Space-efficient Scheduling for Parallel, Multithreaded Computations

Girija Narlikar

May 1999
CMU-CS-99-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Guy Blelloch, Chair
Thomas Gross
Bruce Maggs
Charles Leiserson, Massachusetts Institute of Technology

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and by the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and DABT63-96-C-0071. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARPA or the U.S. Government.

Keywords: Mulththreading, dynamic parallelism, dynamic scheduling, space efficiency, parallel runtime systems, nested parallelism

Abstract

The goal of high-level parallel programming models or languages is to facilitate the writing of well-structured, simple and portable code. However, the performance of a program written using a high-level language may vary significantly, depending on the implementation of the underlying system.

This dissertation presents two asynchronous scheduling algorithms that provide worst-case upper bounds on the space and time requirements of high-level, nested-parallel programs on shared memory machines. In particular, for a program with D depth and a serial space requirement of S_1, both algorithms guarantee a space bound of $S_1 + O(K \cdot p \cdot D)$ on p processors. Here, K is a user-controllable runtime parameter, which specifies the amount of memory a thread may allocate before being preempted by the scheduler. Typically, in practice, K is fixed to be a few thousand bytes. Most parallel programs have a small depth D. For such programs, the above space bound is lower than the space bound provided by any previously implemented system.

The first of the two scheduling algorithms presented in this dissertation, algorithm *AsyncDF*, prioritizes threads at runtime by their serial, depth-first execution order, and preempts threads before they allocate too much memory. This ensures that the parallel execution does not require too much more memory than the serial execution. The second scheduling algorithm, *DFDeques*, enhances algorithm *AsyncDF* by adding ideas from work stealing. It replaces the single, flat priority queue of *AsyncDF* with ordered, per-processor queues of ready threads, and allows some deviation from the depth-first priorities while scheduling threads. Consequently, it results in lower scheduling overheads and better locality than *AsyncDF* for finer-grained threads, at the cost of a slight increase in memory requirement. To ensure scalability with the number of processors, I also describe and analyze fully parallelized versions of the schedulers for both the algorithms.

To verify that the theoretically-efficient scheduling algorithms are also practical, I have implemented them in the context of two multithreaded runtime systems, including a commercial Pthreads package. Parallel benchmarks used to evaluate the schedulers include numerical codes, physical simulations and a data classifier. Experimental results indicate that my algorithms effectively reduce memory usage compared to previous techniques, without compromising time performance. In particular, my schedulers allow simple, high-level, fine-grained benchmarks to run as efficiently as their more complex, hand-partitioned, coarse-grained counterparts. As expected, *DFDeques* achieves better speedups than *AsyncDF* for finer-grained threads. It requires more memory than *AsyncDF*, but less memory compared to previous schedulers. The scheduling algorithms were extended to support the full Pthreads interface, making them available to a large class of applications with irregular and dynamic parallelism. Both the *AsyncDF* and *DFDeques* algorithms provide a user-adjustable trade-off between running time and memory requirement, which I analyze and experimentally demonstrate.

Acknowledgements

First, I would like to thank my advisor Guy Blelloch, for his constant encouragement and guidance. He was always available to patiently listen to half-baked ideas and to provide clever suggestions when I was stuck, and often amazed me with his immediate grasp of the problem at hand. His stress on the importance of both theory and practice have, hopefully, rubbed off a bit on me.

I thank the other members of my thesis committee, Thomas Gross, Charles Leiserson, and Bruce Maggs, for several useful discussions. They also provided some excellent feedback on my writing and presentation skills. Sharon Burks, Catherine Copetas, Cleah Schlueter, and Karen Olack were always willing to help with any administrative problems; I am grateful for their patience and good humor. The Berkeley NOW and Clumps projects generously provided access to their machines for many of my experiments. My interactions over two summers with Satish Rao and Chandu Thekkath provided me with fun opportunities to learn more about distributed systems.

