Space-Efficient Scheduling of Nested Parallelism

GIRIJA J. NARLIKAR and GUY E. BLELLOCH
Carnegie Mellon University

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 article presents an on-line scheduling algorithm that is provably space efficient and time efficient for nested-parallel languages. For a computation with depth D and serial space requirement S_1, the algorithm generates a schedule that requires at most $S_1 + O(K \cdot D \cdot p)$ space (including scheduler space) on p processors. Here, K is a user-adjustable runtime parameter specifying the net amount of memory that a thread may allocate before it is preempted by the scheduler. Adjusting the value of K provides a trade-off between the running time and the memory requirement of a parallel computation. To allow the scheduler to scale with the number of processors, we also parallelize the scheduler and analyze the space and time bounds of the computation to include scheduling costs. In addition to showing that the scheduling algorithm is space and time efficient in theory, we demonstrate that it is effective in practice. We have implemented a runtime system that uses our algorithm to schedule lightweight 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.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*; F.2.0 [**Nonnumerical Algorithms and Problems**]: Analysis Of Algorithms and Problem Complexity

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Dynamic scheduling, multithreading, nested parallelism, parallel language implementation, space efficiency

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 [Blelloch et al. 1994] and HPF [HPF Forum 1993], as well as control-parallel languages such as ID [Arvind et al. 1989], Cilk [Blumofe et al.

Authors' address: CMU Computer Science Department, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: {narlikar, blelloch}@cs.cmu.edu.
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.

1995], CC++ [Chandy and Kesselman 1992], Sisal [Feo et al. 1990], Multilisp [Halstead 1985], Proteus [Mills et al. 1990], and C or C++ with lightweight thread libraries [Powell et al. 1991; Bershad et al. 1988; Mueller 1993]. 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.

Several systems providing dynamic parallelism have been implemented with efficient runtime schedulers [Blumofe and Leiserson 1994; Chandra et al. 1994; Chase et al. 1989; Freeh et al. 1994; Goldstein et al. 1995; Hseih et al. 1993; Hummel and Schonberg 1991; Nikhil 1994; Rinard et al. 1993; Rogers et al. 1995], 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 [Culler and Arvind 1988; Halstead 1985; Rugguero and Sargeant 1987]. 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. Because 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 limit the parallelism [Burton and Sleep 1981; Culler and Arvind 1988; Halstead 1985; Rugguero and Sargeant 1987] and are not guaranteed to be space efficient in general. These were followed by scheduling techniques that provide proven space bounds for parallel programs [Blumofe and Leiserson 1993; 1994; Burton 1988; Burton and Simpson 1994]. 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 [Blumofe et al. 1995].

A recent scheduling algorithm improved these space bounds from a multiplicative factor on the number of processors to an additive factor [Blelloch et al. 1995]. The algorithm generates a schedule that uses only $S_1 + O(D \cdot p)$ space, where D is the depth of the parallel computation (i.e., the length of 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 $D = o(S_1)$, 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 $S_1 + O(D \cdot p)$ is achieved by ensuring that

[1]More recent work provides a stronger upper bound than $p \cdot S_1$ for space requirements of regular

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

In this article we present and analyze an asynchronous scheduling algorithm called *AsyncDF*. This algorithm is a variant of the synchronous scheduling algorithm proposed in previous work [Blelloch et al. 1995] and overcomes the abovementioned problems. We also provide experimental results that demonstrate that the *AsyncDF* 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 nonpreemptively on the same processor until it runs out of memory from that pool (and reaches an instruction that requires more memory), or until it suspends. In practice, instead of preallocating a pool of K units of memory for each thread, we can assign it a counter that keeps track of its net memory allocation. We call this runtime, user-defined constant K the *memory threshold* for the scheduler. When an executing thread suspends or is preempted on a memory allocation, the processor accesses a new thread in a nonblocking manner from a work queue; the threads in the work queue are prioritized according to their depth-first, sequential execution order. The algorithm also delays threads performing large block allocations by effectively lowering their priority. Although the nonpreemptive and asynchronous nature of the *AsyncDF* algorithm results in an execution order that differs from the order generated by the previous algorithm [Blelloch et al. 1995], we show that it maintains an asymptotic space bound of $S_1 + O(K \cdot D \cdot p)$. Since K is typically fixed to be a small, constant amount of memory, the space bound reduces to $S_1 + O(D \cdot p)$, as before. This bound includes the space required by the scheduling data structures.

The scheduler in the *AsyncDF* algorithm is serialized, and it may become a bottleneck for a large number of processors. Therefore, to allow the scheduler to scale with the number of processors, this article also presents a parallelized version of the scheduler. We analyze both the space and time requirements of a parallel computation including the overheads of this parallelized scheduler. Using the parallelized scheduler, we show that a computation with W work (i.e., total number of operations), D depth, and a serial space requirement of S_1 can be executed on p processors using $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space. The additional $\log p$ factor arises because the parallelized scheduler creates more ready threads to keep the processors busy while the scheduler executes; this creation of additional parallelism is required to make the execution time efficient. When the total space allocated in the computation is $O(W)$ (e.g., when every allocated element is touched at least once), we show that the total time required for the parallel execution is $O(W/p + D \cdot \log p)$.

We have built a runtime system that uses the *AsyncDF* algorithm to schedule parallel threads on the SGI Power Challenge. To test its effectiveness in reduc-

divide-and-conquer algorithms in Cilk [Blumofe et al. 1996].

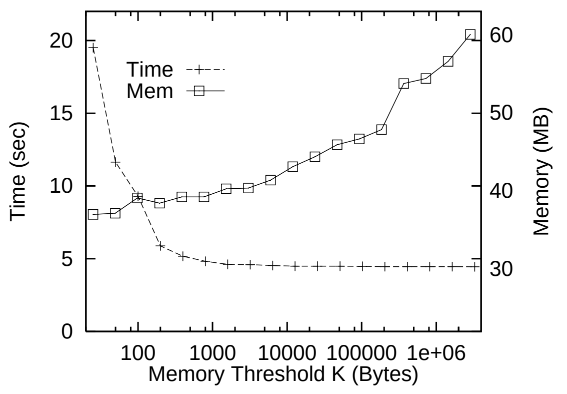

Fig. 1. The variation of running time and memory usage with the memory threshold K (in bytes) for multiplying two 1024×1024 matrices using blocked recursive matrix multiplication on 8 processors. K=500–2000 bytes results in both good performance and low memory usage.

ing 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.

A bigger value of the memory threshold K leads to a lower running time in practice because it allows threads to run longer without preemption and reduces scheduling costs. However, a large K also results in a larger space bound. The memory threshold parameter K therefore provides a trade-off between the running time and the memory requirement of a parallel computation. For example, Figure 1 experimentally demonstrates this trade-off for a parallel benchmark running on our runtime system. Section 7 describes the runtime system and the benchmark in detail. For all the benchmarks used in our experiments, a value of $K = 1000$ bytes yields good performance in both space and time.

The *AsyncDF* 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. Threads in our model are allowed to allocate memory from the shared heap or on their private stacks.

1.1 An Example

The following pseudocode 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 [Blumofe and Leiserson 1993; Burton 1988; Burton and Simpson 1994; Chow and W. L. Harrison 1990; Goldstein et al. 1995; Hummel and Schonberg 1991; Halstead 1985; Rugguero and Sargeant 1987], 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 *AsyncDF* 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. In general, our scheduler allows this parallel computation to run in just $O(n + D \cdot p)$ space,[2] 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, while 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, the *AsyncDF* scheduling algorithm would schedule one program at a time, 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 the *AsyncDF* algorithm is, that, because it often preferentially schedules inner parallelism (which is finer grained), it can cause large scheduling overheads and poor locality 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 7). In the experimental results reported in this article we have blocked the iterations into chunks by hand, but in Section 8 we discuss some ongoing work on automatically

[2]The additional $O(D \cdot p)$ memory is required due to the $O(D \cdot p)$ instructions that may execute "out of order" with respect to the serial execution order for this code.

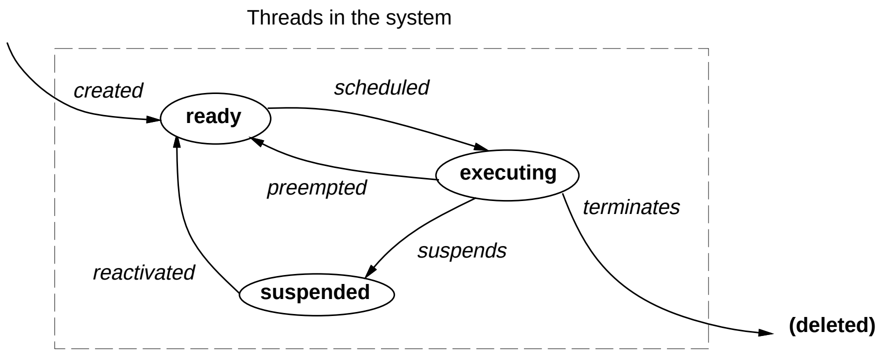

Fig. 2. The state transition diagram for threads. Thread in the system are either executing, ready, or suspended.

chunking the iterations. In general there is a trade-off between memory use and scheduling efficiency.

1.2 Outline of the Article

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 directed acyclic graph—this representation is used in the proofs throughout the article. Section 4 presents our online scheduling algorithm; the space and time bounds for schedules generated by it are proved in Section 5. In Section 6 we describe a parallelized scheduler and analyze the space and time requirements to include scheduling overheads. The implementation of our runtime system and the results of executing parallel programs on it are described in Section 7. Finally, we summarize and describe future work in Section 8.

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 *executing*, *ready* to execute, or *suspended*. A thread is said to be *scheduled* when a processor begins executing its instructions, that is, when the thread moves from the ready state to the executing state. When the executing thread subsequently *suspends* or *preempts* itself, it gives up the processor and leaves the executing state. A thread may suspend on a synchronization with one or more of its child threads. When a thread preempts itself due to a memory allocation (as explained below) it remains in the ready state. A thread that suspends must be *reactivated* (made ready) before it can be scheduled again. An executing thread that terminates is removed from the system. Figure 2 shows the state transition diagram for threads.

Each thread can be viewed as a sequence of actions; an *action* is a unit of computation that must be executed serially, and takes exactly one timestep (clock cycle) to be executed on a processor. A single action may allocate or deallocate space.

Since instructions do not necessarily complete in a single timestep, one machine instruction may translate to a series of multiple actions. The *work* of a parallel computation is the total number of actions executed in it, that is, the number of timesteps required to execute it serially (ignoring scheduling overheads). The *depth* of a parallel computation is the length of the critical path, or the time required to execute the computation on an infinite number of processors (again, ignoring scheduling overheads). The space requirement of an execution is the maximum of the total memory allocated across all processors at any time during the execution, that is, the high-water mark of total memory allocation.

