Scheduling Threads for Low Space Requirement and Good Locality

Girija J. Narlikar

Bell Laboratories, Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
girija@research.bell-labs.com

Appears in the Theory of Computing Systems, Volume 35, 2002.

Keywords: Multithreading, space efficiency, work stealing, dynamic scheduling, nested parallelism, data locality, dynamic dags.

Abstract

The running time and memory requirement of a parallel program with dynamic, lightweight threads depends heavily on the underlying thread scheduler. In this paper, we present a new asynchronous, space-efficient scheduling algorithm for shared memory machines that combines the low scheduling overheads and good locality of work stealing with the low space requirements of depth-first schedulers. For a nested-parallel program with depth D and serial space requirement S_1, we show that the expected space requirement is $S_1 + O(K \cdot p \cdot D)$ on p processors. Here, K is a user-adjustable runtime parameter, which provides a trade-off between running time and space requirement. Our algorithm achieves good locality and low scheduling overheads by automatically increasing the granularity of the work scheduled on each processor.

We have implemented the new scheduling algorithm in the context of a native, user-level implementation of Posix standard threads or Pthreads, and evaluated its performance using a set of C-based benchmarks that have dynamic or irregular parallelism. We compare the performance of our scheduler with that of two previous schedulers: the thread library's original scheduler (which uses a FIFO queue), and a provably space-efficient depth-first scheduler. At a fine thread granularity, our scheduler outperforms both these previous schedulers, but requires marginally more memory than the depth-first scheduler.

We also present simulation results on synthetic benchmarks to compare our scheduler with space-efficient versions of both a work-stealing scheduler and a depth-first scheduler. The results indicate that unlike these previous approaches, the new algorithm covers a range of scheduling granularities and space requirements, and allows the user to trade the space requirement of a program with the scheduling granularity.

1 Introduction

Many parallel programming languages allow the expression of dynamic, lightweight threads. These include data parallel languages like HPF [29] or Nesl [9] (where the sequence of instructions executed over individual data elements are the "threads"), dataflow languages like ID [21], control-parallel languages with fork-join constructs like Cilk [26], CC++ [17], and Proteus [36], languages with futures like Multilisp [49], and various user-level thread libraries [7, 22, 37, 53]. In the lightweight threads model, the programmer simply expresses all the parallelism in the program, while the language implementation performs the task of scheduling the threads onto the processors at runtime. Thus the advantages of lightweight, user-level threads include the ease of programming, automatic load balancing, architecture-independent code that can adapt to a varying number of processors, and the flexibility to use kernel-independent thread schedulers.

Programs with irregular and dynamic parallelism benefit most from the use of lightweight threads. Compile-time analysis of such computations to partition and map the threads onto processors is generally not possible. Therefore, the programs depend heavily on the implementation of the runtime system for good performance. In particular,

1. To allow the expression of a large number of threads, the runtime system must provide fast thread operations such as creation, deletion and synchronization.
2. The thread scheduler must incur low overheads while dynamically balancing the load across all the processors.
3. The scheduling algorithm must be space efficient, that is, it must not create too many simultaneously active threads, or schedule them in an order that results in high memory allocation. A smaller memory footprint results in fewer page and TLB misses. This is particularly important for parallel programs, since they are typically used to solve large problems, and are often limited by the amount of memory available on a parallel machine. Existing commercial thread systems, however, can lead to poor space and time performance for multithreaded parallel programs, if the scheduler is not designed to be space efficient [42].
4. Today's hardware-coherent shared memory multiprocessors (SMPs) typically have a large off-chip data cache for each processor, with a latency significantly lower that the latency to main memory. Therefore, the thread scheduler must also schedule threads for good cache locality. The most common heuristic to obtain good locality for fine grained threads on multiprocessors is to schedule threads close in the computation graph (e.g., a parent thread along with its child threads) on the same processor, since they typically share common data [2, 13, 32, 34, 38, 49].

Work stealing is a runtime scheduling mechanism that can provide a fair combination of the above requirements. Each processor maintains its own queue of ready threads; a processor steals a thread from another processor's ready queue only when it runs out of ready threads in its own queue. Since thread creation and scheduling are typically local operations, they incur low overhead and contention. Further, threads close together in the computation graph are often scheduled on the same processor, resulting in good locality. Several systems have used work stealing to provide high performance [15, 22, 23, 26, 33, 49, 52, 54]. When each processor treats its own ready queue as a LIFO stack (that is, adds or removes threads from the top of the stack) and steals from the bottom of another processor's stack, the scheduler successfully throttles the excess parallelism [12, 49, 51, 54]. For fully strict computations, such a mechanism was proved to require $p \cdot S_1$ space on p processors, where S_1 is the serial, depth-first space requirement [13]. A computation with W work (total number of operations) and D depth (length of the critical path) was shown to require $W/p + O(D)$ time on p processors [13]. We will henceforth refer to such schedulers as ***work-stealing*** schedulers.

Recent work [10, 41] has resulted in ***depth-first*** scheduling algorithms that require $S_1+O(p\cdot D)$ space for nested-parallel computations with depth D. For programs that have a low depth (a high degree of parallelism), such as all programs in the class NC [18], the space bound of $S_1+O(p\cdot D)$ is asymptotically lower than the work stealing bound of $p \cdot S_1$. Further, the depth-first approach allows a more general memory allocation model compared to the stack-based allocations assumed in space-efficient work stealing [10]. The depth-first approach has been extended to handle computations with futures [49] or I-structures [21], resulting in similar space bounds [8]. Experiments showed that an asynchronous, depth-first scheduler often results in lower space requirement in practice, compared to a work-stealing scheduler [41]. However, since depth-first schedulers use a globally ordered queue, they do not provide some of the practical advantages enjoyed by work-stealing schedulers. When the threads expressed by the user are fine grained, the performance may suffer due to poor locality and high scheduling contention (i.e., contention over shared data structures while scheduling) [42]. Therefore, even if basic thread operations are cheap, the threads have to be coarsened for depth-first schedulers to provide good performance in practice.

In this paper, we present a new scheduling algorithm for implementing multithreaded languages on shared memory machines. The algorithm, called ***DFDeques***[1], provides a compromise between previous work-stealing and depth-first schedulers. Ready threads in *DFDeques* are organized in multiple ready queues, that are globally ordered as in depth-first schedulers. The ready queues are treated as LIFO stacks similar to previous work-stealing schedulers. A processor steals from a ready queue chosen randomly from a set of high-priority queues. For nested-parallel (or fully strict) computations, our algorithm guarantees an expected space bound of $S_1 + O(K \cdot p \cdot D)$. Here, K is a user-adjustable runtime parameter called the ***memory threshold***, which specifies the net amount of memory a processor may allocate between consecutive steals. Since K is typically fixed to be a small, constant amount of memory[2], the space bound reduces to $S_1 + O(D \cdot p)$, as with depth-first schedulers. For a simplistic cost model, we show that the expected running time is $O(W/p + D)$ on p processors[3].

We refer to the total number of instructions executed in a thread as the thread's ***granularity***. We also (informally) define ***scheduling granularity*** to be the average number of instructions executed consecutively on a single processor from threads close together in the computation graph. Thus, a larger scheduling granularity typically implies better locality and lower scheduling contention. In the *DFDeques* scheduler, when a processor finds its ready queue empty, it steals a thread from the *bottom* of another ready queue. This thread is typically the coarsest thread in the queue, resulting in a larger scheduling granularity compared to depth first schedulers. Although we do not analytically

[1] *DFDeques* stands for "depth-first deques".

[2] In practice, we set K to a few kilobytes (50KB in our experiments), which is small compared to the several megabytes (or gigabytes) of memory available on today's machines.

[3] When the scheduler in *DFDeques* is parallelized, the costs of all scheduling operations can be accounted for with a more realistic model [40]. Then, in the expected case, the parallel computation can be executed using $S_1 + O(D \cdot p \cdot \log p)$ space and $O(W/p + D \cdot \log p)$ time (including scheduling overheads). However, for brevity, we omit a description and analysis of such a parallelized scheduler.

Benchmark	Max threads			L2 Cache miss rate			8 processor speedup		
	FIFO	ADF	DFD	FIFO	ADF	DFD	FIFO	ADF	DFD
Vol. Rend.	436	36	37	4.2	3.0	1.8	5.39	5.99	6.96
Dense MM	3752	55	77	24.0	13	8.7	0.22	3.78	5.82
Sparse MVM	173	51	49	13.8	13.7	13.7	3.59	5.04	6.29
FFTW	510	30	33	14.6	16.4	14.4	6.02	5.96	6.38
FMM	2030	50	54	14.0	2.1	1.0	1.64	7.03	7.47
Barnes Hut	3570	42	120	19.0	3.9	2.9	0.64	6.26	6.97
Decision Tr.	194	138	149	5.8	4.9	4.6	4.83	4.85	5.39

Figure 1: Summary of experimental results with the Solaris Pthreads library. For each scheduling technique, we show the maximum number of simultaneously active threads (each of which requires min. 8kB stack space), the L2 cache misses rates (%), and the speedups on an 8-processor Enterprise 5000 SMP. "FIFO" is the original Pthreads scheduler, "ADF" is an asynchronous, depth-first scheduler [42], and "DFD" is our new *DFDeques* scheduler.

prove this claim, we present experimental and simulation results to verify it. Adjusting the memory threshold K in the *DFDeques* algorithm provides a user-controllable trade-off between scheduling granularity and space requirement.

Posix threads or Pthreads have recently become a popular standard for shared memory parallel programming. We therefore added the *DFDeques* scheduling algorithm to a native, user-level Pthreads library [53]. Despite being one of the fastest user-level implementations of Pthreads today, the library's scheduler does not efficiently support fine-grained, dynamic threads. In previous work [42], Narlikar and Blelloch showed how its performance can be improved using a space-efficient depth-first scheduler. In this paper, we compare the space and time performance of the new *DFDeques* scheduler with the library's original scheduler (which uses a FIFO scheduling queue), and with a previous implementation of a depth-first scheduler from [42]. To perform the experimental comparison, we used 7 parallel benchmarks written with a large number of dynamically created Pthreads. As shown in Figure 1, the new *DFDeques* scheduler results in better locality and higher speedups compared to both the depth-first scheduler and the FIFO scheduler.

Ideally, we would also like to compare our Pthreads-based implementation of *DFDeques* with a space-efficient work-stealing scheduler (e.g., the scheduler used in Cilk [12]). However, supporting the general Pthreads functionality (which includes various synchronization primitives) with an existing space-efficient work-stealing scheduler [12] would require significant modifications to both the scheduling algorithm and the Pthreads implementation[4]. Therefore, to compare our new scheduler to this work-stealing scheduler, we instead built a simple simulator that implements synthetic, fully-strict benchmarks. Our simulation results indicate that by adjusting the memory threshold, our new scheduler covers a wide range of space requirements and scheduling granularities. At one extreme it performs similar to a depth-first scheduler, with low space requirement and small scheduling granularity. At the other extreme, it behaves exactly like the work-stealing

[4]Even fully strict Pthreads benchmarks cannot be executed using such a work-stealing scheduler in the existing Solaris Pthreads implementation, because the Pthreads implementation itself makes extensive use of blocking synchronization primitives such as Pthread mutexes and condition variables.

scheduler, with higher space requirement and larger scheduling granularity.

2 Background and Previous Work

A parallel computation can be represented by a directed acyclic graph; we will refer to such a computation graph as a ***dag*** in the remainder of this paper. Each node in the dag represents a single ***action*** in a thread; an action is a unit of work that requires a single timestep to be executed. Each edge in the dag represents a dependence between two actions. Figure 2 shows such an example dag for a simple parallel computation. The dashed, right-to-left ***fork*** edges in the figure represent the fork of a child thread. The dashed, left-to-right ***synch*** edges represent a join between a parent and child thread, while each solid vertical ***continue*** edge represents a sequential dependence between a pair of consecutive actions within a single thread. For computations with dynamic parallelism, the dag is revealed and scheduled onto the processors at runtime.

2.1 Scheduling for locality

Detection of data accesses or data sharing patterns among threads in a dynamic and irregular computation is often beyond the scope of the compiler. Further, today's hardware-coherent SMPs do not allow explicit, software-controlled placement of data in processor caches; therefore, owner-compute optimizations for locality that are popular on distributed memory machines typically do not apply to SMPs. However, in many parallel programs with fine-grained threads, the threads close together in the computation's dag often access the same data. For example, in a divide-and-conquer computation (such as quicksort) where a new thread is forked for each recursive call, a thread shares data with all its descendent threads. Therefore, many parallel implementations of lightweight threads use per-processor data structures to store ready threads [22, 26, 31, 32, 49, 52, 54]. Threads created on a processor are stored locally and moved only when required to balance the load. This technique effectively increases scheduling granularity, and therefore provides good locality [11] and low scheduling contention.

Another approach for obtaining good locality is to allow the user to supply hints to the scheduler regarding the data access patterns of the threads [16, 35, 45, 55]. However, such hints can be cumbersome for the user to provide in complex programs, and are often specific to a certain language or library interface. Therefore, our *DFDeques* algorithm instead uses the heuristic of scheduling threads close in the dag on the same processor to obtain good locality.