Being the most junior member of the SCANDAL project, I received a good amount of useful advice and technical knowhow from the other members: John Greiner, Jonathan Hardwick, Margaret Miller, Jay Sipelstein, and Marco Zagha. Other SCS friends were also fine company over the years: Aarti, Arup, Bwolen, Claudson, Chen, Chris, Dafna, Daniel, Hao-Chi, Howard, Marius, Peter, Prem, Sean, and others whom I must have accidently forgotten (it's been nearly six long years!). Joyoni, Madhavi, Rohini, Snehal, and the SPIC-MACAY folks provided welcome diversions from the thesis. Alan, Andrea, Kaustubh, Mrinalini, Puskhar, Rajeshwari, Remzi, Steve, and Subarna helped me feel at home during the one year I spent visiting Berkeley. Also, a big `zephyr++`; although I could rarely keep up with the discussions, the logs were very useful.

Most important, I'd like to thank my family for their constant love and encouragement throughout my education. My parents (I call them *Aai* and *Baba*) have always been an inspiration to me. Geeky as it sounds, they were the first to show me the fun in mathematics, which eventually got me interested in computer science. My older sister Geeta has been a steady source of support, and my younger sister Leelavati, one of entertainment. Finally, a big thanks to Rahul, my soon-to-be husband, who has been a very patient "thesis widower". His love, sensitivity, humor, support, and ability to shake me out of a slump (often through hours on the phone) have helped me get through graduate school, and even made it a very enjoyable experience!

Contents

List of Figures

" . . . a normal adult never stops to think about problems of space and time. These are things which he has thought about as a child. But my intellectual development was retarded, as a result of which I began to wonder about space and time only when I had already grown up."

—Albert Einstein

Chapter 1

Introduction

Parallel machines are fast becoming affordable; in fact, low-end symmetric multiprocessors (SMPs) are even appearing on people's desktops today. However, the inherent complexity of writing parallel programs makes the costs of generating parallel software for these machines prohibitive. Consequently, the SMPs sold today are often restricted to running independent, short jobs on individual processors (*e.g.*, database transactions or web searches), rather than a large parallel job that can make effective use of multiple processors. If parallel programming is to become more prevalent in the future, it is critical to allow parallelism to be expressed in a simple, well-structured, and portable manner. Many of today's high-level parallel programming languages attempt to meet these goals by providing constructs to express dynamic, fine-grained parallelism. Such languages include data-parallel languages such as NESL [17] and HPF [60], as well as control-parallel languages such as ID [5], Cool [36], Cilk [25], CC++ [37], Sisal [59], Multilisp [77], Proteus [109], and C or C++ with lightweight thread libraries [15, 113, 129].

Languages with constructs for dynamic, fine-grained parallelism are particularly useful for expressing applications with irregular, data-dependent parallelism. Such applications include physical simulations on non-uniform inputs, data classifiers, sparse-matrix operations, computational geometry codes, and a large number of divide-and-conquer algorithms with irregular, data-dependent recursion trees. The user simply exposes all the parallelism in the application, which is typically of a much higher degree than the number of processors. The language implementation is responsible for scheduling this parallelism onto the processors. Because static, compile-time partitioning of such programs is generally not possible, the parallel tasks are scheduled onto the processors at runtime. Consequently, the performance of a program depends significantly on the implementation of the runtime system. A number of previous implementations have focused on efficiently balancing the load and providing good data locality in such runtime systems [36, 39, 62, 70, 81, 83, 115, 134, 136]. However, in addition to good time performance, the memory requirements of the parallel computation must be taken into consideration. In particular, unless the scheduler is carefully implemented, a fine-grained parallel program can end up creating excessive amounts of active parallelism, leading to a huge space requirement [26, 48, 77, 114, 138]. The price of the memory is a significant portion of the price of a parallel computer, and parallel computers are typically used to run big problem sizes. Therefore, reducing the memory usage of a parallel program, that is, making the execution ***space efficient***, is often as important as reducing the running time. In fact, making a program more space-efficient often also improves its performance due to fewer page or TLB misses and fewer memory-related system calls.

This dissertation focuses on space- and time-efficient implementations of languages with dynamic, fine-grained parallelism. My thesis statement can be summarized as follows.