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 action 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.

To maintain our space bounds, we impose an additional restriction on the threads. Every time a ready thread is scheduled, it may perform a memory allocation from a global pool of memory only in its first action. The memory allocated becomes its private pool of memory, and may be subsequently used for a variety of purposes, such as, for dynamically allocated heap data or activation records on its stack. When the thread runs out of its private pool and reaches an action that needs to allocate more memory, the thread must preempt itself by giving up its processor and moving from the executing state back to the ready state. The next time the thread is scheduled, the first of its actions to execute may once again allocate a pool of memory from the global pool, and so on. The reason threads are preempted just before they allocate more space is to allow ready 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. REPRESENTING THE COMPUTATION AS A DAG

To formally define and analyze the space and time requirements of a parallel computation, we view the computation as a precedence graph, that is, a *directed acyclic graph* or *DAG.* Each node in the DAG corresponds to an action. The edges of the DAG express the dependencies between the actions. We will refer to the amount of memory allocated from the global pool by the action corresponding to a node v as $m(v)$. If the action performs a deallocation, $m(v)$ is negative; we assume that a single action does not perform both an allocation and a deallocation. The DAG unfolds dynamically and can be viewed as a trace of the execution.

For the nested parallelism model described in Section 2, the dynamically unfolding DAG has a *series-parallel* structure. A series-parallel DAG [Blelloch et al. 1995] can be defined inductively: the DAG G_0 consisting of a single node (which is both its source and sink) and no edges is a series-parallel DAG. If G_1 and G_2

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

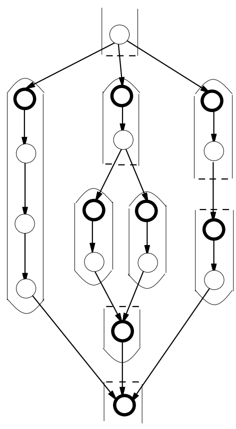

Fig. 3. A portion of a program DAG and the threads that perform its computation. Each node corresponds to a single action; the edges express dependencies between actions. The threads are shown as outlines around the nodes. Dashed lines mark points in a thread where it gets suspended or preempted and is subsequently scheduled again. Only the initial node of each newly scheduled thread may allocate memory, that is, may allocate space from the global pool to be used as the local pool. Such nodes are shown in bold. Any node may deallocate memory.

are series-parallel DAGs, then the graph obtained by adding to $G_1 \cup G_2$ a directed edge from the sink node of G_1 to the source node of G_2 is a series-parallel DAG. If $G_1, \ldots, G_n$, for $n \geq 1$, are series-parallel DAGs, 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 also a series-parallel DAG.

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 actions (units of computation) 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. Because 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 3 shows a small portion of a DAG. Only the first node of a newly scheduled thread may allocate space; a thread must preempt itself on reaching a subsequent node that performs an allocation. The points where threads are suspended or preempted, and subsequently scheduled again, 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. For our space and time analysis we assume that the clocks (timesteps) of the processors are synchronized. Therefore, although we are modeling asynchronous parallel computations, the schedules are represented as sets of nodes executed in discrete timesteps. With this assumption, any execution of the computation on p processors that takes T timesteps can be represented by a *p-schedule* $s_p = V_1, V_2, \ldots, V_T$, 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

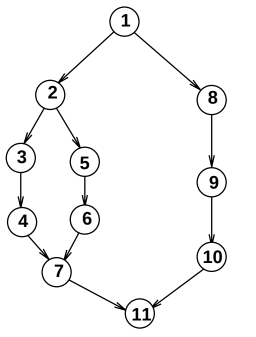

Fig. 4. The 1DF-schedule for a program DAG. Each node is labeled with the order in which it is executed. A node cannot be executed until all the nodes that have edges into it have been executed. Serial executions of computations on most systems execute nodes in this order.

contains at most p nodes. A serial execution of a computation can be represented by a 1-schedule $s_1 = V_1, V_2, \ldots, V_T$, where each set V_i consists of at most one node. Every 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.

We define the space requirement of a parallel (or serial) execution as the high water mark of total memory allocated across all the processors. Therefore, space required by a parallel execution represented as $s_p = V_1, V_2, \ldots, V_T$ is defined as $S_p = n + max_{j=1,\ldots,T}\left(\sum_{i=1}^{j}\sum_{v\in V_i} m(v)\right)$, where n is the space required to store the input. We use the term S_1 to refer to the space requirement of a serial execution.

1DF-schedule. Several different serial schedules (with different space requirements) may exist for a single DAG. However, serial implementations of most languages execute nodes according to a unique, left-to-right, *depth-first* schedule or *1DF-schedule* of the 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 (when children are placed from left to right in program text order). The order in which the nodes are executed in a 1DF-schedule determines their *1DF-numbers.* For example, Figure 4 shows the 1DF-numbering of a simple program DAG. For the class of nested parallel computations, the most natural serial execution follows a 1DF-schedule. Therefore, in the rest of this article, we refer to the 1DF-schedule as the serial schedule s_1, and its space requirement as the serial space requirement S_1. In practice, due to effects such as caching, the exact size of the DAG for a given computation can depend on the particular schedule. For example, the number of clock cycles (actions) required for a given instruction can vary based on the location of its data in the memory hierarchy, which may be affected by the scheduling order. For deterministic programs, however, the value of S_1 is independent of the schedule (or execution), since the number or size of allocations is not affected by such timing effects.

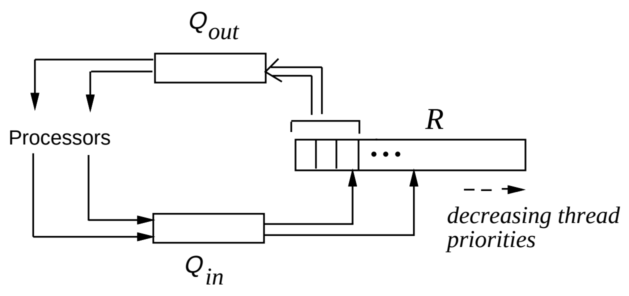

Fig. 5. The movement of threads between the processors and the scheduling queues. Q_{in} and Q_{out} are FIFOs, whereas $\mathcal{R}$ allows insertions and deletions of intermediate threads. Every executing, suspended, or ready thread has an entry in $\mathcal{R}$. Threads in $\mathcal{R}$ are stored from left to right in increasing order of the 1DF-numbers of their leading nodes (the first nodes to be executed when the thread gets scheduled).