2.2 Scheduling for space-efficiency

The thread scheduler plays a significant role in controlling the amount of active parallelism in a fine-grained computation. For example, consider a single-processor execution of the dag in Figure 2. If the scheduler uses a LIFO stack to store ready threads, and a child thread preempts its parent as soon as it is forked, the nodes are executed in a (left-to-right) depth-first order, resulting in at most 5 simultaneously active threads. In contrast, if the scheduler uses a FIFO queue, the threads are executed in a breadth-first order, resulting in all 13 threads being simultaneously active. Systems that support fine-grained, dynamic parallelism can suffer from such a creation of excess

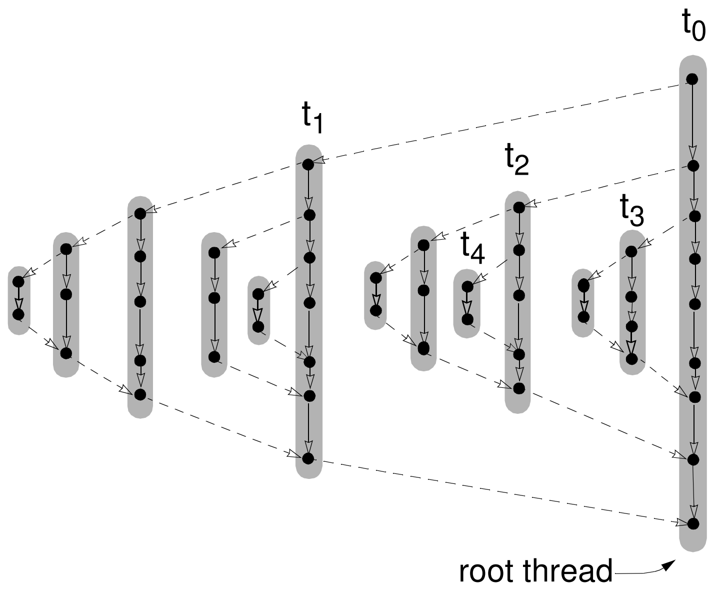


Figure 2: An example dag for a parallel computation; the threads are shown shaded. Each right-to-left edge represents a fork, and each left-to-right edge represents a synchronization of a child thread with its parent. Vertical edges represent sequential dependencies within threads. t_0 is the initial (root) thread, which forks child threads t_1, t_2, and t_3 in that order. Child threads may fork threads themselves; e.g., t_2 forks t_4.

parallelism. Limiting this excess parallelism and lowering the space requirement is critical for parallel programs, since they are often limited by the amount of memory available on a parallel machine.

Initial attempts to control the active parallelism were based on heuristics [7, 21, 38, 50, 49], which included work stealing techniques [38, 49]. Heuristic attempts work well for some programs, but do not guarantee an upper bound on the space requirements of a program. More recently, two different techniques have been shown to be provably space-efficient: work-stealing schedulers, and depth-first schedulers.

In addition to being space efficient [12, 51], work stealing can often result in large scheduling granularities, by allowing idle processors to steal threads higher up in the dag (e.g., see Figure 3(a)). Several systems use such an approach to obtain good parallel performance [12, 22, 33, 49, 54].

Depth-first schedulers guarantee an upper bound on the space requirement of a parallel computation by prioritizing its threads according to their serial, depth-first execution order [10, 41]. In a recent paper [42], Narlikar and Blelloch showed that the performance of a commercial Pthreads implementation could be improved for predominantly nested-parallel benchmarks using a depth-first scheduler. However, depth-first schedulers can result in high scheduling contention and poor locality when the threads in the program are very fine grained [41, 42]. This is because, unlike work stealing schedulers, depth-first schedulers may map threads close together in a computation graph on different processors (e.g., see Figure 3).

The next section describes a new scheduling algorithm that combines ideas from the above two space-efficient approaches.

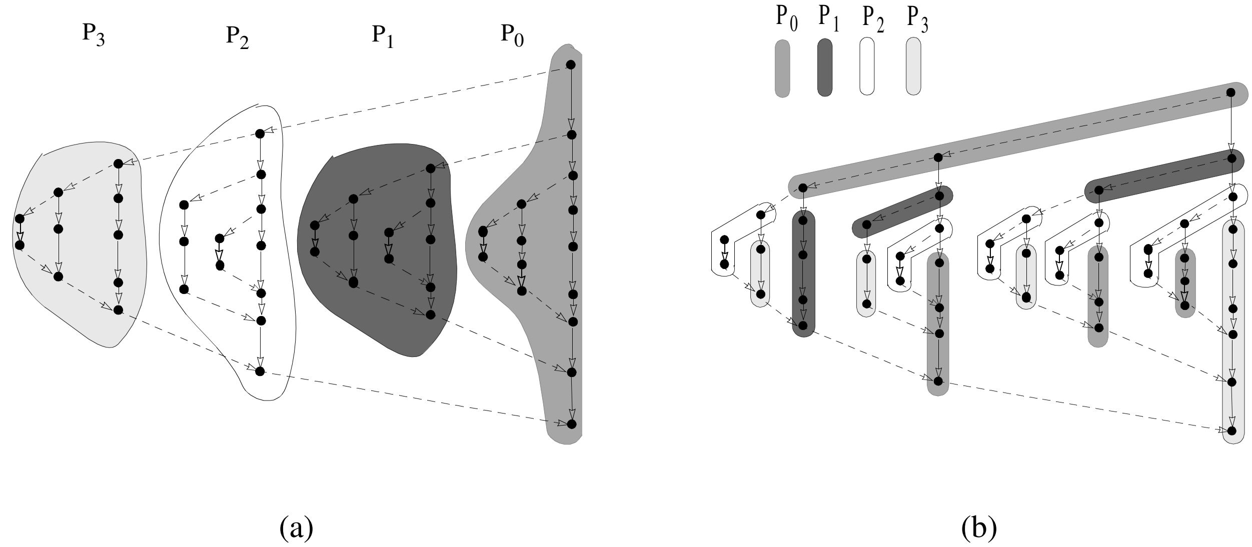


Figure 3: Possible mappings of threads of the dag in Figure 2 onto processors $P_0, \ldots, P_3$ by (a) work-stealing schedulers, and (b) depth-first schedulers. If, say, the i^{th} thread (going from left to right) accesses the i^{th} block or element of an array, then scheduling consecutive threads on the same processor provides better cache locality and lower scheduling overheads.

3 The *DFDeques* Scheduling Algorithm

We first describe the programming model for the multithreaded computations that are executed by the *DFDeques* scheduling algorithm. We then list the data structures used by the scheduler, followed by a description of the *DFDeques* scheduling algorithm.

3.1 Programming model

As with depth-first schedulers, our scheduling algorithm applies to pure, ***nested-parallel*** computations, which can be modeled by series-parallel dags [10]. Our model assumes binary forks and joins; the example dag in Figure 2 represents such a nested-parallel computation. Such nested-parallel computations are equivalent to the subset of fully strict computations that are supported by Cilk's space-efficient work-stealing scheduler [12, 26]. Nested parallelism can be used to express a large variety of parallel programs, including recursive, divide-and-conquer programs and programs with nested-parallel loops.

Although we describe and analyze our algorithm for nested-parallel computations, in practice it can be extended to execute programs with other styles of parallelism. For example, the Pthreads scheduler described in Section 5 supports computations with arbitrary synchronizations, such as mutexes and condition variables. However, our analytical space bound does not apply to such general computations.

A thread is ***active*** if it has been created but has not yet terminated. A parent thread waiting to synchronize with a child thread is said to be ***suspended***. We say an active thread is ***ready*** to be scheduled if it is not suspended, and is not currently being executed by a processor. Each action in

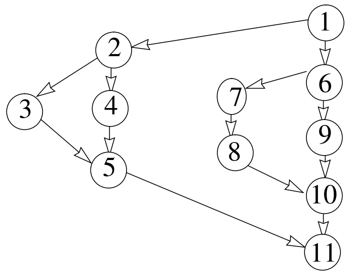


Figure 4: The serial, depth-first execution order for a nested-parallel computation. The i^{th} node executed is labelled i in this dag. The lower the label of a thread's current node (action), the higher is its priority in *DFDeques*.

a thread may allocate an arbitrary amount of space on the thread stack, or on the shared heap.

Every nested-parallel computation has a natural serial execution order, which we call its ***depth-first order***. When a child thread is forked, it is executed before its parent in a depth-first execution (e.g., see Figure 4). Thus, the depth-first order is identical to the unique serial execution order for any stack-based language (such as C), when the thread forks are replaced by simple function calls. Algorithm *DFDeques* prioritizes ready threads according to their serial, depth-first execution order; an earlier serial execution order translates to a higher priority.

3.2 Scheduling data structures

Although the dag for a computation is revealed as the execution proceeds, dynamically maintaining the relative thread priorities for nested-parallel computations is straightforward [10] and inexpensive in practice [41]. In algorithm *DFDeques*, the ready threads are stored in doubly-ended queues or ***deques*** [20]. Each of these deques supports popping from and pushing onto its top, as well as popping from the bottom of the deque. At any time during the execution, a processor ***owns*** at most one deque, and executes threads from it. A single deque has at most one owner at any time. However, unlike traditional work stealing, the number of deques is not limited, and may exceed the number of processors. All the deques are arranged in a global list $\mathcal{R}$ of deques. The list supports adding of a new deque to the immediate right of another deque, deletion of a deque, and finding the m^{th} dequeue from the left end of $\mathcal{R}$.

3.3 The *DFDeques* scheduling algorithm

The processors execute the code in Figure 5 for algorithm *DFDeques*(K); here K is the ***memory threshold***, a user-defined runtime parameter. Each processor treats its own deque as a regular LIFO stack, and is assigned a memory quota of K bytes from which to allocate heap and stack data. This memory threshold K is equivalent to the per-thread memory quota in depth-first schedulers [41]; however, in algorithm *DFDeques*, the memory quota of K bytes can be used by a processor to execute multiple threads from one deque.

The execution starts with a single deque in the system, containing the initial (root) thread. A thread executes without preemption on a processor until it forks a child thread, suspends waiting for a child to terminate, terminates, or the processor runs out of its memory quota. If a terminating

```
while (∃ threads)
   if (currS = NULL) currS := steal();                  # perform steal if no current stack
   if (currT = NULL) currT := pop_from_top(currS);      # get new thread from top of current stack
   execute currT until it forks, suspends,
      terminates, or memory quota exhausted:
      case (fork):
         push_to_top(currT, currS);                     # place current thread on top of current stack
         currT := newly forked child thread;            # begin executing newly forked child thread
      case (suspend):
         currT := NULL;                                 # give up current thread; it will be woken later
      case (memory quota exhausted):
         push_to_top(currT, currS);                     # place current thread on top of current stack
         currT := NULL;                                 # give up current thread
         currS := NULL;                                 # give up current stack
      case (terminate):
         if currT wakes up suspended parent T′
            currT := T′;                                # begin executing newly woken parent thread
         else currT := NULL;                            # give up current thread
   if ((is_empty(currS)) and (currT= NULL))
      currS := NULL;                                    # give up and delete current (empty) stack
endwhile                                                # repeat until end of parallel computation

procedure steal():                                      # returns a new deque with the stolen thread in it
set memory quota to K;
while (∃ threads)
   m := random number in [1 ... p];
   S := m^th deque in R;                                # pick deque to steal from
   T := pop_from_bot(S);                                # attempt to steal a thread
   if (T ≠ NULL)                                        # attempt succeeded
      create new deque S′ containing T
         and become its owner;
      place S′ to immediate right of S in R;
      return S′;
endwhile                                                # repeat until steal is successful or computation ends
```

Figure 5: Pseudocode for the *DFDeques*(K) scheduling algorithm executed by each of the p processors; K is the memory threshold. *currS* is the processor's current deque. *currT* is the current thread being executed; changing its value denotes a context switch. Memory management of the deques is not shown here for brevity.

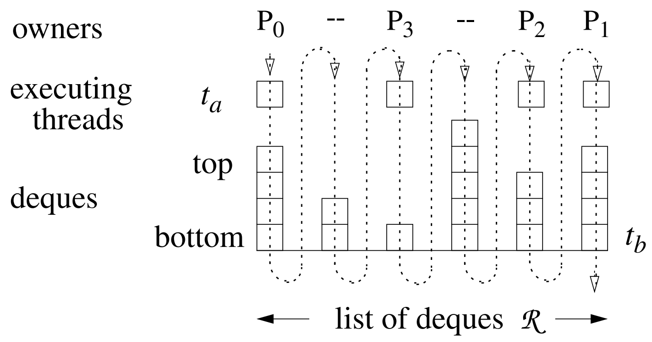


Figure 6: The list $\mathcal{R}$ of deques maintained in the system by algorithm *DFDeques*. Each deque may have one (or no) owner processor. The dotted line traces the decreasing order of priorities of the threads in the system; thus t_a in this figure has the highest priority, while t_b has the lowest priority.

thread wakes up its previously suspended parent, the processor starts executing the parent next; for nested parallel computations, we can show that the processor's deque must be empty at this stage [40]. When an idle processor finds its deque empty, it deletes the deque. When a processor deletes its deque, or when it gives up ownership of its deque due to exhaustion of its memory quota, it uses the `steal()` procedure to obtain a new deque. The main difference from previous depth-first schedulers [10, 41] is in this steal procedure. Every invocation of `steal()` resets the processor's memory quota to K bytes. We call an iteration of the loop in the `steal()` procedure a ***steal attempt***.

A processor executes a steal attempt by picking a random number m between 1 and p, where p is the number of processors. It then tries to steal the bottom thread from the m^{th} deque (starting from the left end) in the global list of deques $\mathcal{R}$. A steal attempt may fail (that is, `pop_from_bot()` returns NULL) if two or more processors target the same deque (see Section 4.1), or if the deque is empty or non-existent. If the steal attempt is successful (`pop_from_bot()` returns a thread), the stealing processor creates a new deque for itself, places it to the immediate right of the target deque, and starts executing the stolen thread. Otherwise, it repeats the steal attempt. Thus the number of deques in $\mathcal{R}$ may grow during the execution beyond the number of processors; however, at any timestep, only the leftmost p deques are potential targets of a steal. When a processor steals the last thread from a deque not currently associated with (owned by) any processor, it deletes the deque.