> **By utilizing provably-efficient, asynchronous scheduling techniques, it is possible to obtain good space and time performance in both theory and practice for high-level, nested parallel programs. These scheduling techniques can, in practice, be extended to general purpose multithreading systems like Pthreads, benefiting a large class of applications with irregular and dynamic parallelism.**

To validate the thesis, I present two asynchronous scheduling algorithms, *AsyncDF* and *DFDeques*, that guarantee worst-case bounds on the space and time required to execute a parallel program. Low space bounds are achieved by prioritizing parallel tasks by their serial execution order, and preempting or delaying tasks that allocate too much memory. In addition to theoretically analyzing the scheduling algorithms, I have implemented them in the context of multithreading runtime systems (including a commercial Pthreads [88] package) to schedule lightweight threads. Experimental results for a variety of parallel benchmarks indicate that the algorithms are effective in reducing space requirements compared to previous scheduling techniques, while maintaining good time performance. In fact, the use of these space-efficient schedulers allows simpler codes with fine-grained threads to achieve performance comparable to their coarse-grained, hand-partitioned counterparts.

This remainder of this chapter is organized as follows. It first presents two simple examples to demonstrate the impact a scheduler can have on the space requirement of a parallel program. The chapter then provides a synopsis of this dissertation, followed by a list of its limitations. It concludes by describing the organization of the remainder of this dissertation.

An example

Consider a very simple dense matrix multiplication program involving the creation of a large number of lightweight threads. The two input matrices, A and B, are multiplied using the divide-and-conquer algorithm shown in Figure 1.1. A new, lightweight child thread is forked to execute each recursive call; the eight recursive calls can execute in parallel[1]. Temporary storage T is allocated to store the results of each recursive call. At the end of the eight recursive calls (after all eight child threads have terminated), the intermediate results in T are added to the result matrix C, and T is deallocated.

This program was implemented using the native Pthreads library on Solaris 2.5; a new, user-level pthread was forked for each recursive call. The existing Solaris Pthreads package uses a simple FIFO (first-in-first-out) queue to store ready threads. I replaced the existing scheduler with my space-efficient scheduler; this modified implementation of the Pthreads library can be used by any existing Pthreads-based program[2]. Figure 1.2 shows the resulting space and time performance on an 8-processor Enterprise 5000 SMP using both the original FIFO scheduler, and the

[1]The child threads, in turn, fork their own child threads at the next level of recursion.

[2]I simply modified the implementation of `SCHED_OTHER`, which, according to the Pthreads standard [88], can be a system-dependent scheduling policy; FIFO and round robin are the other (fixed) policies in the Pthreads standard.

```
begin Matrix_Mult(A, B, C, n)
  if (n ≤ Leaf_Size)
    serial_mult(A, B, C, n);
  else
    T := mem_alloc(n × n);
    initialize smaller matrices as quadrants of A, B, C, and T;
    fork Matrix_Mult(A11, B11, C11, n/2);
    fork Matrix_Mult(A11, B12, C12, n/2);
    fork Matrix_Mult(A21, B12, C22, n/2);
    fork Matrix_Mult(A21, B11, C21, n/2);
    fork Matrix_Mult(A12, B21, T11, n/2);
    fork Matrix_Mult(A12, B22, T12, n/2);
    fork Matrix_Mult(A22, B22, T22, n/2);
    fork Matrix_Mult(A22, B21, T21, n/2);
    join with all forked child threads;
    Matrix_Add(T, C);
    mem_free(T);
end Matrix_Mult
```

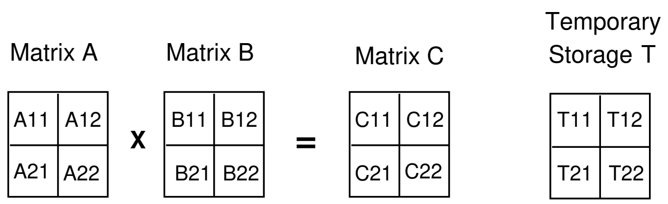


Figure 1.1 : Pseudocode to multiply two $n \times n$ matrices A and B and storing the final result in C using a divide-and-conquer algorithm. The Matrix_Add function is implemented similarly using a parallel divide-and-conquer algorithm. The constant Leaf_Size to check for the base condition of the recursion is set to 64 on a 167 MHz UltraSPARC.