4. THE *ASYNCDF* SCHEDULING ALGORITHM

The key idea behind our scheduling algorithm is to schedule threads in an order that is sufficiently close to a 1DF-numbering (serial ordering) of their nodes. Since we must schedule multiple threads simultaneously to keep all the processors busy, some 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 *AsyncDF* 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 (the user-adjustable memory threshold), and that it preempts itself when it reaches an action requiring more memory. This assumption is valid if no single action 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, $\mathcal{R}$, 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 $\mathcal{R}$ are stored from 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 $\mathcal{R}$ 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 Q_{in} and Q_{out}. 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 $\mathcal{R}$ (see Figure 5); processors can perform fast, nonblocking accesses to these queues. In this section we use a serial algorithm to move threads between the buffers and $\mathcal{R}$; Section 6 describes how this can be done in parallel.

```
begin worker
   while (there exist threads in the system)
      $\tau$ := remove-thread($Q_{out}$);
      if ($\tau$ is a scheduling thread) then scheduler()
      else
         execute the actions associated with $\tau$;
         if ($\tau$ terminates) or ($\tau$ suspends) or ($\tau$ preempts itself)
         then insert-thread($\tau$, $Q_{in}$);
end worker

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

Fig. 6. The *AsyncDF* scheduling algorithm. When the scheduler forks (creates) child threads, it inserts them into $\mathcal{R}$ in the immediate left of their parent thread. This maintains the invariant that the threads in $\mathcal{R}$ 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 Q_{out}. Child threads are forked only when they are to be added to Q_{out}, that is, when they are among the leftmost p ready threads in $\mathcal{R}$.

Algorithm Description. The pseudocode for the *AsyncDF* scheduling algorithm is given in Figure 6. The processors normally act as *workers*, when they take threads from Q_{out}, execute them until they preempt themselves, suspend or terminate, and then return them to Q_{in}. Every time a ready thread is picked from Q_{out} and scheduled on a worker processor, it may allocate space from a global pool in its first action. The thread must preempt itself before any subsequent action that requires more space. A thread that performs a fork must suspend itself; it is reactivated 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 from Q_{out} by a processor turns out to be a scheduling thread, it assumes the role of the scheduling processor and executes the *scheduler* procedure. We call each execution of the *scheduler* procedure a *scheduling step*. Only one processor can be executing a scheduling step at a time due to the scheduler lock. The algorithm begins with a scheduling thread and the first (root) thread of the program on Q_{out}.

A processor that executes a scheduling step starts by putting a new scheduling thread on Q_{out}. Next, it moves all the threads from Q_{in} to $\mathcal{R}$. Each thread has a pointer to a stub that marks its original position relative to the other threads in $\mathcal{R}$; it is inserted back in that position. All threads that were preempted due to a memory allocation are returned to $\mathcal{R}$ in the ready state. The scheduler then compacts $\mathcal{R}$ 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. If a thread performs a fork, its child threads are inserted to its immediate left, and the forking thread suspends. The child threads are placed in $\mathcal{R}$ in order of the 1DF-numbers of their leading nodes. Finally, the scheduler moves the *leftmost* p ready threads from $\mathcal{R}$ to Q_{out}, leaving behind stubs to mark their positions in $\mathcal{R}$. If $\mathcal{R}$ contains less than p ready threads, the scheduler moves them all to Q_{out}. 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 Q_{in} and Q_{out} is at most $3p$ (see Lemma 5.1.4). Further, to limit the number of threads in $\mathcal{R}$, we *lazily* create the child threads of a forking thread: a child thread is not explicitly created until it is to be moved to Q_{out}, that is, when it is among the leftmost p threads represented in $\mathcal{R}$. 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 $\mathcal{R}$ until it has been scheduled at least once before, or is in (or about to be inserted into) Q_{out}. If a thread τ is ready to fork child threads, all its child threads will be forked (created) and scheduled before any other threads in $\mathcal{R}$ to the right of τ 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, where K is the constant memory threshold. This does not allow any single action within a thread to allocate more than K bytes. We now show how such allocations are handled, similar to the technique suggested in previous work [Blelloch et al. 1995]. 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 7). 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 $W + S_a/K$. The transformation takes place at runtime, and the on-line *AsyncDF* 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 Section 5.

We state the following lemma regarding the order of the nodes in $\mathcal{R}$ maintained by the *AsyncDF* algorithm.

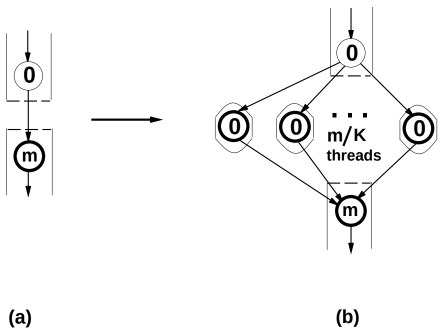

Fig. 7. A transformation of the DAG to handle a large allocation of space at a node without violating the space bound. Each node is labeled with the amount of memory its action allocates. When a thread needs to allocate m space ($m > K$), we insert m/K parallel child threads before the allocation. K is the constant memory threshold. 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 4.1. *The AsyncDF scheduling algorithm always maintains the threads in $\mathcal{R}$ in an increasing order of the 1DF-numbers of their leading nodes.*

PROOF. This lemma can be proved by induction. When the execution begins, $\mathcal{R}$ 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 $\mathcal{R}$ 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 preempts itself due to a memory allocation is returned to its original position in the ready state. Its new leading node has the same 1DF-number as its previous leading node, *relative* to leading nodes of other threads. Deleting threads from $\mathcal{R}$ does not affect their ordering. Therefore the ordering of threads in $\mathcal{R}$ by 1DF-numbers is preserved after every operation performed by the scheduler. □

Lemma 4.1 implies that when the scheduler moves the leftmost p threads from $\mathcal{R}$ to Q_{out}, 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

In this section, we prove that a parallel computation with depth D and work W, which requires S_1 space to execute on one processor, is executed by the *AsyncDF* scheduling algorithm on p processors using $S_1 + O(D \cdot p)$ space (including scheduler space). In Section 6 we describe the implementation of a parallelized scheduler and analyze both the space and time bounds including scheduler overheads.

Recall that we defined a single action as the work done by a thread in one timestep (clock cycle). Since the granularity of a clock-cycle is somewhat arbitrary, especially considering highly pipelined processors with multiple functional units, this would seem to make the exact value of the depth D somewhat arbitrary. For asymptotic bounds this is not problematic, since the granularity will only make constant factor differences. In Appendix A, however, we modify the space bound to be independent of the granularity of actions, making it possible to bound the space requirement within tighter constant factors.

Timing Assumptions. As explained in Section 3, the timesteps are synchronized across all the processors. At the start of each timestep, we assume that a worker processor is either busy executing a thread or is accessing the queues Q_{out} or Q_{in}. An idle processor always busy waits for threads to appear in Q_{out}. We assume a constant-time, atomic fetch-and-add operation in our system. This allows all worker processors to access Q_{in} and Q_{out} in constant time [Narlikar 1999]. Thus, at any timestep, if Q_{out} has n threads, and p_i processors are idle, then $\min(n, p_i)$ of the p_i idle processors are guaranteed to succeed in picking a thread from Q_{out} within a constant number of timesteps. We do not need to limit the duration of each scheduling step to prove the space bound; we simply assume that it takes at least one timestep to execute.

5.1 Space Bound

To prove the space bound, we partition the nodes of the computation DAG into heavy and light nodes. Every time a ready thread is scheduled, we call the node representing its first action (i.e., the thread's leading node) a *heavy* node, and all other nodes *light* nodes. Thus, heavy nodes may allocate space, while light nodes allocate no space (but may deallocate space). The space requirement is analyzed by bounding the number of heavy nodes that execute out of order with respect to the 1DF-schedule. When a thread is moved from $\mathcal{R}$ to Q_{out} by a scheduling processor, we will say its leading heavy node has been inserted into Q_{out}. A heavy node may get executed several timesteps after it becomes ready and after it is put into Q_{out}. However, a light node is executed in the timestep it becomes ready, because a processor executes consecutive light nodes nonpreemptively.

Let $s_p = V_1, \ldots, V_T$ be the parallel schedule of the DAG generated by the *AsyncDF* 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. A *prefix* of s_p is the set $\bigcup_{i=1}^{j} V_i$, that is, the set of all nodes executed during the first j timesteps of s_p, for any $1 \leq j \leq T$. Consider an arbitrary prefix, σ_p, of s_p. Let σ_1 be the largest 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. Then σ_1 is the *corresponding* serial prefix of σ_p. We call the nodes in $\sigma_p - \sigma_1$ the *premature* nodes, because they have been executed out of order with respect to s_1. All other nodes in σ_p (i.e., all nodes in the set σ_1) are called *nonpremature.* For example, Figure 8 shows a simple DAG with a parallel prefix σ_p for some arbitrary p-schedule, and its corresponding serial prefix σ_1.

The parallel execution has higher memory requirements because of the space allocated by the actions 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 actions corresponding to *all* heavy nodes allocate at most K space each, where K is the user-specified memory threshold of the scheduler. Later we will relax this assumption to cover bigger allocations. We first prove the following bound on the number of heavy nodes that

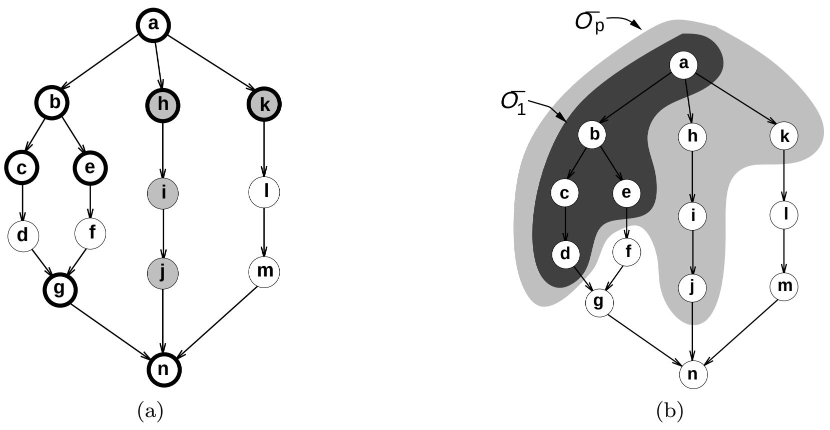

Fig. 8. (a) A simple program DAG in which the heavy nodes (a, b, c, e, g, h, k, and n) are shown as bold. The 1DF-schedule for this DAG is $s_1 = [a, b, c, d, e, f, g, h, i, j, k, l, m, n]$. For $p = 2$, a possible parallel schedule is $s_p = [\{a\}, \{b, h\}, \{c, i\}, \{d, j\}, \{e, k\}, \{f, l\}, \{g, m\}, \{n\}]$. In this schedule, once a heavy node is executed, the processor continues to execute the subsequent light nodes of the thread. (b) shows a prefix $\sigma_p = \{a, b, h, c, i, d, j, e, k\}$ of s_p (after the first 5 timesteps), and the corresponding prefix σ_1 of s_1, where $\sigma_1 = \{a, b, c, d, e\}$. Thus, the premature nodes in σ_p (i.e., the nodes in $\sigma_p - \sigma_1$) are h, i, j, and k, shown shaded in (a). Of these, the heavy premature nodes are h and k.