If a thread contains an action that performs a memory allocation of m units such that $m > K$ (where K is the memory threshold), then $\lfloor m/K \rfloor$ dummy threads must be forked in a binary tree of depth $\Theta(\log m/K)$ before the allocation[5]. We do not show this extension in Figure 5 for brevity. Each dummy thread executes a no-op. However, processors must give up their deques and perform a steal every time they execute a dummy thread. Once all the dummy threads have been executed, a processor may proceed with the memory allocation. This transformation takes place at runtime. The addition of dummy threads effectively delays large allocations of space, so that higher priority threads may be scheduled instead. In practice, K is typically set to a few thousand bytes, so that the runtime overhead due to the dummy threads is negligible (e.g., see Section 5).

[5]This transformation differs slightly from depth-first schedulers [10, 41], which allow dummy threads to be forked in a multi-way fork of constant depth.

We now prove a lemma regarding the order of threads in $\mathcal{R}$ maintained by algorithm *DFDeques*; this order is shown pictorially in Figure 6.

Lemma 3.1 *Algorithm DFDeques maintains the following ordering of threads in the system.*

1. *Threads in each deque are in decreasing order of priorities from top to bottom.*
2. *A thread currently executing on a processor has higher priority than all other threads on the processor's deque.*
3. *The threads in any given deque have higher priorities than threads in all the deques to its right in $\mathcal{R}$.*

Proof: By induction on the timesteps. The base case is the start of the execution, when the root thread is the only thread in the system. Let the three properties be true at the start of any subsequent timestep. Any of the following events may take place on each processor during the timestep; we will show that the properties continue to hold at the end of the timestep.

When a thread forks a child thread, the parent is added to the top of the processor's deque, and the child starts execution. Since the parent has a higher priority that all other threads in the processor's deque (by induction), and since the child thread has a higher priority (earlier depth-first execution order) than its parent, properties (1) and (2) continue to hold. Further, since the child now has the priority immediately higher than its parent, property (3) holds.

When a thread T terminates, the processor checks if T has reactivated a suspended parent thread T_p. In this case, it starts executing T_p. Since the computation is nested parallel, the processor's deque must now be empty (since the parent T_p must have been stolen at some earlier point and then suspended). Therefore, all 3 conditions continue to hold. If T did not wake up its parent, the processor picks the next thread from the top its deque. If the deque is empty, it deletes the deque and performs a steal. Therefore all three properties continue to hold in these cases too.

When a thread suspends or is preempted due to exhaustion of the processor's memory quota, it is put back on the top of its deque, and the deque retains its position in $\mathcal{R}$. Thus all three properties continue to hold.

When a processor steals the bottom thread from another deque, it adds the new deque to the right of the target deque. Since the stolen thread had the lowest priority in the target deque, the properties continue to hold. Similarly, removal of a thread from the target deque does not affect the validity of the three properties for the target deque. A thread may be stolen from a processor's deque while one of the above events takes place on the processor itself; this does not affect the validity of our argument.

Finally, deletion of one or more deques from $\mathcal{R}$ does not affect the three properties. ∎

Work stealing as a special case of algorithm *DFDeques*. Consider the case when we set the memory threshold $K = \infty$. Then, for nested-parallel computations, algorithm *DFDeques*(∞) produces a schedule identical to the one produced by the provably-efficient work-stealing scheduler "WS" [13]. The processors in *DFDeques*(∞) never give up a deque due to exhaustion of their memory quota, and therefore, as with the work stealer, there are never more than p deques in the system. Further, in both algorithms, when a processor's deque becomes empty, it picks another processor uniformly at random, and steals the bottommost thread from that processor's deque.

Similarly, for nested parallel computations, the rule for waking up a suspended parent in *DFDeques*(∞) is equivalent to the corresponding rule in WS[6]. Of course, the resulting schedules are identical provided we assume the same cost model for both algorithms; the model could be either the atomic-access model used to analyze WS [13], or our cost model from Section 4.1.

4 Analysis of Time and Space Using Algorithm *DFDeques*

We now prove the space and time bounds for nested-parallel computations implemented using Algorithm *DFDeques*.

4.1 Cost model

We define the total number of unit actions in a parallel computation (or the number of nodes in its dag) as its ***work*** W. Further, let D be the ***depth*** of the computation, that is, the length of the longest path in its dag. For example, the computation represented in Figure 4 has work $W = 11$ and depth $D = 6$. We assume that an allocation of m bytes of memory (for any $m > 0$) has a depth of $\Theta(\log m)$ units[7].

For this analysis, we assume that timesteps (clock cycles) are synchronized across all the processors. If multiple processors target a non-empty deque in a single timestep, we assume that one of them succeeds in the steal, while all the others fail in that timestep. If the deque targeted by one or more steals is empty, all of those steals fail in a single timestep. When a steal fails, the processor attempts another steal in the next timestep. When a steal succeeds, the processor inserts the newly created deque into $\mathcal{R}$ and executes the first action from the stolen thread in the same timestep. At the end of a timestep, if a processor's current thread terminates or suspends, and it finds its deque to be empty, it immediately deletes its deque in that timestep. Similarly, when a processor steals the last thread from a deque not currently associated with any processor, it deletes the deque in that timestep. Thus, at the start of a timestep, if a deque is empty, it must be owned by a processor that is busy executing a thread.

Our cost model is somewhat simplistic, because it ignores the cost of maintaining the globally ordered set of deques $\mathcal{R}$. If we parallelize the scheduling tasks of inserting and deleting deques in $\mathcal{R}$ (by performing them lazily), we can account for all their overheads in the time bound. We can then show that in the expected case, the computation can be executed in $O(W/p + D \cdot \log p)$ time and $S_1 + O(p \cdot \log p \cdot D)$ space on p processors, including the scheduling overheads [40]. In practice, the insertions and deletions of deques from $\mathcal{R}$ can be either serialized and protected by a lock (for small p), or performed lazily in parallel (for large p).

---

[6] In WS, the reawakened parent is placed added to the current processor's deque (which is empty); for nested parallel computations, the child must terminate at this point, and therefore, the next thread executed by the processor is the parent thread.

[7] This is a reasonable assumption in systems with binary forks that zero out the memory *as soon as* it is allocated. The zeroing then requires a minimum depth of $\Theta(\log m)$; it can be performed in parallel by forking a tree of height $\Theta(\log m)$.

4.2 Space bound

We now analyze the space bound for a parallel computation executed by algorithm *DFDeques*. The analysis uses several ideas from previous work [3, 10, 41]. Because more than one processor is available to execute a parallel computation, some nodes may be executed out of order (i.e., prematurely) with respect to the serial, depth-first schedule. These out-of-order nodes can cause the parallel schedule to require more space than a serial schedule. By bounding the number of such nodes, we can bound the space requirement of the parallel schedule in terms of the serial space requirement.

4.2.1 Definitions

Let G be the dag that represents the parallel computation being executed. Depending on the resulting parallel schedule, we classify its nodes (actions) into one of two types: heavy and light. Every time a processor performs a steal, the first node it executes from the stolen thread is called a ***heavy*** action. All remaining nodes in G are labelled as ***light***.

We first assume that every node allocates at most K space; we will relax this assumption in the end. Recall that a processor may allocate at most K space between consecutive steals; thus, it may allocate at most K space for every heavy node it executes. Therefore, we can attribute all the memory allocated by light nodes to the last heavy node that precedes them. This results in a conservative view of the total space allocation.

Let $s_p = V_1, \ldots, V_\tau$ be the parallel schedule of the dag generated by algorithm *DFDeques*(K). Here V_i is the set of nodes that are executed at timestep i; τ is the ***length*** of the schedule s_p. Let s_1 be the serial, depth-first schedule or the 1DF***-schedule*** for the same dag; e.g., the nodes in Figure 4 are numbered according to their order of execution in a 1DF-schedule.

We now view an intermediate snapshot of the parallel schedule s_p. At any timestep $1 \leq j \leq \tau$ during the execution of s_p, all the nodes executed so far form a ***prefix*** of s_p. This prefix of s_p is defined as $\sigma_p = \bigcup_{i=1}^{j} V_i$. Let σ_1 be the *longest* prefix of s_1 containing only nodes in σ_p, that is, $\sigma_1 \subseteq \sigma_p$. Then the prefix σ_1 is called the ***corresponding*** serial prefix of σ_p. The nodes in the set $\sigma_p - \sigma_1$ are called ***premature*** nodes, since they have been executed out of order with respect to the 1DF-schedule s_1. All other nodes in σ_p, that is, the set σ_1, are called ***non-premature***. For example, Figure 7 shows a simple dag with a parallel prefix σ_p for an arbitrary p-schedule s_p, its corresponding serial prefix σ_1, and a possible classification of nodes as heavy or light. It also highlights the premature nodes in σ_p.

Whether a node is heavy or light is determined by the parallel schedule, while premature nodes are defined relative to a given prefix (snapshot) of the parallel schedule. For example, at the start and at the end of the execution (i.e., for the empty prefix as well as the longest prefix of s_p) there are no premature nodes, while intermediate prefixes of s_p may contain premature nodes. The maximum number of premature nodes (over all prefixes of the parallel schedule) will determine the amount of extra space the parallel schedule requires when compared with the 1DF-schedule (Lemma 4.3, Section 4.2.2); in contrast, the space allocated by non-premature nodes will be bounded by the space usage of the serial execution. Therefore, by bounding the total number of premature nodes (Lemma 4.2, Section 4.2.2) we can bound the space requirement of the p-schedule s_p.

A ready thread being present in a deque is equivalent to its first unexecuted node (action) being

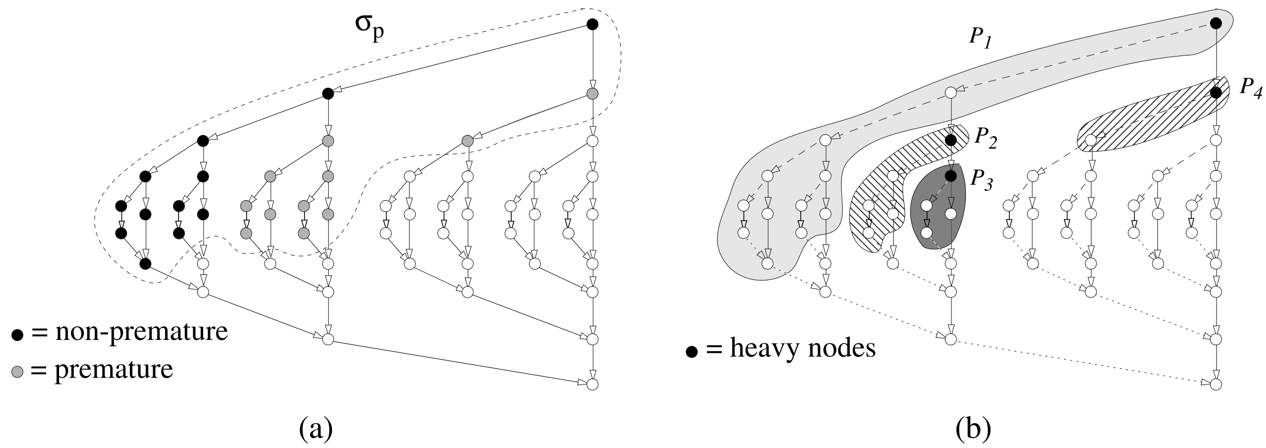


Figure 7: (a) An example snapshot of a parallel schedule for a simple dag. The shaded nodes (the set of nodes in σ_p) have been executed, while the blank (white) nodes have not. Of the nodes in σ_p, the black nodes form the corresponding parallel prefix σ_1, while the remaining grey nodes are premature. (b) A possible partitioning of nodes in σ_p into heavy and light nodes. Each shaded region denotes the set of nodes executed consecutively in depth-first order on a single processor (P_1, P_2, P_3 or P_4) between steals. The heavy node in each region is shown shaded black.

in the deque, and we will use the two phrases interchangeably. Given a p-schedule s_p of a dag G generated by algorithm *DFDeques*, we can find a unique ***last parent*** for every node in G (except for the root node) as follows. The last parent of a node u in G is defined as the last of u's parent nodes to be executed in the schedule s_p. If two or more parent nodes of u were the last to be executed, the processor executing one of them continues execution of u's thread. We label the unique parent of u executed by this processor as its last parent. This processor may have to preempt u's thread without executing u if it runs out of its memory quota; in this case, it puts u's thread on to its deque and then gives up the deque.

Consider the prefix σ_p of the parallel schedule s_p after the first j timesteps, for any $1 \leq j \leq \tau$. Let v be the last non-premature node (i.e., the last node from σ_1) to be executed during the first j timesteps of s_p. If more than one such node exist, let v be any one of them. Let $\mathcal{P}$ be a set of nodes in the dag constructed as follows: $\mathcal{P}$ is initialized to $\{v\}$; for every node u in $\mathcal{P}$, the last parent of u is added to $\mathcal{P}$. Since the root is the only node at depth 1, it must be in $\mathcal{P}$, and thus, $\mathcal{P}$ contains exactly all the nodes along a particular path from the root to v; we will call $\mathcal{P}$ the ***last path*** in σ_p. Further, since v is non-premature, all the nodes in $\mathcal{P}$ are non-premature.

Let u_i be the node in the last path $\mathcal{P}$ 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; then $t_1 = 1$ since the root is executed in the first timestep. For $i = 2, \ldots, \delta$ let I_i be the interval $\{t_{i-1}+1, \ldots, t_i\}$, and let $I_1 = \{1\}$. Let $I_{\delta+1} = \{t_\delta + 1, \ldots, j\}$. Since σ_p consists of all the nodes executed in the first j timesteps, the intervals $I_1, \ldots, I_{\delta+1}$ cover the duration of execution of all nodes in σ_p. We will call this unique set of disjoint intervals the ***covering intervals*** of σ_p. We will analyze the space requirement of the parallel execution by bounding the excess space allocated (compared to the 1DF-schedule) during each covering interval of σ_p.

