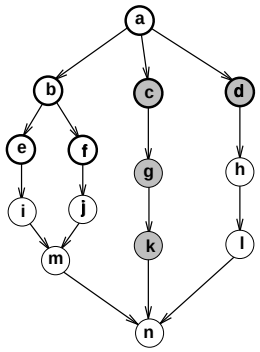

Figure 10: A simple program DAG in which the heavy nodes (a, b, c, d, e, and f) are shown as bold. The 1DF-schedule for this DAG is $s_1 = [a, b, e, i, f, j, m, c, g, k, d, h, l, n]$. For $p = 2$, a possible parallel schedule is $s_p = [\{a\}, \{b, c\}, \{e, g\}, \{i, k\}, \{f, d\}, \{j, h\}, \{m, l\}, \{n\}]$. Once a heavy node is executed, the processor continues to execute the subsequent light nodes of the thread. For a prefix $\sigma_p = [\{a\}, \{b, c\}, \{e, g\}, \{i, k\}, \{f, d\}]$ of s_p, the corresponding prefix of s_1 is $\sigma_1 = [a, b, e, i, f]$. Thus, the premature nodes in σ_p are c, d, g, and k (shown shaded). Of these, the heavy premature nodes are c and d.

The parallel execution has higher memory requirements because of the space allocated by the computations associated with the premature nodes. Hence we need to bound the space overhead of the premature nodes in σ_p. To get this bound, we need to consider only the heavy premature nodes, since the light nodes do not allocate any space. We will assume for now that the computational steps of *all* heavy nodes allocate at most K space each. Later we will relax this assumption to cover bigger allocations. We first prove the following bound on the number of heavy premature nodes that get executed prematurely in any prefix of the parallel schedule.

Lemma 4 *Let G be a DAG of W nodes and depth D. Let s_1 be the* 1DF*-schedule for G, and s_p the parallel schedule for G executed by our scheduling algorithm on p processors. Then the number of heavy premature nodes in any prefix of s_p with respect to the corresponding prefix of s_1 is at most $2pD$.*

Proof: Consider an arbitrary prefix σ_p of s_p, and let σ_1 be the corresponding prefix of s_1. For any node in the DAG, we will call the length of the longest path from the root to the node its *level*. Let L_i be the set of nodes in σ_1 at level i. Let t_i be the earliest timestep in the parallel execution by which all the nodes in $L_1, L_2, \ldots, L_i$ get executed in the parallel schedule. For example, in Figure 10, with $\sigma_p = [\{a\}, \{b, c\}, \{e, g\}, \{i, k\}, \{f, d\}]$ and $\sigma_1 = [a, b, e, i, f]$, $L_1 = \{a\}$, $L_2 = \{b\}$, $L_3 = \{e, f\}$ and $L_4 = \{i\}$. Hence $t_1 = 1$, $t_2 = 2$, $t_3 = 5$ and $t_4 = 5$. Note that $i < j$ implies that $t_i \leq t_j$. After timestep t_i, all the nodes in L_{i+1} have either been executed, or are ready, since all the nodes with edges into them have been executed. Let interval I_i be the sequence of timesteps $(t_i, t_{i+1}]$, that is, $I_i = (t_i + 1), \ldots, t_{i+1}$.

Consider any interval I_i of the parallel execution. We show that no more than two of the scheduling threads that begin execution in the interval I_i can schedule heavy premature nodes. After timestep t_i, the nodes in L_{i+1} are either ready or have already been executed. All light nodes in L_{i+1} that are ready at timestep $t_i + 1$ are executed in that timestep. Further, all the heavy nodes in L_{i+1} get scheduled before any premature nodes are scheduled, because, given a choice between ready nodes, the scheduling processor schedules those that have lower 1DF numbers, and all nodes in σ_1 have lower 1DF numbers than any of the premature nodes. Once all the heavy nodes in L_{i+1} are scheduled, there may be at most two scheduling threads that begin execution before these nodes get executed, that is, before or at timestep t_{i+1}. These two scheduling threads may schedule nodes that are premature, since the nodes in the next level, L_{i+2}, may not be ready yet. The number of steps is two because of the asynchronous nature of the algorithm; there is a delay between the time a thread is inserted into the out-queue, and the time it begins execution.