get executed prematurely in any prefix of the parallel schedule.

Lemma 5.1.1. *Let G be a directed acyclic graph (DAG) of W nodes and depth D. Let s_1 be the 1DF-schedule for G, and let s_p be a parallel schedule for G executed by the AsyncDF 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 $O(D \cdot p)$.*

Proof. Consider an arbitrary prefix σ_p of s_p, and let σ_1 be the corresponding prefix of s_1. Let v be the last nonpremature node to be executed in the prefix σ_p; if there are two or more such nodes, pick any one of them. Let P be a path in the DAG from the root to v constructed such that, for every edge (u, u') along P, u is the last parent (or any one of the last parents) of u' to be executed. (For example, for the σ_p shown in Figure 8, the path P is (a, b, e).) Since v is nonpremature, all the nodes in P are nonpremature.

Let u_i be the node on the path P at depth i; then u_1 is the root, and u_δ is the node v, where δ is the depth of v. Let t_i be the timestep in which u_i is executed; let $t_{\delta+1}$ be the last timestep in σ_p. For $i = 1, \ldots, \delta$, let I_i be the interval $\{t_i + 1, \ldots, t_{i+1}\}$.

Consider any interval I_i for $i = 1, \ldots, \delta - 1$. We now show that at most $O(p)$ heavy premature nodes can be executed in this interval. At the end of timestep t_i, u_i has been executed. If u_{i+1} is a light node, it gets executed in the next timestep (which, by definition, is timestep t_{i+1}), and at most another $(p-1)$ heavy premature nodes can be executed in the same timestep, that is, in interval I_i.

Consider the case when u_{i+1} is a heavy node. After timestep t_i, Q_{out} may contain p nodes. Further, because access to Q_{in} requires constant time, the thread τ that contains u_i must be inserted into Q_{in} within a constant number of timesteps after

t_i. During these timesteps, a constant number of scheduling steps may be executed, adding another $O(p)$ threads into Q_{out}. Thus, because Q_{out} is a FIFO, a total of $O(p)$ heavy nodes may be picked from Q_{out} and executed before u_{i+1}; all of these heavy nodes may be premature. However, once the thread τ is inserted into Q_{in}, the next scheduling step must find it in Q_{in}; since u_i is the last parent of u_{i+1} to execute, this scheduling step makes u_{i+1} available for scheduling. Thus, this scheduling step or any subsequent scheduling step must put u_{i+1} on Q_{out} before it puts any more premature nodes, because u_{i+1} has a lower 1DF-number. When u_{i+1} is picked from Q_{out} and executed by a worker processor, another $p-1$ heavy premature nodes may get executed by the remaining worker processors in the same timestep, which, by definition, is timestep t_{i+1}. Thus, a total of $O(p)$ heavy premature nodes may be executed in interval I_i. Similarly, since $v = u_\delta$ is the last nonpremature node in σ_p, at most $O(p)$ heavy premature nodes get executed in the last interval I_δ. Because $\delta \le D$, σ_p contains a total of $O(D \cdot p)$ heavy premature nodes. □

We have shown that any prefix of s_p has at most $O(D \cdot p)$ heavy premature nodes. Since we have assumed that the action associated with each heavy node allocates at most K space, we can prove the following lemma.

LEMMA 5.1.2. *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 the AsyncDF scheduling algorithm results in an execution on p processors that requires at most $S_1 + O(K \cdot D \cdot p)$space.*

PROOF. Consider any parallel prefix σ_p of the parallel schedule generated by algorithm *AsyncDF*; let σ_1 be the corresponding serial prefix. The net memory allocation of the nodes in σ_1 is at most S_1, because σ_1 is a prefix of the serial schedule. Further, according to Lemma 5.1.1, the set $\sigma_p - \sigma_1$ has $O(D \cdot p)$ heavy nodes, each of which may allocate at most K space. Therefore, the net space allocated by all the nodes in σ_p is at most $S_1 + O(K \cdot D \cdot p)$. Since this bound holds for any arbitrary prefix of the parallel execution, the entire parallel execution also uses at most $S_1 + O(K \cdot D \cdot p)$ space. □

Handling Allocations Bigger than the Memory Threshold 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 executed 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 5.1.2. 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 5.1.2, we can state the following lemma.

LEMMA 5.1.3. *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by the AsyncDF algorithm using $S_1 + O(K \cdot D \cdot p)$ space.* □

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

LEMMA 5.1.4. *The space required by the scheduler is $O(D \cdot p)$.*

PROOF. When a processor starts executing a scheduling step, it first empties Q_{in}. At this time, there can be at most $p - 1$ threads running on the other processors, and Q_{out} can have another p threads in it. The scheduler adds at most another p threads (plus one scheduling thread) to Q_{out}, and no more threads are added to Q_{out} until the next scheduling step. Since all the threads executing on the processors can end up in Q_{in}, Q_{in} and Q_{out} can have a total of at most $3p$ threads at any time. Finally, we bound the number of threads in $\mathcal{R}$. We will call a thread that has been created but not yet deleted from the system a *live* thread; $\mathcal{R}$ 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 executed, plus the maximum number of threads in Q_{out} (which is $2p + 1$), plus the maximum number of live threads in the 1DF-schedule. Any step of the 1DF-schedule can have at most D live threads, because it executes threads in a depth-first manner. Since the number of premature nodes is at most $O(D \cdot p)$, $\mathcal{R}$ has at most $O(D \cdot p + D + 2p + 1) = O(D \cdot p)$ threads. Because each thread requires a small, constant c units of memory to store its state,[4] the total space required by the three scheduling queues is $O(c \cdot D \cdot p) = O(D \cdot p)$. □

Using Lemmas 5.1.3 and 5.1.4, we can now state the following bound on the total space requirement of the parallel computation.

THEOREM 5.1.5. *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by the AsyncDF algorithm using $S_1 + O(K \cdot D \cdot p)$ space (including scheduler space).* □

In practice, K is set to a small, constant amount of memory throughout the execution of the program (see Section 7), reducing the space bound to $S_1 + O(D \cdot p)$.

5.2 Time Bound

Finally, we bound the time required to execute the parallel schedule generated by the scheduling algorithm for a special case; Section 6 analyzes the time bound in the general case. In this special case, we assume that the worker processors never have to wait for the scheduler to add ready threads to Q_{out}. Thus, when there are r ready threads in the system, and n processors are idle, Q_{out} has at least $\min(r, n)$ ready threads. Then $\min(r, n)$ of the idle processors are guaranteed to pick ready threads from Q_{out} within a constant number of timesteps. We can show that the time required for such an execution is within a constant factor of the time required to execute a greedy schedule. A *greedy* schedule is one in which at every timestep, if n nodes are ready, $\min(n, p)$ of them get executed. Previous results have shown that greedy schedules for DAGs with W nodes and D depth require at most $W/p + D$ timesteps to execute [Blumofe and Leiserson 1993]. Our transformed DAG has $W + S_a/K$ nodes and $O(D)$ depth. Therefore, we can show that our scheduler requires $O(W/p + S_a/pK + D)$ timesteps to execute on p processors. When the

[4]Recall that a thread allocates stack and heap data from the global pool of memory that is assigned to it every time it is scheduled; the data are hence accounted for in the space bound proved in Lemma 5.1.3 Therefore, the thread's state here refers simply to its register contents.

allocated space S_a is $O(W)$, the number of timesteps required is $O(W/p + D)$. For a more in-depth analysis of the running time that includes the cost of a parallelized scheduler in the general case, see Section 6.

6. A PARALLELIZED SCHEDULER

The time bound in Section 5 was proved for the special case when the *scheduler* procedure never becomes a bottleneck in making ready threads available to the worker processors. However, recall that the scheduler in the *AsyncDF* algorithm is a serial scheduler, that is, only one processor can be executing the *scheduler* procedure at a given time. Further, the time required for this procedure to execute may increase with the number of processors, causing idle worker processors to wait longer for ready threads to appear Q_{out}. Thus, the scheduler may indeed become a bottleneck on a large number of processors. Therefore, the scheduler must be parallelized to scale with the number of processors. In this section, we describe a parallel implementation of the scheduler and analyze its space and time costs. We prove that a computation with W work and D depth can be executed in $O(W/p + S_a/p + D \cdot \log p)$ time and $S_1 + O(D \cdot p \cdot \log p)$ space on p processors; these bounds include the overheads of the parallelized scheduler. The additional $\log p$ term in the time bound arises due to the parallel prefix operations executed by the scheduler. The $\log p$ term in the space bound is due to the additional number of ready threads created to keep worker processors busy while the scheduler executes.

We give only a theoretical description of a parallelized scheduler in this article; the experimental results presented in Section 7 have been obtained using the serial scheduler from Figure 6. As our results on up to 16 processors demonstrate, the serial scheduler provides good performance on this moderate number of processors.

6.1 Parallel Implementation of a Lazy Scheduler

Instead of using scheduler threads to periodically (and serially) execute the *scheduler* procedure as shown in Figure 6, we devote a constant fraction αp of the processors ($0 < \alpha < 1$) to it. The remaining $(1 - \alpha)p$ processors always execute as workers. To amortize the cost of the scheduler, we place a larger number of threads (up to $p \log p$ instead of p) into Q_{out}. As in Section 5, we assume that a thread can be inserted or removed from Q_{in} or Q_{out} by any processor in constant time. The data structure $\mathcal{R}$ is implemented as an array of threads, stored in decreasing order of priorities from left to right.

As described in Section 4, threads are forked lazily; when a thread reaches a fork, it is simply marked as a *seed* thread. At a later time, when its child threads are to be scheduled, they are placed to the immediate left of the seed in order of their 1DF-numbers. Similarly, we perform all deletions lazily: every thread that terminates is simply marked in $\mathcal{R}$ as a *dead* thread, to be deleted in some subsequent timestep.