4.2.2 Analysis of space bound

To analyze the space bound, we first bound the number of heavy premature nodes in any prefix of the parallel execution (Lemma 4.2). This bound is obtained by counting the maximum number of heavy premature nodes executed in each covering interval of σ_p. Since each heavy premature node can account for up to K additional space compared to the serial execution (here K is the memory threshold), we can relate the number of premature nodes to the space requirement of the parallel execution (Lemma 4.3).

We begin by proving the following lemma regarding the nodes in a deque below any of the nodes on the last path in σ_p.

Lemma 4.1 *Let σ_p be any prefix of the parallel schedule, and let $\mathcal{P}$ be the last path in σ_p. For any $1 \leq i \leq \delta$, if u_i is the node on the path $\mathcal{P}$ at depth i, then*

1. *If during the execution u_i is on some deque, then every node below it in its deque is the right child of some node in $\mathcal{P}$.*

2. *When u_i is executed on a processor, every node on the processor's deque must be the right child of some node in $\mathcal{P}$.*

Proof: We can prove this lemma to be true for any u_i by induction on i. The base case is the root node. Initially it is the only node in its deque, and gets executed before any new nodes are created. Thus, the lemma is trivially true. Let us assume the lemma is true for all u_j, for $0 \leq j \leq i$. We must prove that it is true for u_{i+1}.

Since u_i is the last parent of u_{i+1}, u_{i+1} becomes ready immediately after u_i is executed on some processor. There are two possibilities:

1. u_{i+1} is executed immediately following u_i on that processor. Property (1) hold trivially since u_{i+1} is never put on a deque. If the deque remains unchanged before u_{i+1} is executed, property (2) holds trivially for u_{i+1}. Otherwise, the only change that may be made to the deque is the addition of the right child of u_i before u_{i+1} is executed, if u_i was a fork with u_{i+1} as its left child. In this case too, property (2) holds, since the new node in the deque is right child of some node in $\mathcal{P}$.

2. u_{i+1} is added to the processor's deque after u_i is executed. This may happen because u_i was a fork and u_{i+1} was its right child (see Figure 8), or because the processor exhausted its memory quota. In the former case, since u_{i+1} is the right child of u_i, nothing can be added to the deque before u_{i+1}. In the latter case (that is, the memory quota is exhausted before u_{i+1} is executed), the only node that may be added to the deque before u_{i+1} is the right child of u_i, if u_i is a fork. This does not violate the lemma. Once u_{i+1} is added to the deque, it may either get executed on a processor when it becomes the topmost node in the deque, or it may get stolen. If it gets executed without being stolen, properties (1) and (2) hold, since no new nodes can be added below u_{i+1} in the deque. If it is stolen, the processor that steals and executes it has an empty deque, and therefore properties (1) and (2) are true, and continue to hold until u_{i+1} has been executed.

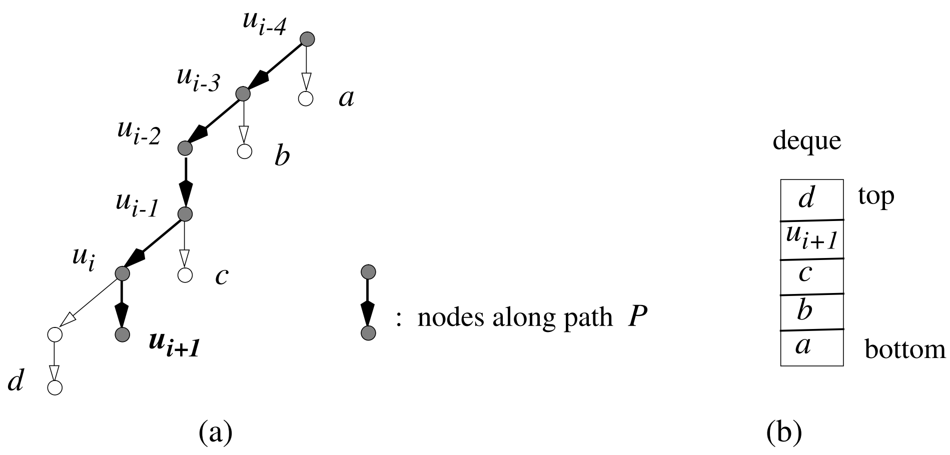


Figure 8: (a) A portion of the dynamically unfolding dag during the execution. Nodes $u_{i-4}, u_{i-3}, \ldots, u_i, u_{i+1}$ lie on the last path $\mathcal{P}$. Node u_{i+1} is ready, and is currently present in some deque. The deque is shown in (b); all nodes below u_{i+1} on the deque must be right children of some nodes on $\mathcal{P}$ above u_{i+1}. In this example, node u_{i+1} was the right child of u_i, and was added to the deque when the fork at u_i was executed. Subsequently, descendents of the left child of u_i (e.g., node d), may be added to the deque above u_{i+1}.

■

To prove the space bound, we first bound the *number of heavy premature nodes* in an arbitrary prefix σ_p of s_p (Lemma 4.2). The proof, detailed below, proceeds as follows: We focus on timesteps in which one or more heavy premature nodes may be executed. These timesteps are split into phases, such that each phase has a limited number of steal attempts. Since a heavy premature node can only be executed as a result of a steal, we bound the number of heavy premature nodes executed by bounding the number of such phases that can occur in the parallel prefix σ_p. The phases are bound by considering each covering interval of σ_p separately. The basic idea is to show that with a constant probability, the current ready node u along the last path in σ_p will get executed during a phase, since the deque containing u must be a candidate for steals. The current covering interval ends as soon as u is executed (by definition), thereby limiting the number of phases in the interval.

Lemma 4.2 *Let σ_p be any parallel prefix of a p-schedule produced by algorithm DFDeques(K) for a computation with depth D, in which every action allocates at most K space. Then the expected number of heavy premature nodes in σ_p is $O(p \cdot D)$. Further, for any $\epsilon > 0$, the number of heavy premature nodes is $O(p \cdot (D + \ln(1/\epsilon)))$ with probability at least $1 - \epsilon$.*

Proof: Consider the start of any covering interval I_i of σ_p, for $i = 1, \ldots, \delta$ (we will look at the last covering interval $I_{\delta+1}$ separately); here δ is the depth of the last non-premature node executed in σ_p. Let u_i be the node at depth i on the last path $\mathcal{P}$ in σ_p. By Lemma 3.1, all nodes in the deques to the left of u_i's deque, and all nodes above u_i in its deque are non-premature. Let x_i be the number of nodes below u_i in its deque. Because steals target the first (leftmost) p deques in the global list of deques $\mathcal{R}$, heavy premature nodes can be picked in any timestep from at most p deques. Further, every time a heavy premature node is picked, the deque containing u_i must also be a candidate deque to be picked as a target for a steal; that is, u_i must be among the leftmost p deques. Consider only the timesteps in which u_i is among the leftmost p deques; we will refer to such timesteps as ***candidate*** timesteps. Because new deques may be created to the left of u_i at any time, the candidate timesteps need not be contiguous.

We now bound the total number of steal attempts that take place during the candidate timesteps. Each such steal attempt may result in the execution of a heavy premature node; steals in all other timesteps result in the execution of heavy, but non-premature nodes. Each timestep can have at most p steal attempts. Therefore, we can partition the candidate timesteps into ***phases***, such that each phase has between p and $2p-1$ steal attempts. We call a phase in interval I_i ***successful*** if at least one of its $\Theta(p)$ steal attempts targets the deque containing u_i. Let X_{ij} be the random variable with value 1 if the j^{th} phase in interval I_i is successful, and 0 otherwise. Because targets for steal attempts are chosen at random from the leftmost p deques with uniform probability, and because each phase has at least p steal attempts,

$$\begin{aligned}
\Pr\left[X_{ij}=1\right] &\geq 1-\left(1-\frac{1}{p}\right)^{p} \\
&\geq 1-\frac{1}{e} \\
&\geq \frac{1}{2}
\end{aligned}$$

Thus, each phase succeeds with probability at least $1/2$. Because u_i had x_i nodes below it in its deque, it must get executed before or by the time x_i+1 successful steals target u_i's deque. Therefore, there can be at most x_i+1 successful phases in the covering interval I_i. The node u_i may get executed before x_i+1 steal attempts target its deque, if its owner processor executes u_i off the top of the deque. Let there be some $n_i \leq (x_i+1)$ successful phases in the interval I_i. From Lemma 4.1, the x_i nodes below u_i are right children of nodes on the last path $\mathcal{P}$. There are $(\delta-1) < D$ nodes along $\mathcal{P}$ not including u_δ, and each of them may have at most one right child. Since each of these right children can only get executed once during one of the first δ covering intervals, $\sum_{i=1}^{\delta} x_i < D$. Therefore, the total number of successful phases in the first δ intervals is $\sum_{i=1}^{\delta} n_i = \sum_{i=1}^{\delta}(x_i+1) < (D+\delta) \leq 2D$.

Finally, consider the last covering interval $I_{\delta+1}$. Let z be the ready node at the start of the interval with the highest priority. Then, $z \notin \sigma_p$, because otherwise z (or some other node), and not v, would have been the last non-premature node to be executed in σ_p. Hence, if z is about to be executed on a processor, then interval $I_{\delta+1}$ is empty. Otherwise, z must be at the top of the leftmost deque at the start of interval $I_{\delta+1}$. Using an argument similar to that of Lemma 4.1, we can show that the nodes below z in the deque must be right children of nodes along a path from the root to z. Thus, z can have at most $(D-2)$ nodes below it. Because z must be among the leftmost p deques throughout the interval $I_{\delta+1}$, the phases in this interval are formed from all its timesteps. We call a phase ***successful*** in interval $I_{\delta+1}$ if at least one of the $\Theta(p)$ steal attempts in the phase targets the deque containing z. Then this interval must have less than D successful phases. As before, the probability of a phase being successful is at least $1/2$.

We have shown that for any $j \leq \tau$ (here τ is the length of the parallel schedule s_p), the first j timesteps of the parallel execution, represented by the parallel prefix σ_p, must have less than $3D$ successful phases[8]. Since a heavy premature node can only be executed after a steal, each phase may result in less than $2p$ heavy premature nodes being stolen and executed. Further, for $i = 1, \ldots, \delta$, in each interval I_i of σ_p, another $p-1$ heavy premature nodes may be executed in the same timestep that u_i is executed. Since $\delta \leq D$, if σ_p has a total of, say, N phases, the number of heavy premature nodes in σ_p is less than $(2N + D) \cdot p$. Because the entire execution must have less than $3D$ successful phases, and each phase succeeds with probability at least $1/2$, the expected number of total phases before we see $3D$ successful phases is less than $6D$, that is, $\mathrm{E}[N] < 6D$. Therefore, the expected number of heavy premature nodes in σ_p is bounded by $\mathrm{E}[(2N + D) \cdot p] < (12D + D) \cdot p = O(p \cdot D)$.

The high probability bound can be proved as follows. Suppose the execution takes at least $12D + 8\ln(1/\epsilon)$ phases. Then the expected number of successful phases is at least $\mu = 6D + 4\ln(1/\epsilon)$. Using the Chernoff bound [39, Theorem 4.2] on the number of successful phases X, and setting $a = 6D + 8\ln(1/\epsilon)$, we get[9]

$$\Pr[X < \mu - a/2] \quad < \quad \exp\left[\frac{-(a/2)^2}{2\mu}\right]$$

Therefore,

$$\begin{aligned}
\Pr[(X < 3D)] \quad &< \quad \exp\left[\frac{-a^2/4}{12D + 8\ln(1/\epsilon)}\right] \\
&= \quad \exp\left[\frac{-a^2}{4 \cdot (2a - 8\ln(1/\epsilon))}\right] \\
&\leq \quad e^{-a^2/8a} \\
&= \quad e^{-a/8} \\
&= \quad e^{-(6D + 8\ln(1/\epsilon))/8} \\
&< \quad e^{-8\ln(1/\epsilon)/8} \\
&= \quad \epsilon
\end{aligned}$$

Because there can be at most $3D$ successful phases, algorithm *DFDeques* requires $12D + 8\ln(1/\epsilon)$ or more phases with probability at most ϵ. Recall that each phase consists of $\Theta(p)$ steal attempts. Therefore, σ_p has $O(p \cdot (D + \ln(1/\epsilon)))$ heavy premature nodes with probability at least $1 - \epsilon$. ∎

We can now state a lemma relating the number of heavy premature nodes in σ_p with the memory requirement of s_p.

Lemma 4.3 *Let G be a dag with depth D, in which every node allocates at most K space, and for which the serial depth-first execution requires S_1 space. Let s_p be the p-schedule of length T*

[8]Remember that the parallel prefix σ_p, and hence the classification of nodes as premature or non-premature, or phases as successful, depends on the choice of this j.

[9]The probability of success for a phase is not necessarily independent of previous phases; however, because each phase succeeds with probability at least $1/2$, independent of other phases, we can apply the Chernoff bound.