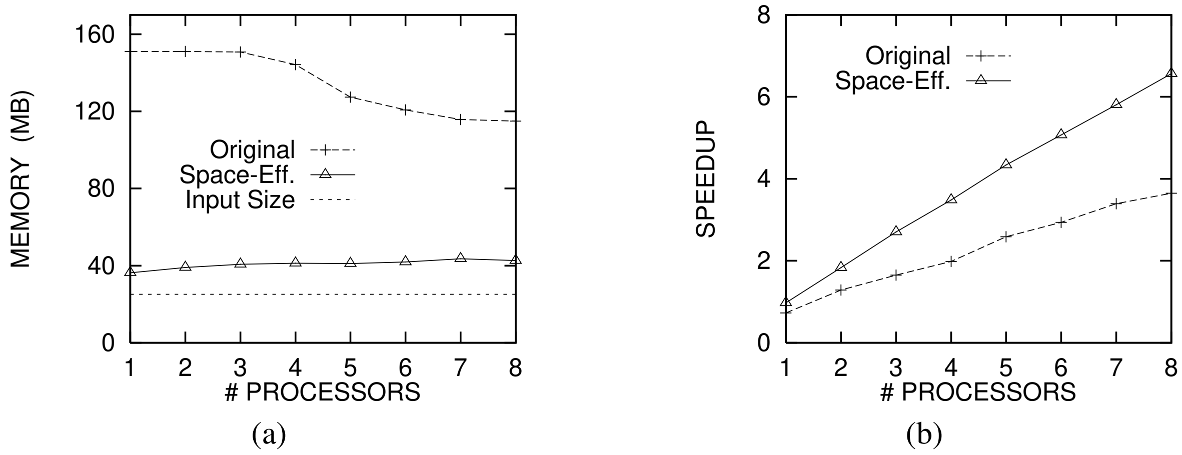


Figure 1.2: Space and time performance for the matrix multiply code shown in Figure 1.1. The Pthreads-based program was executed using both the original Pthreads implementation, and the modified implementation that uses my space-efficient scheduler. (a) The high-water mark of heap memory allocation. The input size is the size of the three matrices A, B, and C; the remaining memory allocated is the temporary storage. (b) The speedups with respect to a serial, C version of the program.

new, space-efficient scheduler[3]. The space-efficient scheduler used in this example is described in Chapter 3. Further details on the experiment, along with results for other, more complex and irregular applications, are presented in Chapter 5. The results in Figure 1.2 indicate that both the space and time performance is heavily dependent on the underlying thread scheduler. Unlike the space-efficient scheduler, the original LIFO queue results in excessive memory allocation (including both heap and stack space). This significantly reduces the speedup, due to high contention in system calls related to memory allocation (see Section 5.2). Even with a better, highly concurrent kernel implementation, space-inefficiency will limit the largest problem size can be run on a machine without paging. Thus, a space-efficient scheduler is required to get good performance in both space and time.

Another example

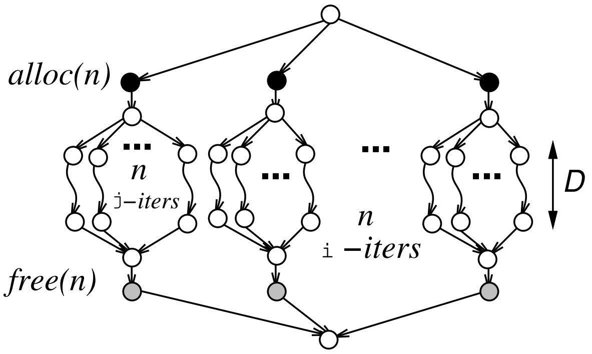
I now explain how the space-efficient scheduling techniques presented in this dissertation effectively reduce the high-water mark of memory requirement of the program, in comparison to previous schedulers. Consider the example code in Figure 1.3(a). The code has two levels of parallelism: the `i`−loop at the outer level and the `j`-loop at the inner level. In general, the number of iterations in each loop may not be known at compile time. Space for an array `B` is allocated at the start of each `i`-iteration, and is freed at the end of the iteration. Assuming that `F(B,i,j)` does not allocate any space, the most natural serial execution requires $O(n)$ space, since the space for array `B` is reused for each `i`-iteration. Figure 1.3(b) shows the corresponding computation graph for this example code; each node in the graph represents an instruction, while each edge represents