Since at most two scheduling threads that begin execution in interval I_i may schedule premature nodes, and each scheduling thread can schedule at most p heavy nodes, at most $2p$ heavy nodes get scheduled in the interval I_i. Since the depth of the DAG is D, there can be at most D such intervals, and hence at most $2pD$ heavy premature nodes can be scheduled. ∎

We have shown that any prefix of s_p has at most $2pD$ heavy premature nodes. Since we have assumed that the computation associated with each heavy node allocates at most K space, we can state the following lemma.

Lemma 5 *Let G be a program DAG with depth D, in which every heavy node allocates at most K space. If the serial execution of the DAG requires S_1 space, then our scheduling algorithm results in an execution on p processors that requires at most $S_1 + 2pDK = S_1 + O(pD)$ space.* ∎

Handling allocations bigger that K. We described how to transform the program DAG to handle allocations bigger than K bytes in Section 4. Consider any heavy premature node v that allocates $m > K$ space. The m/K dummy nodes inserted before it would have been scheduled before it. Being dummy nodes, they do not actually allocate any space, but are entitled to allocate a total of m space (K units each) according to our scheduling technique. Hence v can allocate these m units without exceeding the space bound in Lemma 5. With this transformation, a parallel computation with W work and D depth that allocates a total of S_a units of memory results in a DAG with at most $W + S_a/K$ nodes and $O(D)$ depth. Therefore, using Lemma 5, we can state the following lemma.

Lemma 6 *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by our scheduling algorithm using $S_1 + O(pD)$ space.* ∎

Finally, we bound the space required by the scheduler to store the three queues.

Lemma 7 *The space required by the scheduler is $O(pD)$.*

Proof: When a processor starts executing a scheduling iteration, it first empties the in-queue. At this time, there can be at most $p - 1$ threads running on the other processors, and the out-queue can have another p threads in it. The scheduler adds at most another p threads (plus one scheduling thread) to the out-queue, and no more threads are added to the out-queue until the next scheduling iteration. Since all the threads executing on the processors can end up in the in-queue, the in-queue and out-queue can have a total of at most $3p$ threads at any time. Finally, we bound the number of threads in the ready-queue. We will call a thread that has been created but not yet deleted from the system a *live* thread; the ready queue has one entry for each live thread. At any stage during the execution, the number of live threads is at most the number of premature nodes scheduled, plus the maximum number of threads in the out-queue ($2p+1$), plus the maximum number of live threads in the 1DF-schedule. Any step of the 1DF-schedule can have at most D live threads, since it executes threads in a depth-first manner. Since the number of premature nodes is at most $2pD$, the ready-queue has at most $2pD + D + 2p + 1$ threads. Since each thread can be stored using a constant c units of memory, the total space required by the three scheduling queues is $(2pD + D + 5p + 1) \cdot c = O(pD)$. ∎

Theorem 2 follows from Lemmas 6 and 7.

A.2 Time bound

Finally, we prove Theorem 3, which bounds the time required to execute the parallel schedule generated by our scheduling algorithm. Our algorithm generates a *greedy* schedule, that is, whenever at least p nodes are ready, p nodes will be executed. Greedy schedules for DAGs with W nodes and D depth require at most $W/p+D$ timesteps to execute [6]. This bound does not include scheduling overheads. Since our transformed DAG has $W + S_a/K$ nodes and $O(D)$ depth, the schedule generated by our algorithm requires $O(W/p + S_a/Kp + D)$ timesteps. If the allocated space S_a is $O(W)$, the number of timesteps required is $O(W/p + D)$. ∎