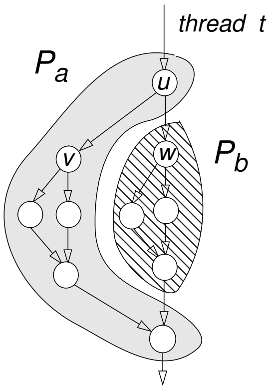


Figure 9: An example scenario when a processor (P_a in this example) may not execute a contiguous subsequence of nodes between steals. The shaded regions indicate the subset of nodes executed on each of the two processors, P_a and P_b. Here, processor P_a steals the thread t and executes node u. It then forks a child thread (containing node v), puts thread t on its deque, and starts executing the child. In the mean time, processor P_b steals thread t from the deque belonging to P_a, and executes it until it suspends. Subsequently, P_a finished executing the child thread, and wakes up the suspended parent t and resumes execution of t. The combined sets of nodes executed on both processors forms a contiguous subsequence of 1DF-schedule.

generated for G by algorithm DFDeques(K). *If for any i such that $1 \leq i \leq T$, the prefix σ_p of s_p representing the computation after the first i timesteps contains at most r heavy premature nodes, then the parallel space requirement of s_p is at most $S_1 + r \cdot \min(K, S_1)$. Further, there are at most $D + r \cdot \min(K, S_1)$ active threads during the execution.*

Proof: We can partition σ_p into the set of non-premature nodes and the set of premature nodes. Since, by definition, all non-premature nodes form some serial prefix of the 1DF-schedule, their net memory allocation cannot exceed S_1. We now bound the net memory allocated by the premature nodes. Consider a steal that results in the execution of a heavy premature node on a processor P. The nodes executed by P until its next steal, cannot allocate more than K space. Because there are at most r heavy premature nodes executed, the total space allocated across all processors after i timesteps cannot exceed $S_1 + r \cdot K$.

The maximum number of active threads is at most the number of threads with premature nodes, plus the maximum number of active threads during a serial execution, which is D. Assuming that each thread needs to allocate at least a unit of space when it is forked (e.g., to store its register state), at most K threads with premature nodes can be forked for each heavy premature node executed. Therefore, the total number of active threads is at most $D + r \cdot K$. ∎

Note that each active thread requires at most a constant amount of space to be stored by the scheduler (not including its stack space). We now extend the analysis to handle large allocations (of more than K space).

Handling large allocations of space. We had assumed earlier in this section that every node allocates at most K units of memory. Individual nodes that allocate more than K space are handled as described in Section 3. The key idea is to delay the big allocations, so that if threads with higher priorities become ready, they will be executed instead. The solution is to insert before every allocation of m bytes ($m > K$), a binary fork tree of depth $\log(m/K)$, so that m/K dummy threads are created at its leaves. Each of the dummy threads simply performs a no-op that takes one timestep, but the threads at the leaves of the fork tree are treated as if it were allocating K space; a processor gives up its deque and performs a steal after executing each of these dummy threads. Therefore, by the time the m/K dummy threads are executed, a processor may proceed with the allocation of m bytes without exceeding our space bound. Recall that in our cost model, an allocation of m bytes requires a depth of $O(\log m)$; therefore, this transformation of the dag increases its depth by at most a constant factor. The transformation takes place at runtime, and the on-line *DFDeques* algorithm generates a schedule for this transformed dag. The final bound on the space requirement of the generated schedule, is stated below. Also, in Lemma 4.3, each node was assumed to allocate at most K space; since the execution can never have a net negative space allocation, this assumption required that $KleS_1$. Without this assumption, we need to additionally bound memory allocation for very large K; we therefore prove below a tighter bound for the space requirement when $K > S_1$.

Theorem 4.4 *(***Upper bound on space requirement)**
Consider a nested-parallel computation with depth D and serial, depth-first space requirement S_1. Then, for any $K > 0$, the expected value of the space required to execute the computation on p processors using algorithm DFDeques(K), including the space required to store active threads, is $S_1 + O(\min(K, S_1) \cdot p \cdot D)$. Further, for any $\epsilon > 0$, the probability that the computation requires $S_1 + O(\min(K, S_1) \cdot p \cdot (D + \ln(1/\epsilon)))$ space is at least $1 - \epsilon$.

Proof: Lemmas 4.2 and 4.3 hold for any prefix (snapshot) of the parallel computation. Therefore, using the above transformation of the dag for allocations larger than K, it follows that the expected amount of space required by the parallel computation is $S_1 + O(K \cdot p \cdot D)$. Further, it follows that for any $\epsilon > 0$, the probability that the computation requires $S_1 + O(K \cdot p \cdot (D + \ln(1/\epsilon)))$ space is at least $1 - \epsilon$.

We now obtain a tighter bound when $K > S_1$. Consider the case when a processor P steals a thread and executes a heavy premature node. The nodes executed by P before the next steal are all premature, and form a series of one or more subsequences of the 1DF-schedule. The intermediate nodes between these subsequences (in depth-first order) are executed on other processors (e.g., see Figure 9). These intermediate nodes occur when other processors steal threads from the deque belonging to P, and finish executing the stolen threads before P finishes executing all the remaining threads in its deque. Subsequently, when P's deque becomes empty, the thread executing on P may wake up its parent, so that P starts executing the parent without performing another steal. Therefore, the set of nodes executed by P before the next steal, possibly along with premature nodes executed on other processors, form a contiguous subsequence of the 1DF-schedule.

Assuming that the net space allocated during the 1DF-schedule can never be negative, this subsequence cannot allocate more than S_1 units of net memory. Therefore, if there are r heavy premature nodes, the net memory allocation of all the premature nodes cannot exceed $r \cdot \min(K, S_1)$, and the total space allocated across all processors after i timesteps cannot exceed $S_1 + r \cdot \min(K, S_1)$. Therefore, using Lemma 4.2, the above expected and high probability space bounds follow. ∎

We now show that the above space bound is tight (within constant factors) in the expected case, for algorithm *DFDeques*.

Theorem 4.5 (Lower bound on space requirement)
For any $S_1 > 0$, $p > 0$, $K > 0$, and $D \geq 24\log p$, there exists a nested parallel dag with a serial space requirement of S_1 and depth D, such that the expected space required by algorithm DFDeques(K) to execute it on p processors is $\Omega(S_1 + \min(K, S_1) \cdot p \cdot D)$.

Proof: Consider the dag shown in Figure 10. The black nodes denote allocations, while the grey nodes denote deallocations. The dag essentially has a fork tree of depth $\log(p/2)$, at the leaves of which exist subgraphs[10]. The root nodes of these subgraphs are labelled $u_1, u_2, \ldots, u_n$, where $n = p/2$. The leftmost of these subgraphs, G_0, shown in Figure 10 (b), consists of a serial chain of d nodes. The remaining subgraphs are identical, have a depth of $2d + 1$, and are shown in Figure 10 (c). The amount of space allocated by each of the black nodes in these subgraphs is defined as $A = \min(K, S_1)$. Since we are constructing a dag of depth D, the value of d is set such that $2d + 1 + 2\log(p/2) = D$. The space requirement of a 1DF-schedule for this dag is S_1.

We now examine how algorithm *DFDeques*(K) would execute such a dag. One processor starts executing the root node, and executes the left child of the current node at each timestep. Thus, within $\log(p/2) = \log n$ timesteps, it will have executed node u_1. Now consider node u_n; it is guaranteed to be executed once $\log n$ successful steals target the root thread. (Recall that the right child of a forking node, that is, the next node in the parent thread, must be executed either before or when the parent thread is next stolen.) Because there are always $n = p/2$ processors in this example that are idle and attempt steals targeting p deques at the start of every timestep, the probability p_{steal} that a steal will target a particular deque is given by

$$\begin{aligned} p_{\text{steal}} &\geq 1 - \left(1 - \frac{1}{p}\right)^{p/2} \\ &\geq 1 - e^{-1/2} \\ &> \frac{1}{3} \end{aligned}$$

We call a timestep i ***successful*** if some node along the path from the root to u_n gets executed; this happens when a steal targets the deque containing that node. Thus, after $\log n$ successful timesteps, node u_n must get executed; after that, we will consider every subsequent timestep to be successful. Let S be the number of successful timesteps in the first $12\log n$ timesteps. Then, the expected value is given by

$$\begin{aligned} \mathrm{E}[S] &\geq 12\log n \cdot p_{\text{steal}} \\ &\geq 4\log n \end{aligned}$$

Using the Chernoff bound [39, Theorem 4.2] on the number of successful timesteps, we have

$$\Pr[S < \left(1 - \frac{3}{4}\right) \cdot \mathrm{E}[S]\,] \;\leq\; \exp\left[-\left(\frac{3}{4}\right)^2 \cdot \frac{\mathrm{E}[S]}{2}\right]$$

[10] All logarithms denoted as $\log$ are to the base 2.

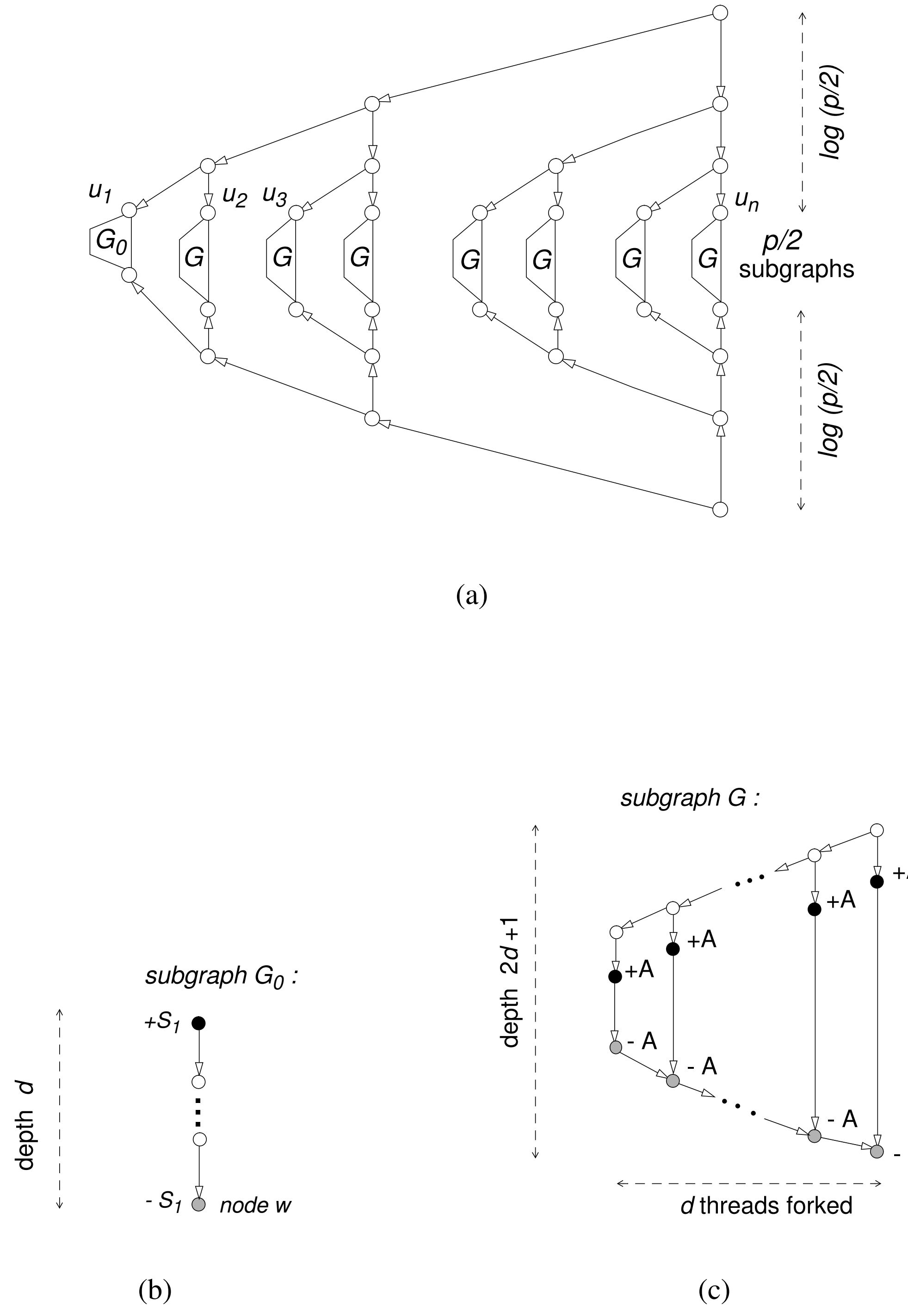


Figure 10: (a) The dag for which the existential lower bound holds. (b) and (c) present the details of the subgraphs shown in (a). The black nodes denote allocations and grey nodes denote deallocations; the nodes are marked with the amount of memory (de)allocated. Here, $A = \min(K, S_1)$.

Therefore,

$$\begin{aligned}
\Pr[S < \log n] &\leq \exp\left[-\frac{9}{8}\log n\right] \\
&= \exp\left[-\frac{9}{8} \cdot \frac{\ln n}{\ln 2}\right] \\
&< e^{-1.62 \cdot \ln n} \\
&= n^{-0.62} \cdot \frac{1}{n} \\
&< \frac{2}{3} \cdot \frac{1}{n} \quad \text{for } p \geq 4
\end{aligned}$$

Recall that $n = p/2$. (The case of $p < 4$ can be easily handled separately.) Let $\mathcal{E}_i$ be the event that node u_i is *not* executed within the first $12 \log n$ timesteps. We have showed that $\Pr[\mathcal{E}_n] < (2/3) \cdot (1/n)$. Similarly, we can show that for each $i = 1, \ldots, n-1$, $\Pr[\mathcal{E}_i] < (2/3) \cdot (1/n)$. Therefore, $\Pr[\bigcup_1^n \mathcal{E}_i] < 2/3$. Thus, for $i = 1, \ldots, n$, all the u_i nodes get executed within the first $12 \log n$ timesteps with probability greater than $1/3$.