The synchronization (join) between child threads of a forking thread is implemented using a fetch-and-decrement operation on a synchronization counter associated with the fork. Each child that reaches the synchronization point decrements the counter by one and checks its value. If the counter has nonzero value, it simply mark itself as dead. The last child thread to reach the synchronization point (the one that decrements the counter to zero) marks itself as ready in $\mathcal{R}$, and subsequently continues as the parent. Thus, when all the child threads have been created, the seed that originally represented the parent thread can be deleted.

We will refer to all the threads that have an entry in $\mathcal{R}$ but are not dead as *live* threads. For every ready (or seed) thread τ that has an entry in $\mathcal{R}$, we use a nonnegative integer $c(\tau)$ to describe its state: $c(\tau)$ equals the number of ready threads τ represents. Then for every seed τ, $c(\tau)$ is the number of child threads still to be created from it. For every other ready thread in $\mathcal{R}$, $c(\tau) = 1$, because it represents itself.

The *scheduler* procedure from Figure 6 can now be replaced by a while loop that runs until the entire computation has been executed. Each iteration of this loop, which we call a *scheduling iteration*, is executed in parallel by only the αp scheduler processors. Therefore, it need not be protected by a scheduler lock as in Figure 6. Let r be the total number of ready threads represented in $\mathcal{R}$ after threads from Q_{in} are moved to $\mathcal{R}$ at the beginning of the iteration. Let $q_o = min(r, p \log p - |Q_{out}|)$ be the number of threads the scheduling iteration will move to Q_{out}. The scheduling iteration of a lazy scheduler is defined as follows.

(1) Collect all the threads from Q_{in} and move them to $\mathcal{R}$, that is, update their states in $\mathcal{R}$.

(2) Delete all the dead threads up to the leftmost $(q_o + 1)$ ready or seed threads.

(3) Perform a prefix-sums computation on the $c(\tau)$ values of the leftmost q_o ready or seed threads to find the set C of the leftmost q_o ready threads represented by these threads. For every thread in C that is represented implicitly by a seed, create an entry for the thread in $\mathcal{R}$, marking it as a ready thread. Mark the seeds for which all child threads have been created as dead.

(4) Move the threads in the set C from $\mathcal{R}$ to Q_{out}, leaving stubs in $\mathcal{R}$ to mark their positions.

Consider a thread τ that is the last child thread to reach the synchronization point in a fork, but was not the rightmost thread among its siblings. Some of τ's siblings, which have terminated, may be represented as dead threads to its right. Since τ now represents the parent thread after the synchronization point, it has a higher 1DF-number than these dead siblings to its immediate right. Thus, due to lazy deletions, dead threads may be out of order in $\mathcal{R}$. However, the scheduler deletes all dead threads up to the first $(q_o + 1)$ ready or seed threads, that is, all dead threads to the immediate right of any ready thread (or seed representing a ready thread) before it is scheduled. Therefore, no descendents of a thread may be created until all dead threads out of order with respect to the thread are deleted. Thus, a thread may be out of order with only the dead threads to its immediate right.

We say a thread is *active* when it is either in Q_{out} or Q_{in}, or when it is being executed on a processor. Once a scheduling iteration empties Q_{in}, at most $p \log p + (1 - \alpha)p$ threads are active. The iteration creates at most another $p \log p$ active threads before it ends, and no more threads are made active until the next scheduling step. Therefore at most $2p \log p + (1 - \alpha)p$ threads can be active at any timestep, and each has one entry in $\mathcal{R}$. We now prove the following bound on the time required to execute a scheduling iteration.

LEMMA 6.1.1. *For any $0 < \alpha < 1$, a scheduling iteration that deletes n dead threads runs in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ time on αp processors.*

PROOF. Let $q_o \leq p \log p$ be the number of threads the scheduling iteration must move to Q_{out}. At the beginning of the scheduling iteration, Q_{in} contains at most $2p \log p + (1 - \alpha)p$ threads. Since each of these threads has a pointer to its stub in $\mathcal{R}$, αp processors can move the threads to $\mathcal{R}$ in $O(\frac{\log p}{\alpha})$ time. Let τ be the $(q_o + 1)th$ ready or seed thread in $\mathcal{R}$ (starting from the left end). The scheduler needs to delete all dead threads to the left of τ. In the worst case, all the stubs are also to the left of τ in $\mathcal{R}$. However, the number of stubs in $\mathcal{R}$ is at most $2p \log p + (1 - \alpha)p$. Because there are n dead threads to the left of τ, they can be deleted from $n + 2p \log p + (1 - \alpha)p$ threads in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ timesteps on αp processors. After the deletions, the leftmost $q_o \leq p \log p$ ready threads are among the first $3p \log p + (1 - \alpha)p$ threads in $\mathcal{R}$; therefore the prefix-sums computation will require $O(\frac{\log p}{\alpha})$ time. Finally, q_o new child threads can be created and added in order to the left end of $\mathcal{R}$ in $O(\frac{\log p}{\alpha})$ time. Note that all deletions and additions are at the left end of $\mathcal{R}$, which are simple operations in an array.[5] Thus, the entire scheduling iteration runs in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ time. □

6.2 Space and Time Bounds Using the Parallelized Scheduler

We now state the space and time bounds of a parallel computation, including scheduling overheads. The bounds assume that a constant fraction α of the p processors (for any $0 < \alpha < 1$) are dedicated to the task of scheduling. The detailed proofs are given in Appendix B.

THEOREM 6.2.1. *Let S_1 be the space required by a 1DF-schedule for a computation with work W and depth D, and let S_a be the total space allocated in the computation. The parallelized scheduler with a memory threshold of K units, generates a schedule on p processors that requires $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space and $O(W/p + S_a/pK + D \cdot \log p)$ time to execute.* □

These time and space bounds include scheduling overheads. The time bound is derived by counting the total number of timesteps during which the worker processors may be either idle or busy executing actions. The space bound is proved using an approach similar to that used in Section 5. When the total space allocated $S_a = O(W)$, the time bound reduces to $O(W/p + D \cdot \log p)$. As with the serial scheduler, when the memory threshold K is set to a constant, the asymptotic space bound reduces to $S_1 + O(D \cdot p \cdot \log p)$.

7. EXPERIMENTAL RESULTS

We have built a runtime system that uses the *AsyncDF* 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.

[5]The additions and deletions must skip over the stubs to the left of τ, which can add at most a $\frac{\log p}{\alpha}$ delay.

7.1 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. We implemented the serial version of the scheduler presented in Figure 6, because the number of processors on this architecture is not very large. The set of ready threads $\mathcal{R}$ is implemented as a simple, singly-linked list. Q_{in} and Q_{out}, which are accessed by the scheduler and the workers, are required to support concurrent enqueue and dequeue operations. They are implemented using variants of previous lock-free algorithms based on atomic fetch-and-Φ primitives [Mellor-Crummey 1987].

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[6] [Blumofe et al. 1995]. 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 Q_{out}, 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 equally sized chunks, provided it does not contain calls to any recursive functions.[7] Scheduling a chunk at a time improves performance by reducing scheduling overheads and providing good locality, especially for fine-grained iterations.

Instead of preallocating a pool of memory for a thread every time it is scheduled, we use a *memory counter* to keep track of a thread's net memory allocation. The memory counter is initialized to the value of the memory threshold K when the thread is scheduled. The counter is appropriately decremented (incremented) when the thread allocates (deallocates) space. When the thread reaches a memory allocation that requires more memory than the current value of the counter, the thread is preempted, and the counter is reset to K units. Instead of explicitly creating dummy threads to delay an allocation of m bytes ($m > K$) in a thread, the thread is inserted into $\mathcal{R}$ with a *delay counter* initialized to the value m/K. The delay counter is appropriately decremented by the scheduling thread; each decrement by 1 represents the creation and scheduling of one dummy thread. The original thread is ready to execute once the value of the delay counter is reduced to zero. Unless stated otherwise, all the experiments described in this section were performed using $K = 1000$ bytes.

7.2 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.

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

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

(1) *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.

(2) *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 [Strassen 1969]. 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.

(3) *Fast multipole method (FMM).* This is an n-body algorithm that calculates the forces between n bodies using $O(n)$ work [Greengard 1987]. 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.

(4) *Sparse matrix-vector multiplication (Sparse MV).* This multiplies an $m \times n$ sparse matrix with an $n \times 1$ 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 $n = 1,500,000$, and 30% of the elements were nonzeroes. (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.)

(5) *ID3.* The ID3 algorithm [Quinlan 1986] 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

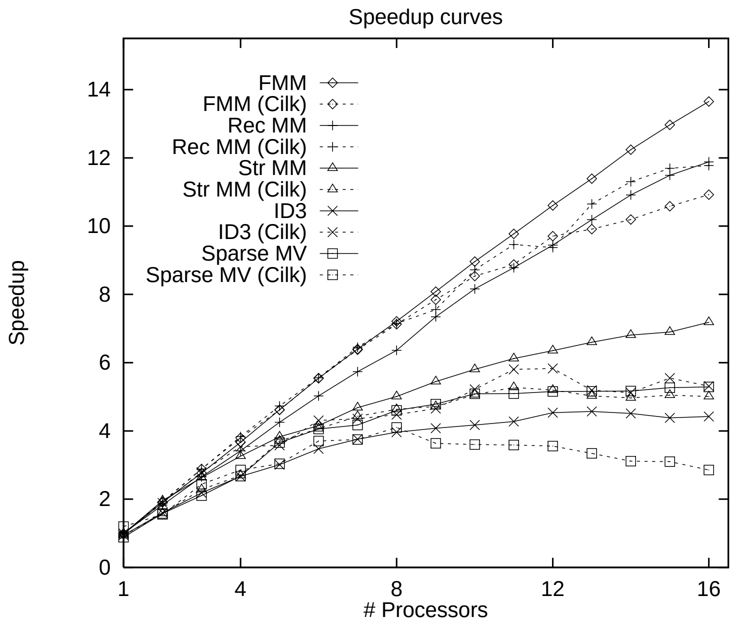

Fig. 9. The speedups achieved on up to 16 R10000 processors of a Power Challenge machine, using a value of K=1000 bytes. The speedup on p processors is the time taken for the serial C version of the program divided by the time for our runtime system to run it on p processors. For each application, the 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`.

is built. We built a tree from 4 million test examples, each with 4 multivalued attributes.

7.3 Time Performance

Figure 9 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 ($K = 1000$ bytes) for an allocation of m bytes. Figure 9 also shows the speedups for the same programs running on an existing space-efficient system, Cilk [Blumofe et al. 1995], version 5.0. 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 10 shows the breakdown of the running time for one of the programs, blocked recursive matrix multiplication. The results show that the percentage of

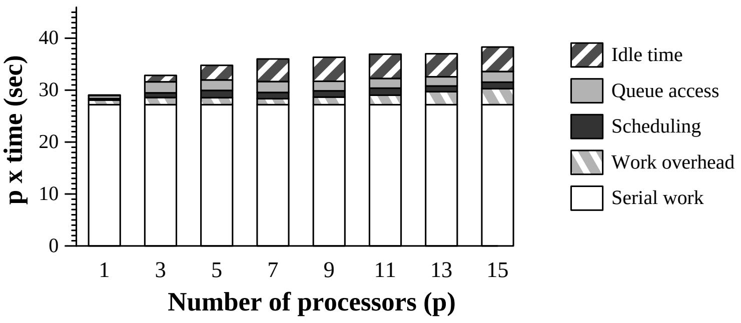

Fig. 10. The total processor time (the running time multiplied by the number of processors p) for blocked recursive matrix multiplication. "Serial work" is the time taken by a single processor executing the equivalent serial C program. For ideal speedups, all the other components would be zero. The other components are overheads of the parallel execution and the runtime system. "Idle time" is the total time spent waiting for threads to appear in Q_{out}; "queue access" is the total time spent by the worker processors inserting threads into Q_{in} and removing them from the Q_{out}. "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.