[3]The results for the space-efficient scheduler also include an optimization that caches a small number of thread stacks in the library, yielding an additional 13% speedup.

a dependence between two instructions. The serial execution of the code now corresponds to a depth-first execution of the computation graph.

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

(a)



(b)

Figure 1.3: (a) The pseudocode for a simple program with dynamic parallelism and dynamic memory allocation. (b) The computation graph that represents this program. For brevity, the curved lines are used to represent the multiple (D) instructions executed serially within each call to `F()`; the shaded nodes denote the memory allocations (black) and deallocations (grey).

Now consider the parallel implementation of this function on p processors, where $p < n$. If we use a simple FIFO queue to schedule parallel tasks (loop iterations in this case), all the `i`-iterations would first begin execution and allocate n space each, followed by the execution of their `j`-iterations. Thus, as much as $O(n^2)$ space would be allocated at some time during the execution; this space requirement is significantly higher than the serial space requirement of $O(n)$. Such a parallel schedule corresponds to a breadth-first execution of the computation graph. Several thread packages, such as the standard Pthreads [88] library, make use of such FIFO schedulers. A simple alternative for limiting the excess parallelism is to schedule the outer level of parallelism first, and create only as much parallelism as is required to keep processors busy. This results in each of the p processors executing one `i`-iteration at any time (the processor executes the inner `j`-loop serially), and hence the total space allocated is $O(p \cdot n)$. Several previous scheduling systems [26, 30, 41, 70, 77, 83, 143], which include both heuristic-based and provably space-efficient techniques, adopt such an approach.

This dissertation claims that even a linear expansion of space with processors (*i.e.*, p times the serial space requirement) is not always necessary to enable an efficient parallel execution. For example, the ***AsyncDF*** ("asynchronous, depth-first") algorithm presented in this dissertation also starts by scheduling the outer parallelism, but stalls big allocations of space. Moreover, it prioritizes operations by their serial (depth-first) execution order. As a result, the processors suspend the execution of their respective `i`-iterations before they allocate $O(n)$ space each, and execute `j`-iterations belonging to the first `i`-iteration instead. Thus, if each `i`-iteration has sufficient parallelism to keep the processors busy, our technique schedules iterations of a single `i`-loop at a time. This parallel execution order is closer to the serial, depth-first execution order of the computation graph. In general, our scheduler allows this parallel computation to run in just $O(n + p \cdot D)$ space[4], where D is the depth (critical path length) of the function `F`. For programs with a large degree of parallelism, the depth D is typically very small. (Computation graphs, their depth, and depth-first schedules are formally defined in Chapter 2.)

The *AsyncDF* algorithm presented in this dissertation often preferentially schedules inner parallelism (which is finer grained). Consequently, when the threads in the program are very fine grained, it can cause large scheduling overheads and poor locality compared to algorithms that schedule the outer parallelism. We therefore have to manually group fine-grained iterations of innermost loops into chunks to get good performance with *AsyncDF*. However, the second scheduling algorithm presented in this dissertation, ***DFDeques***, automatically and dynamically achieves this coarsening by scheduling multiple threads close in the computation graph on the same processor.

1.1 Synopsis of the Dissertation

This dissertation presents two asynchronous, provably space-efficient scheduling algorithms, and describes their implementations in the contexts of two multithreading runtime systems. Here, I first describe the programming model assumed in this dissertation, followed by an overview of each of the scheduling algorithms and the runtime systems.