Each subgraph G has d nodes at different depths that allocate memory; the first of these nodes cannot be executed before timestep $\log n$. Let t be the first timestep at which all the u_i nodes have been executed. Then, at this timestep, there are at least $(d + \log n - t)$ nodes remaining in each subgraph G that allocate A bytes each, but have not yet been executed. Similarly, node w in subgraph G_0 will not be executed before timestep $(d + \log n)$, that is, another $(d + \log n - t)$ timesteps after timestep t. Therefore, for the next $(d + \log n - t)$ timesteps there are always $n - 1 = (p/2) - 1$ non-empty deques (out of a total of p deques) during the execution. Each time a thread is stolen from one of these deques, a black node (from Figure 10 (c)) is executed, and the thread then suspends. Because $p/2$ processors become idle and attempt a steal at the start of each timestep, we can show that in the expected case, at least a constant fraction of the $p/2$ steals are successful in every timestep. Each successful steal results in $A = \min(S_1, K)$ units of memory being allocated. Consider the case when $t = 12 \log n$. Then, using linearity of expectations, over the $d - 11 \log n$ timesteps after timestep t, the expected value of the total space allocated is $S_1 + \Omega(A \cdot p \cdot (d - 11 \log n)) = S_1 + \Omega(A \cdot p \cdot (D - \log p))$. ($D \geq 24 \log p$ ensures that $(d - 11 \log n) > 0$.)

We showed that with constant probability ($> 1/3$), all the u_i nodes will be executed within the first $12 \log n$ timesteps. Therefore, in the expected case, the space allocated (at some point during the execution after all u_i nodes have been executed) is $\Omega(S_1 + \min(S_1, K) \cdot (D - logp) \cdot p)$. ∎

Corollary 4.6 (Lower bound using work stealing)
For any $S_1 > 0$, $p > 0$, and $D \geq 24 \log p$, there exists a nested parallel dag with a serial space requirement of S_1 and depth D, such that the expected space required to execute it using the space-efficient work stealer from [13] on p processors is $\Omega(S_1 \cdot p \cdot D)$. ∎

The corollary follows from Theorem 4.5 and the fact that algorithm *DFDeques* behaves like the space-efficient work-stealing scheduler for $K = \infty$. Blumofe and Leiserson [13] presented an upper bound on space of $p \cdot S_1$ using randomized work stealing. Their result is not inconsistent with the above corollary, because their analysis allows only "stack-like" memory allocation[11], which is more restricted than our model. For such restricted dags, their space bound of $p \cdot S_1$ also applies directly to *DFDeques*(∞). Our lower bound for $K = \infty$ is also consistent with the upper bound of $p \cdot S$ by Simpson and Burton [51], where S is the maximum space requirement over all possible depth-first schedules. In this example, $S = S_1 \cdot D$, since the right-to-left depth-first schedule requires $S_1 \cdot D$ space.

4.3 Time bound

We now prove the time bound required for a parallel computation using algorithm *DFDeques*. This time bound does not include the scheduling costs of maintaining the relative order of the deques (i.e., inserting and deleting deques in the global list of deques $\mathcal{R}$), or finding the m^{th} deque. Elsewhere [40], we describe how the scheduler can be parallelized, and then prove the time bound including these scheduling costs. We first assume that every action allocates at most K space (where K is the memory threshold used by the *DFDeques* algorithm) and prove the time bound. We then relax this assumption and provide the modified time bound at the end of this subsection.

Lemma 4.7 *Consider a parallel computation with work W and depth D, in which every action allocates at most K space. The expected time to execute this computation on p processors using the DFDeques(K) scheduling algorithm is $O(W/p + D)$. Further, for any $\epsilon > 0$, the time required to execute the computation is $O(W/p + D + \ln(1/\epsilon))$ with probability at least $1 - \epsilon$.*

Proof: Consider any timestep i of the p-schedule; let n_i be the number of deques in $\mathcal{R}$ at timestep i. We first classify each timestep i into one of two types (A and B), depending on the value of n_i. We then bound the total number of timesteps T_A and T_B of types A and B, respectively.

Type A: $n_i \geq p$. At the start of timestep i, let there be $r \leq p$ steal attempts in this timestep. Then the remaining $p - r$ processors are busy executing nodes, that is, at least $p - r$ nodes are executed in timestep i. Further, at most $p - r$ of the leftmost p deques may be empty; the rest must have at least one thread in them.

Let X_j be the random variable with value 1 if the j^{th} non-empty deque in $\mathcal{R}$ (from the left end) gets exactly one steal request, and 0 otherwise. Then, $\mathrm{E}[X_j] = \Pr[X_j = 1] = (r/p) \cdot (1 - 1/p)^{r-1}$. Let X be the random variable representing the total number of non-empty deques that get exactly one steal request[12]. Because there are at least r non-empty deques, the expected value of X (assuming that $p \geq 2$) is given by

$$\begin{aligned}
\mathrm{E}[X] &\geq \sum_{j=1}^{r} \mathrm{E}[X_j] \\
&= r \cdot \frac{r}{p} \cdot (1 - \frac{1}{p})^{r-1}
\end{aligned}$$

[11]Their model does not allow allocation of space on a global heap. An instruction in a thread may allocate stack space only if the thread cannot possibly have a living child when the instruction is executed. The stack space allocated by the thread must be freed when the thread terminates.

[12]For simplicity, we only count the deques that get exactly one request, instead of any non-zero number of requests.

$$\begin{aligned}
&\geq \frac{r^2}{p}\cdot(1-\frac{1}{p})^p \\
&\geq \frac{r^2}{p}\cdot(1-\frac{1}{p})\cdot\frac{1}{e} \\
&\geq \frac{r^2}{2\cdot p\cdot e}
\end{aligned}$$

Recall that $p-r$ nodes are executed by the busy processors. Therefore, if Y is the random variable denoting the total number of nodes executed during this timestep, then

$$\begin{aligned}
\mathrm{E}\left[Y\right] &\geq (p-r)+\frac{r^2}{2ep} \\
&\geq \frac{p}{2e} \\
\text{Therefore,}\quad \mathrm{E}\left[p-Y\right] &\leq p-\frac{p}{2e} \\
&= p(1-\frac{1}{2e})
\end{aligned}$$

The quantity $(p-Y)$ must be non-negative; therefore, using the Markov's inequality [39, Theorem 3.2], we get

$$\begin{aligned}
\Pr\left[(p-Y)>p(1-\frac{1}{4e})\right] &< \frac{\mathrm{E}\left[(p-Y)\right]}{p\left(1-\frac{1}{4e}\right)} \\
&\leq \frac{\left(1-\frac{1}{2e}\right)}{\left(1-\frac{1}{4e}\right)} \\
\text{Therefore,}\ \Pr\left[Y<\frac{p}{4e}\right] &< \frac{9}{10} \\
\text{that is,}\ \Pr\left[Y\geq\frac{p}{4e}\right] &> \frac{1}{10}
\end{aligned}$$

We will call each timestep of type A ***successful*** if at least $p/4e$ nodes get executed during the timestep. Then the probability of the timestep being successful is at least $1/10$. Because there are W nodes in the entire computation, there can be at most $4e\cdot W/p$ successful timesteps of type A. Therefore, the expected value for T_A is at most $40e\cdot W/p$.

The analysis of the high probability bound is similar to that for Lemma 4.2. Suppose the execution takes more than $80eW/p+40\ln(1/\epsilon)$ timesteps of type A. Then the expected number μ of successful timesteps of type A is at least $8eW/p+4\ln(1/\epsilon)$. If Z is the random variable denoting the total number of successful timesteps, then using the Chernoff bound [39, Theorem 4.2], and setting $a=40eW/p+40\ln(1/\epsilon)$, we get[13]

$$\Pr\left[Z<\mu-a/10\right] \;<\; \exp\left[\frac{-\left(a/10\right)^2}{2\mu}\right]$$

[13]As with the proof of Lemma 4.2, we can use the Chernoff bound here because each timestep succeeds with probability at least $1/10$, even if the exact probabilities of successes for timesteps are not independent.

Therefore,

$$\begin{aligned}
\Pr\left[Z < 4eW/p\right] &< e^{-a^2/200\mu} \\
&= \exp\left[-\frac{a^2}{200(a/5 - 4\ln(1/\epsilon))}\right] \\
&\leq \exp\left[-\frac{a^2}{200 \cdot a/5}\right] \\
&= e^{-a/40} \\
&= e^{-eW/p - \ln(1/\epsilon)} \\
&\leq e^{-\ln(1/\epsilon)} \\
&= \epsilon
\end{aligned}$$

We have shown that the execution will not complete even after $80eW/p + 40\ln(1/\epsilon)$ type A timesteps with probability at most ϵ. Thus, for any $\epsilon > 0$, $T_A = O(W/p + \ln(1/\epsilon))$ with probability at least $1 - \epsilon$.

Type B: $n_i < p$. We now consider timesteps in which the number of deques in $\mathcal{R}$ is less than p. As with the proof of Lemma 4.2, we split type B timesteps into phases such that each phase has between p and $2p - 1$ steal attempts. We can then use a potential function argument similar to the dedicated machine case by Arora et al. [3]. Composing phases from only type B timesteps (ignoring type A timesteps) retains the validity of their analysis. We briefly outline the proof here. Nodes are assigned exponentially decreasing potentials starting from the root downwards. Thus, a node at a depth of d is assigned a potential of $3^{2(D-d)}$, and in the timestep in which it is about to be executed on a processor, a weight of $3^{2(D-d)-1}$. They show that in any phase during which between p and $2p - 1$ steal attempts occur, the total potential of the nodes in all the deques drops by a constant factor with at least a constant probability. Since the potential at the start of the execution is 3^{2D-1}, the expected value of the total number of phases is $O(D)$. The difference with our algorithm is that a processor may execute a node, and then put up to 2 (instead of 1) children of the node on the deque if it runs out of memory; however, this difference does not violate the basis of their arguments. Since each phase has $\Theta(p)$ steal attempts, the expected number of steal attempts during type B timesteps is $O(pD)$. Further, for any $\epsilon > 0$, we can show that the total number of steal attempts during timesteps of type B is $O(p \cdot (D + \ln(1/\epsilon)))$ with probability at least $1 - \epsilon$.

Recall that in every timestep, each processor either executes a steal attempt that fails, or executes a node from the dag. Therefore, if N_{steal} is the total the number of steal attempts during type B timesteps, then T_B is at most $(W + N_{\text{steal}})/p$. Therefore, the expected value for T_B is $O(W/p + D)$, and for any $\epsilon > 0$, the number of timesteps is $O(W/p + D + \ln(1/\epsilon))$ with probability at least $1 - \epsilon$.

The total number of timesteps in the entire execution is $T_A + T_B$. Therefore, the expected number of timesteps in the execution is $O(W/p + D)$. Further, combining the high probability bounds for timesteps of type A and B, (and using the fact that $\Pr\left[X \cup Y\right] \leq \Pr\left[X\right] + \Pr\left[Y\right]$), we can show that for any $\epsilon > 0$, the total number of timesteps in the parallel execution is $O(W/p + D + \ln(1/\epsilon))$ with probability at least $1 - \epsilon$. ∎

To handle each large allocation of m units (where $m > K$), recall that we add $\lfloor m/K \rfloor$ dummy threads; the dummy threads are forked in a binary tree of depth $\Theta(\log(m/K))$. Because we assume a depth of $\Theta(\log m)$ for every allocation of m bytes, 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 $W + S_a/K$. Therefore, using Lemma 4.7, the modified time bound is stated as follows.

Theorem 4.8 *(***Upper bound on time requirement)**
The expected time to execute a parallel computation with W work, D depth, and total space allocation S_a on p processors using algorithm DFDeques(K) *is $O(W/p + S_a/pK + D)$. Further, for any $\epsilon > 0$, the time required to execute the computation is $O(W/p + S_a/pK + D + \ln(1/\epsilon))$ with probability at least $1 - \epsilon$.*

In a system where every memory location allocated must be zeroed, $S_a = O(W)$. The expected time bound therefore becomes $O(W/p + D)$. This time bound, although asymptotically optimal [14], is not as low as the time bound of $W/p + O(D)$ for work stealing [13].

Trade-off between space, time, and scheduling granularity. As the memory threshold K is increased, the scheduling granularity increases, since a processor can execute more instructions between steals. In addition, the number of dummy threads added before large allocations decreases. However, the space requirement increases with K. Thus, adjusting the value of K provides a trade-off between running time (or scheduling granularity), and space requirement.

5 Experiments with Pthreads

We implemented the scheduler as part of an existing library for Posix standard threads or Pthreads [30]. The library is the native, user-level Pthreads library on Solaris 2.5 [46, 53]. Pthreads on Solaris are multiplexed at the user level on top of kernel threads, which act like virtual processors. The original scheduler in the Pthread library uses a FIFO queue. Our experiments were conducted on an 8 processor Enterprise 5000 SMP with 2GB main memory. Each processor is a 167 MHz UltraSPARC with a 512 kB L2 cache.

Having to support the general Pthreads functionality prevents even a user-level Pthreads implementation from being extremely lightweight. For example, a thread creation is two orders of magnitude more expensive than a null function call on the UltraSPARC. Therefore, the user is required to create Pthreads that are coarse enough to amortize the cost of thread operations. However, with a depth-first scheduler, threads at this granularity had to be coarsened further to get good parallel performance [42]. We show that using algorithm *DFDeques*, good speedups can be achieved using Pthreads without this additional coarsening. Thus, the user can now fix the thread granularity to amortize thread operation costs, and expect to get good parallel performance in both space and time.