time spent by workers waiting for threads to appear in Q_{out} increases as the number of processors increases (because we use a serial scheduler). A parallel implementation of the scheduler, such as the one described in Section 6, will be more efficient on a larger number of processors.

7.4 Space Performance

Figure 11 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 [Chow and W. L. Harrison 1990; Hummel and Schonberg 1991], 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. A number of previous techniques [Burton 1988; Burton and Simpson 1994; Goldstein et al. 1995; Halbherr et al. 1994; Mohr et al. 1991; Nikhil 1994; Vandevoorde and Roberts 1988] use a strategy similar to that of Cilk. Our results show that when big allocations are delayed with dummy

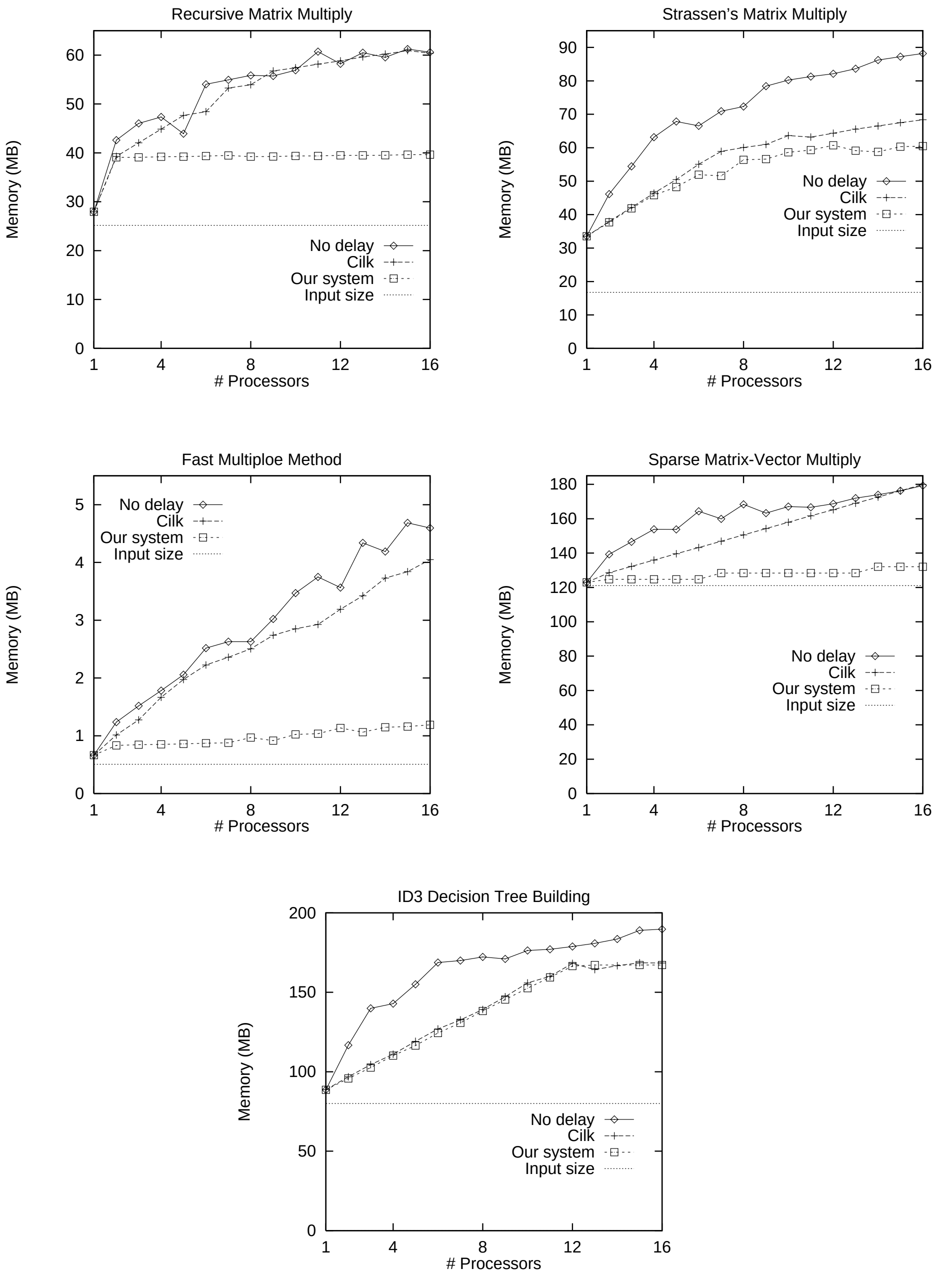

Fig. 11. The memory requirements of the parallel programs. For $p = 1$ the memory usage shown is for the serial C version. We compare the memory usage of each program when the big memory allocations are delayed by inserting dummy threads (using $K = 1000$), with when they are allowed to proceed without any delay (as an approximation of previous schedulers), as well as with the memory usage on Cilk.

nodes, our algorithm results in a significantly lower memory usage, particularly as the number of processors increases. A notable exception is the ID3 benchmark, for which our scheduler results in a similar space requirement as that of Cilk. This is because the value of K (1000 bytes) is too large to sufficiently delay the large allocations of space until higher-priority threads become ready. Note that we have not compared our scheduler to naive scheduling techniques, such as breadth-first schedules resulting from the use of FIFO queues, which have much higher memory requirements.

Space-Time Trade-off. The number of dummy nodes introduced before a large allocation and the frequency of thread preemptions depend on the value of the memory threshold K. Adjusting the value of K provides a trade-off between the memory usage and running time of a parallel computation. For example, Figure 1 shows how the running time and memory usage for blocked recursive matrix multiplication are affected by K. For very small K, many dummy nodes are inserted, and threads are preempted often; this results in a high running time. However, the execution order (and therefore the space requirement) of the parallel schedule is close to that of the serial schedule, leading to a low memory requirement. For very large K, very few dummy nodes are inserted, and threads are allowed to execute longer without being preempted; this leads to a high memory requirement and low running time. 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.

8. SUMMARY AND DISCUSSION

We have presented an asynchronous scheduling algorithm, *AsyncDF*, 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 article is significantly lower than space bounds on existing space-efficient systems for programs with a sufficient amount of parallelism ($D \ll S_1$). Most parallel programs, including all programs in NC [Cook 1985], fall in this category. This article analyzes the *AsyncDF* algorithm with both a serial and a parallel scheduler. We have built a low-overhead runtime system that schedules parallel threads using the 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 can be found elsewhere [Narlikar and Blelloch 1996].

We have added our scheduling technique to a native, lightweight implementation of POSIX standard threads [IEEE 1985] or Pthreads on Solaris [Powell et al. 1991]. The results indicate that the new scheduler, unlike the existing FIFO scheduler, allows irregular and dynamic parallel programs written with a very large number of Pthreads to execute efficiently [Narlikar and Blelloch 1998]. We are currently working on methods to further improve the scheduling algorithm, particularly to provide better support for fine-grained threads. At present, fine-grained iterations of innermost loops are statically grouped into fixed-size chunks. A dynamic, decreasing-size chunking scheme [Hummel et al. 1992; Kuck 1987; Tzen and Ni 1993] can be used instead. We are working on an algorithm to automatically coarsen the computations at runtime by allowing the execution order to differ to a limited extent from the 1DF-numbers, and by using ordered, per-processor queues. Preliminary results indicate that this algorithm also results in reduced scheduling contention, because the processors access separate queues for a majority of the time.

APPENDIX

A. TIGHTER BOUND ON THE SPACE REQUIREMENT

In Section 5 we showed that algorithm *AsyncDF* executes a parallel computation with depth D and serial space requirement S_1 on p processors using $S_1 + O(p \cdot D)$ space. In particular, when actions that allocate space are represented by heavy nodes, and each heavy node allocates at most K space (the value of the memory threshold), we showed that any prefix of the parallel computation has $O(p \cdot D)$ heavy premature nodes. Here D is the maximum number of actions along any path in the program DAG. Therefore, the value of D and the number of premature nodes (and hence the space bound) depends on the definition of an action; recall that an action is a "unit" of work that may allocate or deallocate space, and requires a timestep to be executed. An action may be as small as a fraction of a machine instruction, or as large as several machine instructions, depending on the definition of a timestep. In this section, we give a more precise space bound by specifying the bound in terms of a ratio of the depth D, and the number of actions between consecutive heavy nodes. Being a ratio of two values specified in terms of actions, it is no longer dependent on the granularity of an action.

Recall that heavy nodes may allocate K space and use it for subsequent actions, until the thread runs out of space and needs to perform another allocation. Thus, threads typically have heavy nodes followed by a large number of light nodes, and the number of allocations (heavy nodes) along any path may be much smaller than the depth of the computation. We define the *granularity* g of the computation to be the minimum number of actions (nodes) between two consecutive heavy nodes on any path in the program DAG, that is, the minimum number of actions executed nonpreemptively by a thread every time it is scheduled. Note that the granularity of a computation depends on the value of the memory threshold K. In this section, we prove that the number of heavy premature nodes is $O(p \cdot D/g)$, and therefore the parallel space requirement is $S_1 + O(p \cdot D/g)$.

LEMMA A.1. *Let G be a DAG with W nodes, depth D, and granularity g, in which every node allocates at most K space. Let s_1 be the 1DF-schedule for G, and s_p the parallel schedule for G executed by the AsyncDF 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 $O(p \cdot D/g)$.*

PROOF. The proof is similar to the proof for Lemma 5.1.1. Consider an arbitrary prefix σ_p of s_p, and let σ_1 be the corresponding prefix of s_1. As with Lemma 5.1.1, we pick a path P from the root to the last nonpremature node v to be executed in σ_p, such that for every edge (u, u') along the path, u is the last parent of u' to be executed. Let u_i be the *ith heavy* node along P; let δ be the number of heavy nodes on P. Let t_i be the timestep in which u_i gets executed; let $t_{\delta+1}$ be the last timestep in σ_p. For $i = 1, \ldots, \delta$, let I_i be the interval $\{t_i + 1, \ldots, t_{i+1}\}$.

Consider any Interval I_i, for $i = 1, \ldots, \delta - 1$. Let l_i be the number of nodes between u_i and u_{i+1}. Since all these nodes are light nodes, they get executed at timesteps $t_i + 1, \ldots, t_i + l_i$. During these timesteps, the other $p - 1$ worker processors may execute heavy nodes; however, for each heavy node, they must execute at least $(g - 1)$ light nodes. Therefore, each of these worker processors may execute at most $\lceil l_i/g \rceil$ heavy premature nodes during the first l_i timesteps of interval I_i. At the end of timestep $t_i + l_i$, there may be at most p nodes in Q_{out}. Before the thread τ containing u_i is inserted into Q_{in}, at most another $O(p)$ heavy premature nodes may be added to Q_{out}. Further, $(p - 1)$ heavy premature nodes may execute along with u_{i+1}. Hence $O(p) < c \cdot p$ heavy premature nodes (for some constant c) may be executed before or with u_{i+1} after timestep $t_i + l_i$. Thus, a total of $((p - 1) \cdot \lceil l_i/g \rceil + c \cdot p)$ heavy premature nodes get executed in the interval I_i. Similarly, we can bound the number of heavy premature nodes executed in the last interval I_δ to $((p - 1) \cdot \lceil l_\delta/g \rceil + c \cdot p)$.

Because there are $\delta \le D/g$ such intervals, and since $\sum_{i=1}^{\delta} l_i \le D$, the total number of heavy premature nodes executed over all the δ intervals is at most

$$\begin{aligned}\sum_{i=1}^{\delta}\left((p-1)\cdot\left\lceil\frac{l_i}{g}\right\rceil + c\cdot p\right) &\le \sum_{i=1}^{\delta}(p\cdot(l_i/g+1)+c\cdot p)\\ &\le (c+1)p\cdot D/g + p/g\cdot\sum_{i=1}^{\delta} l_i\\ &= O(p\cdot D/g).\end{aligned}$$

□

Similarly, we can prove that the scheduling queues require $O(p \cdot D/g)$ space; therefore, the computation requires a total of $S_1 + O(p \cdot D/g)$ space to execute on p processors.

Now consider a computation in which individual nodes allocate greater than K space. Let g_1 be the original granularity of the DAG (by simply treating the nodes that perform large allocations as heavy nodes). Now the granularity of this DAG may change when we add dummy nodes before large allocations. Let g_2 be the number of actions associated with the creation and execution of each of the dummy threads that we add to the DAG. Then the granularity of the transformed DAG is $g = \min(g_1, g_2)$. The space bound using the parallelized scheduler can be similarly modified to $S_1 + O(D \cdot p \cdot \log p/g)$.

Besides making the space bound independent of the definition of an action, this modified bound is significantly lower than the original bound for programs with high granularity g, that is, programs that perform a large number of actions between allocations, forks, or synchronizations.

B. PROOFS FOR THE SPACE AND TIME BOUNDS USING THE PARALLELIZED SCHEDULER

In this section, we prove the space and time bounds for a parallel computation executed using the parallelized scheduler, as stated in Section 6 (Theorem 6.2.1). These bounds include the space and time overheads of the scheduler. We first define a class of DAGs that are more general than the DAGs used to represent parallel computations so far. This class of DAGs will be used to represent the computation that is executed by using the parallelized scheduler.