1.1.1 Programming model

This dissertation assumes a shared memory programming model. All the experiments were conducted on shared-memory, symmetric multiprocessors (SMPs). Because of their favorable price-to-performance ratios, such machines are common today as both desktops and high-end servers. The underlying architecture may differ from machine to machine, but I assume that it provides a fast, hardware-coherent shared memory. The programmer views this memory as uniformly accessible, with little control on the explicit placement of data in processor caches. The analysis of the space requirements in this dissertation includes memory allocated both on the stack(s) and the shared heap; the model allows individual instructions to allocate and deallocate arbitrary amounts of memory.

The scheduling algorithms have been analyzed for purely nested parallel programs; these include programs with nested parallel loops or nested forks and joins. For instance, the programming examples in Figures 1.1 and 1.3 are nested parallel. The nested parallel model is described in more detail in Chapter 2. Although the theoretical analysis is restricted to nested parallel programs, the experiments with Pthreads described in Chapters 5 and 7 indicate that the scheduling algorithms can effectively execute more general styles of parallel programs, such as programs with mutexes and condition variables. This is the first space-efficient system that supports an interface as general as that of Pthreads.

In the remainder of this dissertation, we refer to any independent flow of control within a program as a ***thread***, irrespective of its duration. Then fine-grained parallelism in high-level languages

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

can be viewed in terms of such threads. Not all parallel languages that support fine-grained parallelism provide constructs to explicitly create threads in the traditional sense. For example, in High Performance Fortran (HPF), fine-grained parallelism can be exploited across elements of an array through high-level array operations. Similarly, NESL provides constructs to exploit parallelism over elements of a vector. Nonetheless, we view each independent flow of control in such languages as a separate thread, even if it is very fine grained. For example, consider an HPF statement such as `C = A + B` that performs as element-wise addition of two arrays `A` and `B` of the same size, and stores the result in `C`. This statement can be viewed as one that forks (creates), and subsequently joins (synchronizes) and terminates n fine-grained, parallel threads (where n is the size of arrays `A` and `B`); each thread simply adds the corresponding elements of arrays `A` and `B`. Similarly, each `i`-iteration and each `j`-iteration in Figure 1.3(a) is considered to be a separate thread. The underlying implementation of such threads may never create separate execution stacks or program counters to represent their states. This dissertation uses threads as a more abstract, high-level concept independent of the implementation strategy.

Advantages of fine-grained, dynamic threads

Despite the involved complexity, shared memory parallel programs today are often written in a coarse-grained style with a small number of threads, typically one per processor. In contrast, a fine-grained program expresses a large number of threads, where the number grows with the problem size, rather than the number of processors. In the extreme case, as in Figure 1.3(a), a separate thread may be created for each function call or each iteration of a parallel loop. Note that the use of the term "fine-grained" in this dissertation does not refer exclusively to this extreme case. For example, in Chapter 5, the term is used in the context of a general Posix threads (Pthreads) package which explicitly allocates resources such as a stack and register state for each thread expressed. Hence basic operations on these threads (such as thread creation or termination) are significantly more expensive than function calls. Therefore, using them in the extremely fine-grained style is impractical; however, they can be fine-grained enough to allow the creation of a large number of threads, such that each thread performs (on average) sufficient work to amortize its costs. For example, if the `j`-iterations in Figure 1.3(a) are of extremely short duration, a fine-grained, Pthreads version of the program would require each Pthread to execute multiple `j`-iterations.
The advantages of fine-grained threads are summarized below.

- **Simplicity.** The programmer can create a new thread for each parallel task (or each small set of parallel tasks), without explicitly mapping the threads to processors. This results in a simpler, more natural programming style, particularly for programs with irregular and dynamic parallelism. The implicit parallelism in functional languages, or the loop parallelism extracted by parallelizing compilers is fine grained, and can be more naturally expressed as lightweight threads.
- **Architecture independence.** The resulting program is architecture independent, since the parallelism is not statically mapped to a fixed number of processors. This is particularly useful in a multiprogramming environment, where the number of processors available to the computation may vary over the course of its execution [3, 156].
- **Load balance.** Since the number of threads expressed is much larger than the number of processors, the load can be transparently and effectively balanced by the implementation.

The programmer does not need to implement a load balancing strategy for every application that cannot be statically partitioned.