The Pthreads model supports a binary fork and join mechanism. We modified memory allocation routines `malloc` and `free` to keep track of the memory quota of the current processor (or kernel thread) and to fork dummy threads before an allocation if required. Our scheduler implementation is a simple extension of algorithm *DFDeques* that supports the full Pthreads functionality (including blocking[14] mutexes and condition variables) by maintaining additional entries in $\mathcal{R}$ for threads suspended on synchronizations. Our benchmarks are predominantly nested parallel, and make limited use of mutexes and condition variables. For example, the tree-building phase in Barnes-Hut uses mutexes to protect modifications to the tree's cells. However, the Solaris Pthreads implementation itself makes extensive use of blocking synchronization primitives such as Pthread mutexes and condition variables.

Since our execution platform is an SMP with a modest number of processors, access to the ready threads in $\mathcal{R}$ was serialized. $\mathcal{R}$ is implemented as a linked list of deques protected by a shared scheduler lock. We optimized the common cases of pushing and popping threads onto a processor's current deque by minimizing locking time. A steal requires the lock to be acquired more often and for a longer period of time.

In the existing Pthreads implementation, it is not always possible to place a reawakened thread on the same deque as the thread that wakes it up; therefore, our implementation of *DFDeques* is an approximation of the pseudocode in Figure 5. Further, since we serialize access to $\mathcal{R}$, and support mutexes and condition variables, setting the memory threshold K to infinity does not produce the same schedule as the space-efficient work-stealing scheduler intended for fully strict computations [13]. Therefore, we can use this setting only as a *rough approximation* of a pure work-stealing scheduler.

We first list the benchmarks used in our experiments. Next, we compare the space and time performance of the library's original scheduler (labelled "FIFO") with an asynchronous, depth-first scheduler [42] (labelled "ADF"), and the new *DFDeques* scheduler (labelled "DFD") for a fixed value of the memory threshold K. We also use *DFDeques*(∞) as an approximation for a work-stealing scheduler (labelled "DFD-inf"). To study how the performance of the schedulers is affected by thread granularity, we present results of the experiments at two different thread granularities. Finally, we measure the trade-off between running time, scheduling granularity, and space for algorithm *DFDeques* by varying the value of the memory threshold K for one of the benchmarks.

5.1 Parallel benchmarks

The benchmarks were either adapted from publicly available coarse grained versions [25, 44, 52, 56], or written from scratch using the lightweight threads model [42]. The parallelism in both divide-and-conquer recursion and parallel loops was expressed as a binary tree of forks, with a separate Pthread created for each recursive call. Thread granularity was adjusted by serializing the recursion near the leafs. In the comparison results in Section 5.2, ***medium*** granularity refers to the thread granularity that provides good parallel performance using the depth-first scheduler [42]. Even at medium granularity, the number of threads significantly exceeds the number of processors; this allows simple coding and automatic load balancing, while resulting in performance equivalent to hand-partitioned, coarse-grained code using the depth-first scheduler [42]. ***Fine*** granularity refers to the finest thread granularity that allows the cost of thread operations in a single-processor

[14] We use the term "blocking" for synchronization that causes the calling thread to block and suspend, rather than spin wait.

Benchmark	Input size	Medium grained				Fine grained			
		total	FIFO	ADF	DFD	total	FIFO	ADF	DFD
Vol. Rend.	256^3 vol, 375^2 img	1427	195	29	29	4499	436	36	37
Dense MM	1024×1024 doubles	4687	623	33	48	37491	3752	55	77
Sparse MVM	30K nodes, 151K edges	1263	54	31	31	5103	173	51	49
FFTW	$N = 2^{22}$	177	64	13	18	1777	510	30	33
FMM	$N = 10K$, 5 mpl terms	4500	1314	21	29	36676	2030	50	54
Barnes Hut	$N = 100K$, Plmr model	40893	1264	33	106	124767	3570	42	120
Decision Tree	133,999 instances	3059	82	60	77	6995	194	138	149

Figure 11: Input sizes for each benchmark, total number of threads expressed in the program at medium and fine granularities, and max. number of simultaneously active threads created by each scheduler at both granularities, for K = 50,000 bytes. “DFD-inf” creates at most twice as many threads as “DFD” for Dense MM, and at most 15% more threads than “DFD” for the remaining benchmarks.

execution to be up to 5% of the serial execution time[15]. The parallel benchmarks are briefly described below.

1. **Volume Rendering.** This application was adapted from the Splash-2 volume rendering benchmark [56, 52]. A ray is cast from the viewing position through each pixel in the image plane; parallelism is exploited across these pixels. We create a separate Pthread to handle each set of tiles in the image, where a tile is 4×4 pixels. The granularity is varied by limiting the number of tiles in each set. Rays cast through tiles close together in the image are likely to access much of the same volume data, therefore processing such tiles on the same processor provides good locality. Each thread processes up to 100 pixel tiles of the rendered image at medium granularity, and up to 5 tiles at fine granularity.

2. **Dense Matrix Multiply.** We use a recursive, divide-and-conquer algorithm to multiply two dense matrices. Each matrix is split into four quadrants; the quadrants are multiplied recursively, and the resulting matrices are added to get the final result. Matrix addition is also implemented in a recursive, divide-and-conquer fashion. A new thread is forked to execute each recursive call. Serial matrix multiply is performed at the leaves of the recursion tree when the matrix size falls below a specified block size. Such hierarchical matrix multiply algorithms have been shown to achieve high performance due to their good caching behavior [1, 24, 43]. The medium grained version uses blocks of size 64×64, while the fine grained version uses 32×32 blocks.

3. **Sparse Matrix Vector Multiply.** This code to multiply a sparse, unsymmetric matrix with a dense vector was adapted from the Spark98 kernels [44]. The sparse matrix is generated from a finite element mesh used to simulate the motion of the ground after an earthquake in the San Fernando valley [5, 4]. Each thread calculates the product for a contiguous set of rows of the matrix; with a sufficiently large number of threads, the load is automatically balanced. Since the rows of the matrix are ordered by a graph partitioner, neighboring rows may access common

[15]The exception was the dense matrix multiply, which we wrote for $n \times n$ blocks, where n is a power of two. Therefore, fine granularity involved reducing the block size by a factor of 4, and increasing the number of threads by a factor of 8, resulting in 10% additional overhead.

data. At medium granularity, 64 threads are forked to perform the multiplication, while 256 threads are forked at fine granularity.

4. **Fast Fourier Transform.** We used the Pthreads-based code from the FFTW library [25], which is typically faster than all other publicly available code to compute one- and multidimensional complex discrete Fourier transforms (DFTs). The code implements the Cooley-Tukey algorithm [19]; a new thread is forked to perform each recursive transform until a user-specified number of threads have been created. We use 64 threads to compute the 1D FFT at medium granularity, and 512 threads at fine granularity.

5. **Fast Multipole Method.** This multithreaded code implements the uniform FMM [28], an N-body algorithm that calculates forces between N bodies in $O(N)$ time. Although every phase in the computation was parallelized, we only varied the granularity of the most time consuming phase, the top-down traversal. For each cell, we fork a separate thread to calculate its interaction with a fixed number of its neighbors (the cells on its interaction list). At medium granularity, each thread calculates 50 interactions of a cell with its neighboring cells, while at fine granularity, each thread computes 5 such interactions.

6. **Barnes-Hut.** The code for this $O(N \log N)$ N-body algorithm [6] was adapted from the Splash-2 benchmark suite [56]. All the original load balancing code was removed since our simpler, rewritten version is automatically load balanced by the Pthreads library. As with FMM, we varied the granularity within the most time consuming phase, which is the force calculation. We rewrote this phase to recursively traverse down the octree by forking a new thread for each subtree, and terminating the forking after a fixed number of levels. After that, the force on all the particles in each subtree is calculated by a separate thread. Since particles close together in the tree are likely to require common data for force calculation, this provides good locality. The granularity is adjusted by varying the cut-off level at which parallel recursion is terminated. At medium granularity, each thread computes the forces on particles in up to 4 leaf cells (that is, each subtree contains 4 leaves on average), while at fine granularity, each thread handles one leaf cell.

7. **Decision Tree Builder.** This data classification program implements a top-down, divide-and-conquer tree building algorithm ID3 [47], with C4.5-like additions to handle continuous attributes [48]. A new thread is forked to execute each recursive call. The resulting divide-and-conquer dag is highly irregular and data dependent, where each stage of the recursion itself involves a parallel divide-and-conquer quicksort to split the instances. We used a speech recognition dataset with 4 continuous attributes and a true/false classification as the input. The thread granularity is adjusted by setting a threshold for the number of instances, below which the recursion is executed serially. The thresholds were set to 2000 and 200 for medium and fine granularities, respectively.

Figure 11 lists the total number of threads expressed in each benchmark at both the thread granularities.

5.2 Comparison results

In all the comparison results, we use a memory threshold of $K = 50,000$ bytes for "ADF" and "DFD"[16]. Each active thread is allocated a minimum 8kB (1 page) stack. Therefore, the space-efficient schedulers effectively conserve stack memory by creating fewer simultaneously active threads compared to the original FIFO scheduler (see Figure 11). The FIFO scheduler spends significant portions of time executing system calls related to memory allocation for the thread stacks [42]; this problem is aggravated when the threads are made fine grained.

The 8-processor speedups for all the benchmarks at medium and fine thread granularities are shown in Figure 12. To concentrate on the effect of the scheduler, and to ignore the effect of increased thread overheads (up to 5% for all except dense matrix multiply) at the fine granularity, speedups for each thread granularity are with respect to the single-processor multithreaded execution at that granularity. The speedups show that both the depth-first scheduler and the new *DFDeques* scheduler outperform the library's original FIFO scheduler. However, at the fine thread granularity, the new scheduler provides better performance than the depth-first scheduler. This difference can be explained by the better locality and lower scheduling contention experienced by algorithm *DFDeques*.

We measured the external (L2) cache miss rates for each benchmark using on-chip UltraSPARC performance counters. Figure 1, which lists the results at the fine thread granularity, shows that our scheduler achieves relatively low cache miss rates (i.e., results in better locality).

Three out of the seven benchmarks make significant use of heap memory. For these benchmarks, we measured the high water mark for heap memory allocation using the three schedulers. Figure 14 shows that algorithm *DFDeques* results in slightly higher heap memory requirement compared to the depth-first scheduler, but still outperforms the original FIFO scheduler.

The Cilk runtime system [26] uses a provably space-efficient work stealing algorithm to schedule threads[17]. Figure 13 compares the space performance of Cilk with the depth-first and *DFDeques* schedulers for the dense matrix multiply benchmark (at the fine thread granularity). The figure indicates that *DFDeques* requires more memory than the depth-first scheduler, but less memory than Cilk. In particular, similar to the depth-first scheduler, the memory requirement of *DFDeques* increases slowly with the number of processors.

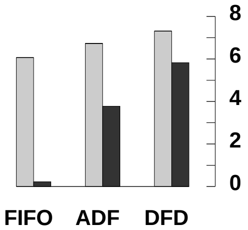
5.3 Measuring the tradeoff between space, time, and scheduling granularity

We studied the effect of the size of memory threshold K on the running time, memory requirement, and scheduling granularity using *DFDeques*(K). Each processor keeps track of the number of times a thread from its own deque is scheduled, and the number of times it has to perform a steal. The ratio of these two counts, averaged over all the processors, is our approximation of the scheduling granularity. The trade-off is best illustrated in the dense matrix multiply benchmark, which allocates significant amounts of heap memory. Figure 15 shows the resulting trade-off for

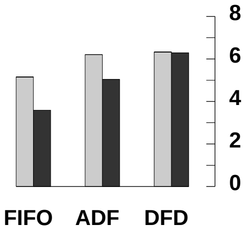
[16] In the depth-first scheduler, the memory threshold K is the memory quota assigned to each thread between thread preemptions [42].

[17] Because Cilk requires `gcc` to compile the benchmarks (which results in slower code for floating point operations compared to the native `cc` compiler on UltraSPARCs), we do not show a direct comparison of running times or speedups of Cilk benchmarks with our Pthreads-based system here.

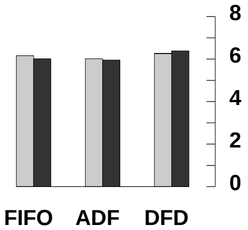
(a) Volume Rendering



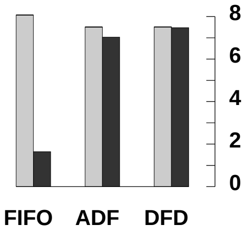
(b) Dense Matrix Multiply



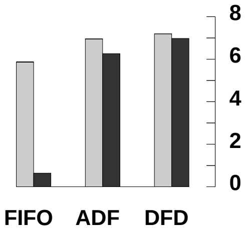
(c) Sparse Matrix Multiply



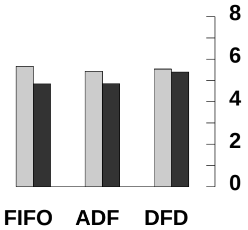
(d) Fast Fourier Transform



(e) Fast Multipole Method



(f) Barnes Hut



(g) Decision Tree Builder

Figure 12: Speedups on 8 processors with respect to single-processor executions for the three schedulers (the original "FIFO", the depth-first "ADF", and the new "DFD" or *DFDeques*) at both medium and fine thread granularities, with K = 50,000 bytes. Performance of "DFD-inf" (or *DFDeques*(∞)), being very similar to that of "DFD", is not shown here. All benchmarks were compiled using `cc -fast -xarch=v8plusa -xchip=ultra -xtarget=native -xO4`.