Latency-Weighted DAGs. We extend the definition of a program DAG by allowing nonnegative weights on the edges; we call this new DAG a *latency-weighted DAG*. Let $G = (V, E)$ be a latency-weighted DAG representing a parallel computation. Each edge $(u, v) \in E$ has a nonnegative weight $l(u, v)$ which represents the *latency* between the actions of the nodes u and v. The *latency-weighted length* of a path in G is the sum of the total number of nodes in the path *plus* the sum of the latencies on the edges along the path. We define *latency-weighted depth* D_l of G to be the maximum over the latency-weighted lengths of all paths in G. Since all latencies are nonnegative, $D_l \geq D$. The program DAG described in Section 3 is a special case of a latency-weighted DAG, in which the latencies on all the edges are zero. We will use nonzero latencies to model the delays caused by the parallelized scheduler.

Let $t_e(v)$ be the timestep in which a node $v \in V$ gets executed. Then v becomes *ready* at a timestep $i \leq t_e(v)$ such that $i = \max_{(u,v)\in E}(t_e(u) + l(u, v) + 1)$. Thus, a p-schedule $V_1, V_2, \ldots, V_T$ for a latency-weighted DAG must obey the latencies on the edges, that is, $\forall (u, v) \in E$, $u \in V_j$, and $v \in V_i \Rightarrow i > j + l(u, v)$. We can now bound the time required to execute a greedy schedule for latency-weighted DAGs; our proof uses an approach similar to that used by Blumofe and Leiserson [1993] for DAGs without latencies.

Lemma B.1. *Given a latency-weighted computation graph G with W nodes and latency-weighted depth D_l, any greedy p-schedule of G will require at most $W/p + D_l$ timesteps.*

Proof. We transform G into a DAG G' without latencies by replacing each edge (u, v) with a chain of $l(u, v)$ *dummy nodes*. The dummy nodes do not represent real work, but require a timestep to be executed. Any dummy node that becomes ready at the end of timestep $t-1$ is automatically executed in timestep t. Therefore, replacing each edge (u, v) with $l(u, v)$ dummy nodes imposes the required condition that v becomes ready $l(u, v)$ timesteps after u is executed. The depth of G' is D_l.

Consider any greedy p-schedule $s_p = (V_1, \ldots, V_T)$ of G. s_p can be converted to a schedule $s'_p = (V'_1, \ldots, V'_T)$ of G' by adding (executing) dummy nodes as soon as they become ready. Thus, for $i = 1, \ldots, T$, V'_i may contain at most p real nodes, and an arbitrary number of dummy nodes, because dummy nodes do not require processors to be executed. A real node in s'_p becomes ready at the same timestep as it does in s_p, since dummy nodes now represent the latencies on the edges. Therefore, s'_p is also a "greedy" p-schedule for G', that is, at a timestep when n real nodes are ready, $\min(n, p)$ of them get executed, and all dummy nodes ready in that timestep get executed.

We now prove that any greedy p-schedule s'_p of G' will require at most $W/p + D_l$ timesteps to execute. Let G'_i denote the subgraph of G' containing nodes that have not yet been executed at the beginning of timestep i; then $G'_1 = G'$. Let n_i be the number of real nodes (not including dummy nodes) executed in timestep i. Since s'_p is a p-schedule, $n_i \le p$. If $n_i = p$, there can be at most W/p such timesteps, because there are W real nodes in the graph. If $n_i < p$, consider the set of nodes R_i that are ready at the beginning of timestep i, that is, the set of root nodes in G'_i. Since this is a greedy schedule, there are less than p real nodes in R_i. Hence all the real nodes in R_i get executed in timestep i. In addition, all the dummy nodes in R_i get executed in this step, because they are ready, and do not require processors. Since all the nodes in R_i have been executed, the depth of G'_{i+1} is one less than the depth of G'_i. Because D_l is the depth of G'_1, there are at most D_l such timesteps. Thus, s'_p (and hence s_p) can require at most $W/p + D_l$ timesteps to execute. □

With the parallelized scheduler, we consider a thread (or its leading heavy node v) to become *ready* when all the parents of v have been executed *and* the scheduler has made v available for scheduling. Since this may require a scheduling iteration after the parents of v have been executed and inserted into Q_{in}, the cost of this iteration imposes latencies on edges into v, resulting in a latency-weighted DAG. We now characterize the latency-weighted DAG generated by the parallelized scheduler. The constant-time accesses to the queues Q_{out} and Q_{in} are represented as additional actions in this DAG, while the cost of the scheduling iterations are represented by the latency-weighted edges. As with the serial scheduler in Section 5, we assume for now that every action allocates at most K space (where K is the user-specified, constant memory threshold), and we deal with larger allocations later.

LEMMA B.2. *Consider a parallel computation with work W and depth D, in which every action allocates at most K space. Using the parallelized scheduler with αp processors acting as schedulers, the remaining $(1-\alpha)p$ worker processors execute a latency-weighted DAG with $O(W)$ work, $O(D)$ depth, and a latency-weighted depth of $O(\frac{W}{\alpha p} + \frac{D \cdot \log p}{\alpha})$. Further, after the last parent of a node in the DAG is executed, at most one iteration may complete before the node becomes ready.*

PROOF. Let G be the resulting DAG executed by the worker processors. Each thread is executed nonpreemptively as long as it does not terminate or suspend, and does not need to allocate more than a net of K units of memory. Each time a thread is scheduled and then preempted or suspended, a processor performs two constant-time accesses to the queues Q_{out} and Q_{in}. As shown in Figure 12, we represent these accesses as a series of a constant number of actions (nodes) added to the thread; these nodes are added both before a heavy node (to model the delay while accessing Q_{out}) and after the series of light nodes that follow the heavy node (to model the delay while accessing Q_{in}). We will now consider the first of these added nodes to be the heavy node, instead of the real heavy node that allocates space; this gives us a conservative bound on the space requirement of the parallel computation, because we are assuming that the memory allocation has moved to an earlier time. A thread executes at least one action from the original computation every time it is scheduled. Since the original computation has W nodes, the total work performed by the worker processors is $O(W)$, that is, the resulting DAG has

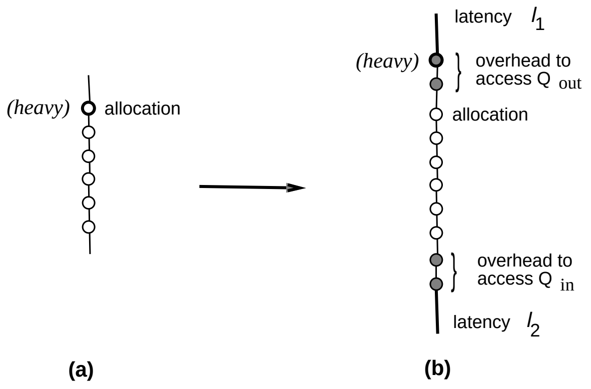

Fig. 12. (a) A portion of the original computation DAG and (b) the corresponding portion of the DAG as executed by the parallelized scheduler. This portion is the sequence of nodes in a thread executed nonpreemptively on a processor, and therefore consists of a heavy node followed by a series of light nodes. The DAG executed by the parallelized scheduler has latencies l_1 and l_2 imposed by scheduling iterations (shown as bold edges here), while the additional gray nodes represent the constant delay to access Q_{in} and Q_{out}. We consider the first of these gray nodes to be a heavy node, instead of the original heavy node that performs the real allocation.

$O(W)$ work; similarly, its depth is $O(D)$.

Next, we show that at most one scheduling iteration begins or completes after a node is executed and before its child becomes ready. Consider any thread τ in G, and let v be a node in the thread. Let t be the timestep in which the last parent u of v is completed. If v is a light node, it is executed in the next timestep. Else, the thread containing u is placed in Q_{in} at timestep t. (Recall that we have already added nodes such as u to represent the access overhead for Q_{in}.) In the worst case, a scheduling iteration may be in progress. However, the next scheduling iteration must find u in Q_{in}; this scheduling iteration moves u to $\mathcal{R}$ and makes v ready to be scheduled before the iteration completes.

Finally, we show that G has a latency-weighted depth of $O(\frac{W}{\alpha p}+\frac{D\cdot\log p}{\alpha})$. Consider any path in G. Let l be its length. For any edge $e = (u, v)$ along the path, if u is the last parent of v, we just showed that v becomes ready by the end of at most two scheduling iterations after u is executed. Therefore the latency $l(u, v)$ is at most the duration of these two scheduling iterations. Let n and n' be the number of dead threads deleted by these two scheduling iterations, respectively. Then, using Lemma 6.1.1, $l(u, v) = O(\frac{n}{\alpha p} + \frac{n'}{\alpha p} + \frac{\log p}{\alpha})$. Because each thread is deleted by the scheduler at most once, a total of $O(W)$ deletions take place. Since any path in G has $O(D)$ edges, the latency weighted depth of the path is $O(D)$ plus the sum of the latencies on $O(D)$ edges, which is $O(\frac{W}{\alpha p} + \frac{D\cdot\log p}{\alpha})$. □

The schedule generated by the parallelized scheduler for the latency-weighted DAG is a $(1-\alpha)p$-schedule, because it is executed on $(1-\alpha)p$ worker processors.

B.1 Time bound

We can now bound the total running time of the resulting schedule.

LEMMA B.3. *Consider a parallel computation with depth D and work W. For any $0 < \alpha < 1$, when αp of the processors are dedicated to execute as schedulers, while the remaining act as worker processors, the parallel computation is executed in $O(\frac{W}{\alpha(1-\alpha)p} + \frac{D \cdot \log p}{\alpha})$ time.*

PROOF. Let G be the DAG executed by the parallelized scheduler for this computation. We will show that the generated schedule s_p of G is a greedy schedule, with $O(W/p)$ additional timesteps in which the worker processors may be idle. Consider any scheduling iteration. Let t_i be the timestep at which the *ith* scheduling iteration ends. After threads are inserted into Q_{out} by the *ith* scheduling iteration, there are two possibilities:

(1) $|Q_{out}| < p \log p$. This implies that all the ready threads are in Q_{out}, and no threads become ready until the end of the next scheduling iteration. Therefore, at every timestep j such that $t_i < j \leq t_{i+1}$, if m_j processors become idle and r_j threads are ready, $\min(m_j, r_j)$ threads are scheduled on the processors. (Recall that we have already added nodes to the DAG G to model the overheads of accessing Q_{in} and Q_{out}.)

(2) $|Q_{out}| = p \log p$. Since $(1-\alpha)p$ worker processors will require at least $\frac{\log p}{(1-\alpha)}$ timesteps to execute $p \log p$ actions, none of the worker processors will be idle for the first $\frac{\log p}{(1-\alpha)}$ steps after t_i. However, if the $(i+1)th$ scheduling iteration, which is currently executing, has to delete n_{i+1} dead threads, it may execute for $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ timesteps (using Lemma 6.1.1). Thus, in the worst case, the processors will be busy for $\frac{\log p}{(1-\alpha)}$ steps and then remain idle for another $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ steps, until the next scheduling iteration ends. We call such timesteps *idling* timesteps. Of the $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ idling steps, $\Theta(\frac{\log p}{\alpha})$ steps are within a factor of $\frac{c(1-\alpha)}{\alpha}$ of the preceding $\frac{\log p}{(1-\alpha)}$ steps when all worker processors were busy (for some constant c); therefore, they can add up to $O(\frac{W}{p} \cdot \frac{(1-\alpha)}{\alpha}) = O(\frac{W}{\alpha p})$. In addition, because each thread is deleted only once, at most W threads can be deleted. Therefore, if the $(i+1)th$ scheduling iteration results in an additional $O(\frac{n_{i+1}}{\alpha p})$ idle steps, they add up to $O(\frac{W}{\alpha p})$ idle steps over all the scheduling iterations. Therefore, a total of $O(\frac{W}{\alpha p})$ idling steps can result due to the scheduler.