- **Flexibility.** Lightweight threads in a language or thread library are typically implemented at the user level. Therefore, the implementation can provide a number of alternate scheduling techniques, independent of the kernel scheduler. Modifying the execution order of individual parallel tasks in a fine-grained program may then simply involve modifying user-assigned priorities or switching between schedulers for the corresponding threads. In contrast, since the execution order of tasks is explicitly coded in a coarse-grained program, changing it may involve extensive modification to the program itself.

1.1.2 Space-efficient scheduling algorithms

Although the fine-grained threads model simplifies the task of the programmer, he or she relies heavily on the underlying thread implementation to deliver good performance. In particular, the implementation must use an efficient scheduler to map threads to processors at runtime. This dissertation presents an two dynamic, asynchronous scheduling algorithms, *AsyncDF* and *DFDeques*, that are provably efficient in terms of both space and time.

Algorithm *AsyncDF* is an asynchronous and practical variant of the synchronous scheduling algorithm proposed in previous work [21]. The main goal in the design of algorithm *AsyncDF* was to maintain space-efficiency while allowing threads to execute non-preemptively and asynchronously, leading to good locality and low scheduling overheads. The algorithm maintains the threads in a shared work queue; threads in the queue are prioritized by their serial, depth-first execution order. Further, threads that perform large allocations are stalled to allow higher priority threads to execute instead. Each thread is assigned a fixed-size memory quota of K units (which we call the ***memory threshold***) every time it is scheduled. The thread is preempted when it exhausts its memory quota and reaches an instruction requiring more memory. These basic ideas ensure that the execution order of threads, and hence also the memory requirement of the parallel program, are sufficiently close to the execution order and memory requirements, respectively, of the serial, depth-first schedule. In particular, algorithm *AsyncDF* guarantees that a parallel computation with a serial, depth-first space requirement of S_1 and depth D can be executed on p processors using $S_1 + O(K \cdot p \cdot D)$ space. Most parallel programs have a small depth, since they have a high degree of parallelism. For example, a simple algorithm to multiply two $n \times n$ matrices has depth $D = \Theta(\log n)$ and serial space requirement $S_1 = \Theta(n^2)$. Further, in practice, the memory threshold K is fixed to be a small constant amount of memory. Therefore, the space bound provided by algorithm *AsyncDF* is asymptotically lower than the space bound of $p \cdot S_1$ guaranteed by previous asynchronous, space-efficient schedulers [24, 26, 30, 143].

Although algorithm *AsyncDF* provides non-blocking access to the shared queue of ready threads, the queue can become a bottleneck if the threads are very fine grained. Further, even though a single thread may execute on the same processor for a long period of time, fine-grained threads close together in the computation graph may get executed on different processors. Such threads often access the same data, and therefore, should ideally be executed in quick succession on the same processor to obtain good data locality. Therefore, fine-grained threads in *AsyncDF* have to be manually chunked to get good time performance[5]. The second scheduling algorithm,

[5] Alternatively, compiler support or a dynamic chunking scheme [84, 128, 158] would be required.

DFDeques, was designed to overcome this drawback of *AsyncDF*.

Algorithm *DFDeques* uses a hybrid approach that combines ideas from *AsyncDF* with ideas from work stealing [24, 31, 77, 92, 95, 137]. The aim in designing *DFDeques* was to allow dynamic clustering of fine-grained threads close together in the computation graph into a single scheduling unit that is executed on one processor. This clustering is achieved by storing threads in a queue of subqueues (deques); each processor owns a unique subqueue at any given time. Similar to *AsyncDF*, the subqueues are prioritized by the serial execution orders of their threads, and a processor gives up its subqueue when it exhausts its memory quota. However, unlike in *AsyncDF*, the memory quota can be used for multiple threads close together in the computation, leading to better locality. When a processor finds its subqueue of ready threads empty (or when it gives up its subqueue), it obtains new work by selecting a subqueue at random from a set of high-priority subqueues. It steals the lowest priority thread from the selected subqueue, which is typically of the coarsest granularity. Common operations such as thread creation and finding a ready thread typically involve access to the processor's own subqueue, resulting in lower scheduling overheads. Algorithm *DFDeques* guarantees the same space bound of $S_1 + O(K \cdot p \cdot D)$ as algorithm *AsyncDF*, but, in practice, results in a slightly higher space requirement compared to *AsyncDF*.