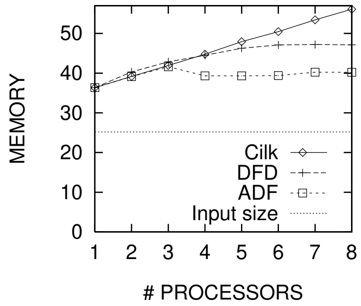
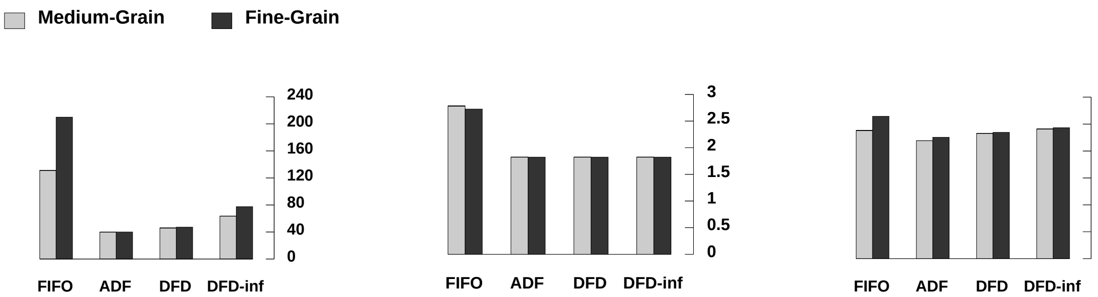
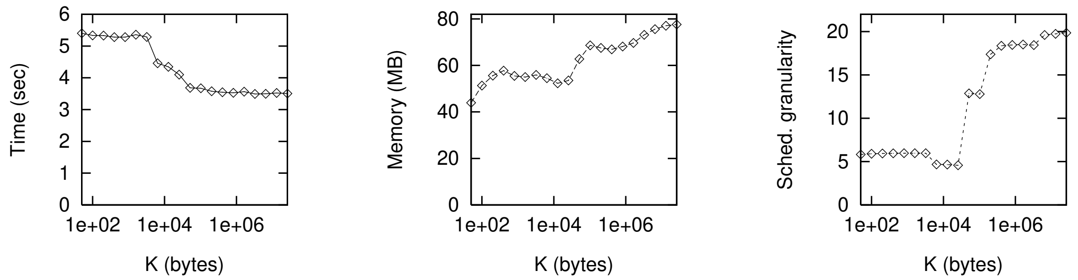


Figure 13: Variation of the memory requirement with the number of processors for dense matrix multiply using three schedulers: depth-first (“ADF”), *DFDeques* (“DFD”), and Cilk (“Cilk”).



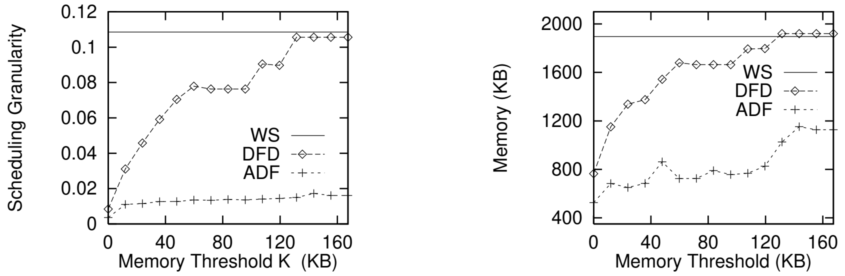
(a) Dense Matrix Multiply (b) Fast Multipole Method (c) Decision Tree Builder

Figure 14: High water mark of heap memory allocation (in MB) on 8 processors for benchmarks involving dynamic memory allocation (K = 50,000 bytes for “ADF” and “DFD”), at both thread granularities. “DFD-inf” is our approximation of work stealing using *DFDeques*(∞).



(a) Running time (b) Memory Allocation (c) Scheduling granularity

Figure 15: Trade-off between running time, memory allocation and scheduling granularity using algorithm *DFDeques* as the memory threshold K is varied, for the dense matrix multiply benchmark at fine thread granularity.



(a) Scheduling granularity (b) Memory

Figure 16: Simulation results for a divide-and-conquer benchmark with 15 levels of recursion running on 64 processors. The memory requirement and thread granularity decrease geometrically (by a factor of 2) down the recursion tree. Scheduling granularity is shown as a percentage of the total work in the dag. "WS" is the space-efficient work-stealing scheduler, "ADF" is the space-efficient depth-first scheduler, and "DFD" is our new *DFDeques* scheduler.

this benchmark at the fine thread granularity. As expected, both memory and scheduling granularity increase with K, while running time reduces as K is increased.

6 Simulating the schedulers

To compare algorithm *DFDeques* with a work-stealing scheduler, we built a simple system that simulates the parallel execution of synthetic, nested-parallel, divide-and-conquer benchmarks[18]. Our implementation simulates the execution of the space-efficient work-stealing scheduler [13] (labeled "WS"), the space-efficient, asynchronous depth-first scheduler [41] ("ADF"), and our new *DFDeques* scheduler (labeled "DFD").

We present results for one of the synthetic benchmarks here[19], in which both the memory requirement and the thread granularity decrease geometrically down the recursion tree. A number of divide-and-conquer programs exhibit such properties. Scheduling granularity was measured as the average number of actions executed by a processor between two steals. Figure 16 shows that work stealing results in high scheduling granularity and high space requirement, the depth first scheduler results in low scheduling granularity and low space requirement, while *DFDeques* allows scheduling granularity to be traded with space requirement by varying the memory threshold K.

7 Summary and Discussion

Depth-first schedulers are space-efficient, but unlike work-stealing schedulers, they require the user to explicitly increase the thread granularity beyond what is required to amortize basic thread

[18]To model irregular applications, the space and time requirements of a thread at each level of the recursion are selected uniformly at random with the specified mean.

[19]Results for other benchmarks and a detailed description of the simulator can be found in the author's dissertation [40].

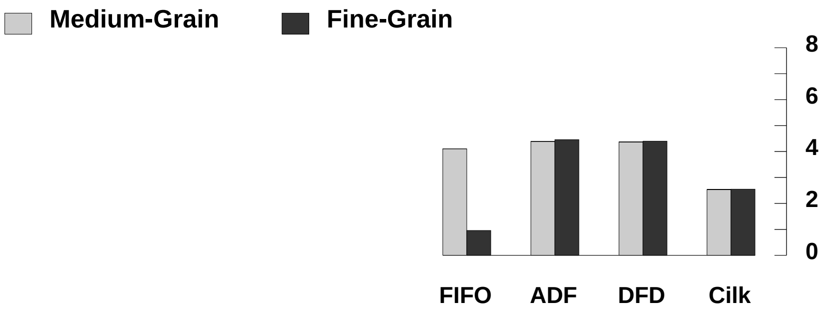


Figure 17: Speedups for the tree-building phase of Barnes Hut (for 1M particles). The phase involves extensive use of locks on cells of the tree to ensure mutual exclusion. The Pthreads-based schedulers (all except Cilk) support blocking locks. "DFD" does not result in a large scheduling granularity due to frequent suspension of the threads on locks; therefore, its performance is similar to that of "ADF". Cilk [26] uses a pure work stealer and supports spin waiting locks. For this benchmark, the single-processor execution time on Cilk is comparable with that on the Pthreads-based system.

costs. In contrast, algorithm *DFDeques* automatically increases the scheduling granularity by executing neighboring, fine-grained threads on the same processor to yield good locality and low scheduling contention. In theory, for nested-parallel programs with a large amount of parallelism, algorithm *DFDeques* has a lower space bound than work-stealing schedulers. We showed that in practice, it requires more memory than a depth-first scheduler, and less memory than work stealing. *DFDeques* also allows the user to control the trade-off between space requirement and running time (or scheduling granularity). Because algorithm *DFDeques* allows more deques than processors, it can be easily extended to support blocking synchronizations. For example, experiments with a benchmark that makes a significant use of locks indicate that *DFDeques* with blocking locks results in better performance than a work stealer that uses spin-waiting locks (see Figure 17).

Since Pthreads are not very lightweight, serializing access to the set of ready threads $\mathcal{R}$ did not significantly affect the performance in our implementation. However, serial access to $\mathcal{R}$ can become a bottleneck if threads are extremely fine grained, and require frequent suspension due to memory allocation or synchronization. To support such threads, the scheduling operations (such as updates to $\mathcal{R}$) need to be parallelized [40].

Each processor in *DFDeques* treats its deque as a regular stack. Therefore, in a system that supports very lightweight threads, the algorithm should benefit from stack-based optimizations such as lazy thread creation [27, 38]; these methods avoid allocating resources for a thread unless it is stolen, thereby making most thread creations nearly as cheap as function calls.

Increasing scheduling granularity typically serves to enhance data locality on SMPs with limited-size, hardware-coherent caches. However, on distributed memory machines (or software-coherent clusters), executing threads where the data permanently resides becomes important. A multi-level scheduling strategy may allow the thread implementation to scale to clusters of SMPs. For example, the *DFDeques* algorithm could be deployed within a single SMP, while some scheme based on data affinity is used across SMPs.

An open question is how to automatically find the appropriate value of the memory threshold K, which may depend on the benchmark, and on the thread implementation. One possible solution is for the user (or the runtime system) to set K to an appropriate value after running the program for a range of values of K on smaller input sizes. Alternatively, it may be possible for the system to keep statistics to dynamically adjust K to an appropriate value during the execution.

Acknowledgements

This research was conducted while the author was at Carnegie Mellon University, and was supported by ARPA Contract No. DABT63-96-C-0071. Guy Blelloch, Robert Blumofe, Bwolen Yang, and the anonymous referees provided valuable feedback on previous versions of this paper. We also thank Adam Kalai and Avrim Blum for useful discussions.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, New York City, NY, May 1987.

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Performance Evaluation Review*, 17:49–60, May 1989.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM symp. Parallel Algorithms and Architectures*, 1998.

[4] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and Jifeng Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, January 1998.

[5] H. Bao, J. Bielak, O. Ghattas, D. R. O'Hallaron, L. F. Kallivokas, J. R. Shewchuk, and J. Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96*, November 1996.

[6] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[7] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *J. Parallel and Distributed Computing*, 37(1):113–121, August 1996.

[8] G. Blelloch, P. Gibbons, Y. Matias, and G. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 12–23, 1997.

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

[10] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. ACM symp. Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 17–19, 1995.

[11] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Par. and Distr. Computing*, 37(1):55–69, August 1996.

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

[14] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[15] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.

[16] R. Chandra, A. Gupta, and J. L. Hennessy. Data locality and load balancing in COOL. In *Proc. ACM symp. Principles & Practice of Parallel Programming*, pages 239–259, 1993.

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

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

[19] J. W. Cooley and J. W Tukey. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation*, 19:297–301, Apr. 1965.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[21] D. E. Culler and G. Arvind. Resource requirements of dataflow programs. In *Proc. Intl. Symp. on Computer Architecture*, pages 141–151, 1988.

[22] D. R. Engler, G. R. Andrews, and D. K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report 93-13, University of Arizona. Dept. of Computer Science, 1993.

[23] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation (extended abstract). In *ACM Symp. Parallel Algorithms and Architectures*, pages 94–103, 1994.

[24] Frigo, Leiserson, Prokop, and Ramachandran. Cache-oblivious algorithms. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[25] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[26] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 212–223, 1998.

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

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

[29] High Performance Fortran Forum. High performance fortran language specification vertion 1.0, 1993.

[30] IEEE. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. IEEE/ANSI Std 1003.1, 1996 Edition.

[31] V. Karamcheti, J. Plevyak, and A. A. Chien. Runtime mechanisms for efficient dynamic multithreading. *J. Parallel and Distributed Computing*, 37(1):21–40, August 1996.

[32] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proc. Symp. Theory of Computing*, pages 290–300, 1988.

[33] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proc. Programming Language Design and Implementation*, Portland, Oregon, June 21–23, 1989.

[34] E. P. Markatos and T. J. LeBlanc. Locality-based scheduling in shared-memory multiprocessors. Technical Report 94, Inst for ICS-FORTH, Heraklio, Crete, Greec, 1993.

[35] Evangelos Markatos and Thomas LeBlanc. Locality-based scheduling in shared-memory multiprocessors. Technical Report TR93-0094, ICS-FORTH, Heraklio, Crete, Greece, 1993.

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

[37] T. Miyazaki, C. Sakamoto, M. Kuwayama, L. Saisho, and A. Fukuda. Parallel pthread library (PPL): user-level thread library with parallelism and portability. In *Proc. Intl. Computer Software and Applications Conf. (COMPSAC)*, pages 301–306, November 1994.

[38] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 1990.

[39] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[40] G. J. Narlikar. *Space-Efficient Scheduling for Parallel, Multithreaded Computations*. PhD thesis, Carnegie Mellon University, 1999. Available as CMU-CS-99-119.

[41] G. J. Narlikar and G. E. Blelloch. Space-efficient implementation of nested parallelism. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pages 25–36, June 1997.

[42] G. J. Narlikar and G. E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proc. of Supercomputing '98*, November 1998.

[43] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic parallelism. Technical Report CMU-CS-98-114, Computer Science Dept., Carnegie Mellon University, April 1998.

[44] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, 1997.

[45] J. Philbin, J. E., O. J. Anshus, and C. C. Douglas. Thread scheduling for cache locality. In *Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[46] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proc. Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, January 1991.

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

[48] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

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

[50] C. A. Ruggiero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, Berlin, DE, 1987.

[51] D. J. Simpson and F. W. Burton. Space efficient execution of deterministic parallel programs. *IEEE Transactions on Software Engineering*, 25(3), May/June 1999.

[52] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[53] D. Stein and D. Shah. Implementing lightweight threads. In *Proc. Summer 1992 USENIX Technical Conference and Exhibition*, pages 1–10, San Antonio, TX, 1992. USENIX.

[54] M. T. Vandevoorde and E. S. Roberts. WorkCrews: an abstraction for controlling parallelism. *Intl. J. Parallel Programming*, 17(4):347–366, August 1988.

[55] B. Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1998.

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proc. Intl. Symp. Computer Architecture*, pages 24–37, June 1995.