All timesteps besides the idling steps caused by the scheduler obey the conditions required to make it a greedy $(1-\alpha)p$-schedule, and therefore add up to $O(\frac{W}{(1-\alpha)p} + \frac{W}{\alpha p} + \frac{D \log p}{\alpha})$ (using Lemmas B.1 and B.2). Along with the additional $O(\frac{W}{\alpha p})$ idling steps, the schedule requires a total of $O(\frac{W}{\alpha(1-\alpha)p} + \frac{D \cdot \log p}{\alpha})$ timesteps. ☐

Because α is a constant, that is, a constant fraction of the processors are dedicated to the task of scheduling, the running time reduces to $O(W/p + D \log p)$; here p is the total number of processors, including both the schedulers and the workers.

B.2 Space bound

We now show that the total space requirement of the parallel schedule exceeds the serial schedule by $O(D \cdot p \cdot \log p)$. We first bound the number of premature nodes that may exist in any prefix of the parallel schedule, and then bound the space required to store threads in the three scheduling queues.

For a parallel computation with depth D, the parallelized scheduler executes a DAG of depth $O(D)$ (using Lemma B.2). Therefore, using an approach similar to that of Lemma 5.1.1, we can prove the following bound for the parallelized scheduler.

LEMMA B.4. *For a parallel computation with depth D executing on p processors, the number of premature nodes in any prefix of the schedule generated by the parallelized scheduler is $O(D \cdot p \cdot \log p)$.* □

LEMMA B.5. *The total space required for storing threads in Q_{in}, Q_{out}, and $\mathcal{R}$ while executing a parallel computation of depth D on p processors is $O(D \cdot p \cdot \log p)$.*

PROOF. Q_{out} may hold at most $p \log p$ threads at any time. Similarly, Q_{in} may hold at most $2p \cdot \log p + (1 - \alpha)p$ threads, which is the maximum number of active threads. Each thread can be represented using a constant amount of space. Therefore the space required for Q_{in} and Q_{out} is $O(p \cdot \log p)$.

We now bound the space required for $\mathcal{R}$. Recall that $\mathcal{R}$ consists of live threads and dead threads. Consider any prefix σ_p of the parallel schedule s_p. σ_p may have at most $O(D \cdot p \cdot \log p)$ heavy premature nodes (using Lemma B.4). We call a thread a *premature thread* if at least one of its heavy nodes that was scheduled or put on Q_{out} is premature. The number of live threads is at most the number of premature threads, plus the number of stubs (which is $O(p \cdot \log p)$), plus the number of live threads that are not premature (which is bounded by the maximum number of live threads in the serial schedule). A 1DF-schedule may have at most D live threads at any timestep. Therefore, the total number of live threads at any timestep is at most $O(D \cdot p \cdot \log p)$.

The scheduler performs lazy deletions of dead threads; therefore, the number of dead threads in $\mathcal{R}$ must also be counted. Let λ_i be the *ith* scheduling iteration that schedules at least one thread. Consider any such iteration λ_i. Recall that this iteration must delete at least all the dead threads up to the second ready or seed thread in $\mathcal{R}$. We will show that after scheduling iteration λ_i performs deletions, all remaining dead threads in $\mathcal{R}$ must be premature. Let τ_1 and τ_2 be the first two ready (or seed) threads in $\mathcal{R}$. Since the scheduler deletes all dead threads up to τ_2, there can be no more dead threads to the immediate right of τ_1 that may have a lower 1DF-number than τ_1, that is, τ_1 now has a lower 1DF-number number than all the dead threads in $\mathcal{R}$. Because all the remaining dead threads have been executed before the ready thread τ_1 (or the ready threads represented by τ_1), they must be premature. Therefore, all dead threads in $\mathcal{R}$ at the end of scheduling iteration λ_i must be premature, and are therefore $O(D \cdot p \cdot \log p)$ in number (using Lemma B.4). The scheduling iterations between λ_i and λ_{i+1} do not schedule any threads, and therefore do not create any new entries in $\mathcal{R}$. They may, however, mark existing live threads as dead. Thus, the number of dead threads in $\mathcal{R}$ may increase, but the total number of threads in $\mathcal{R}$ remains the same. Scheduling iteration λ_{i+1} must delete all dead threads up to the second ready thread in $\mathcal{R}$. Therefore, before it creates any new threads, the iteration reduces the number of dead threads back to $O(D \cdot p \cdot \log p)$. Thus, at any time, the total space required for $\mathcal{R}$ is $O(D \cdot p \cdot \log p)$.
□

Since every premature node may allocate at most K space, we can now state the following space bound using Lemmas B.4 and B.5.

LEMMA B.6. *A parallel computation with work W and depth D, in which every node allocates at most K space, and which requires S_1 space to execute on one processor, can be executed on p processors in $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space (including scheduler space) using the parallelized scheduler.* □

As in Section 5, allocations larger than K units are handled by delaying the allocation with parallel dummy threads. If S_a is the total space allocated, the number of dummy nodes added is at most S_a/K, and the depth is increased by a constant factor. Therefore, using the parallelized scheduler, the final time bound of $O(W/p + S_a/pK + D \cdot \log p)$ and the space bound of $S_1 + O(K \cdot D \cdot p \cdot \log p)$ follow, as stated in Theorem 6.2.1. These bounds include the scheduler overheads.

ACKNOWLEDGMENTS

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

ARVIND, NIKHIL, R. S., AND PINGALI, K. 1989. I-structures: Data structures for parallel computing. *ACM Trans. on Programm. Lang. Syst. 11,* 4 (Oct.), 598–632.

BERSHAD, B. N., LAZOWSKA, E., AND LEVY, H. 1988. PRESTO : A system for object-oriented parallel programming. *Soft. Pract. and Exper. 18,* 8 (Aug.), 713–732.

BLELLOCH, G. E., CHATTERJEE, S., HARDWICK, J. C., SIPELSTEIN, J., AND ZAGHA, M. 1994. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing 21,* 1 (April), 4–14.

BLELLOCH, G. E., GIBBONS, P. B., AND MATIAS, Y. 1995. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. Symposium on Parallel Algorithms and Architectures*, Santa Barbara, pp. 420–430.

BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. 1996. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proc. Symp. on Parallel Algorithms and Architectures*, pp. 297–308.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proc. Symposium on Principles and Practice of Parallel Programming*, pp. 207–216.

BLUMOFE, R. D. AND LEISERSON, C. E. 1993. Space-efficient scheduling of multithreaded computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pp. 362–371.

BLUMOFE, R. D. AND LEISERSON, C. E. 1994. Scheduling multithreaded computations by work stealing. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pp. 356–368.

BURTON, F. W. 1988. Storage management in virtual tree machines. *IEEE Trans. on Computers 37,* 3, 321–328.

BURTON, F. W. AND SIMPSON, D. J. 1994. Space efficient execution of deterministic parallel programs. Manuscript.

BURTON, F. W. AND SLEEP, M. R. 1981. Executing functional programs on a virtual tree of processors. In *Conference on Functional Programming Languages and Computer Architecture.*

CHANDRA, R., GUPTA, A., AND HENNESSY, J. 1994. COOL: An object-based language for parallel programming. *IEEE Computer 27,* 8 (Aug.), 13–26.

CHANDY, K. M. AND KESSELMAN, C. 1992. Compositional c++: compositional parallel programming. In *Proc. 5th. Intl. Wkshp. on Languages and Compilers for Parallel Computing*, New Haven, CT, pp. 124–144.

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

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

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

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

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

Freeh, V. W., Lowenthal, D. K., and Andrews, G. R. 1994. Distributed filaments: efficient fine-grain parallelism on a cluster of workstations. In *1st Symposium on Operating Systems Design and Implementation*, Monterey, CA, pp. 201–212.

Goldstein, S. C., Culler, D. E., and Schauser, K. E. 1995. Enabling primitives for compiling parallel languages. In *3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Rochester, NY.

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

Halbherr, M., Zhou, Y., and Joerg, C. F. 1994. Parallel programming based on continuation-passing thread. In *Proc. 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, Capri, Italy.

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

HPF Forum 1993. High Performance Fortran language specification, Version 1.0.

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

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

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

IEEE 1985. Threads extension for portable operating systems (draft 6).

Kuck, C. D. P. D. 1987. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers C-36,* 12 (Dec.), 1425–39.

Mellor-Crummey, J. M. 1987. Concurrent queues: Practical Fetch-and-Φ algorithms. Technical Report 229 (Nov.), University of Rochester.

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

Mohr, E., Kranz, D., and Halstead, R. 1991. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. on Parallel and Distributed Systems 2,* 3 (July), 264–280.

Mueller, F. 1993. A library implementation of POSIX threads under unix. In *Proc. Winter 1993 USENIX Technical Conference and Exhibition*, San Diego, CA, USA, pp. 29–41.

Narlikar, G. J. 1999. Space-efficient multithreading. School of Computer Science, Carnegie Mellon University. Ph.D. thesis, to appear.

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

Narlikar, G. J. and Blelloch, G. E. 1998. Pthreads for dynamic and irregular parallelism. In *Proc. SC98: High Performance Networking and Computing*, Orlando, FL. IEEE.

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

POWELL, M. L., KLEIMAN, S. R., BARTON, S., SHAH, D., STEIN, D., AND WEEKS, M. 1991. SunOS multi-thread architecture. In USENIX ASSOCIATION (Ed.), *Proc. Winter 1991 USENIX Conference: Dallas, TX, USA*, pp. 65–80.

QUINLAN, J. R. 1986. Induction of decision trees. *Machine learning 1,* 1, 81–106.

RINARD, M. C., SCALES, D. J., AND LAM, M. S. 1993. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer 26,* 6 (June), 28–38.

ROGERS, A., CARLISLE, M., REPPY, J., AND HENDREN, L. 1995. Supporting dynamic data structures on distributed memory machines. *ACM Trans. on Programm. Lang. Syst. 17,* 2 (March), 233–263.

RUGGUERO, C. A. AND SARGEANT, J. 1987. Control of parallelism in the manchester dataflow machine. In *Functional Programming Languages and Computer Architecture*, Volume 174 of *Lecture Notes in Computer Science*, pp. 1–15. Springer-Verlag.

STRASSEN, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik 13*, 354–356.

TZEN, T. H. AND NI, L. M. 1993. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Trans. on Parallel and Distributed Systems 4,* 1 (Jan.), 87–98.

VANDEVOORDE, M. T. AND ROBERTS, E. S. 1988. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming 17,* 4 (Aug.), 347–366.

Received May 1998; accepted November 1998