1.1.3 Multithreaded runtime systems

Besides presenting space-efficient scheduling algorithms, this dissertation describes efficient implementations of runtime systems that use the algorithms to schedule parallel threads. I have implemented a lightweight, multithreaded runtime system on the SGI Power Challenge, specifically to evaluate algorithm *AsyncDF*. I present the results of executing a set of parallel benchmarks on this system, and compare their space and time requirements with previous space-efficient systems such as Cilk [25]. The results indicate that algorithm *AsyncDF* provides equivalent time performance as previous systems. However, as shown in Figure 1.4, algorithm *AsyncDF* is effective in lowering the space requirements of the benchmarks in comparison to previous schedulers.

Posix standard threads or Pthreads [88] have become a popular standard for shared memory parallel programming. However, despite providing lightweight thread operations, existing Pthread implementations do not handle fine-grained, dynamic parallelism efficiently. In particular, current Pthread schedulers are not space-efficient due to their use of FIFO scheduling queues. Consequently, they result in poor space and time performance for programs with a large number of fine-grained threads. Therefore, I added both my scheduling algorithms, *AsyncDF* and *DFDeques*, to a popular, commercial Pthreads package, and evaluated the performance using a set of parallel benchmarks. The benchmarks include a variety of numerical codes, physical simulations, and a data classifier. Figure 1.5 highlights some of the results of implementing *AsyncDF* in the context of the Pthreads package; it shows the speedups for fine-grained and coarse-grained versions of a subset of the Pthreads benchmarks. The results indicate that the use of lightweight Pthreads can allow moderately fine-grained programs with simpler code to perform as well as their hand-partitioned, coarse-grained counterparts, provided a space-efficient scheduler is utilized. The results of using algorithm *DFDeques* for finer-grained Pthreads benchmarks are summarized in Figure 1.6. Unlike *AsyncDF*, algorithm *DFDeques* maintains high performance and good locality even when the granularity of the threads is reduced further (*i.e.*, when the threads are made more fine grained). As shown in the figure, the performance of the original FIFO scheduler deterio-

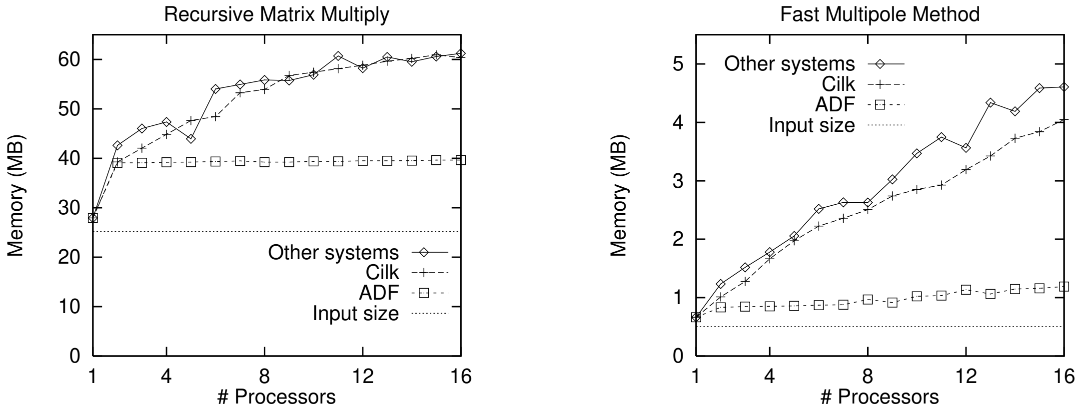


Figure 1.4: High-water mark of memory allocation for two multithreaded parallel benchmarks on a 16-processor SGI Power Challenge. The memory usage shown for one processor is the memory requirement of the serial C version of the program. "Other systems" is an estimate of the memory required by the benchmark on previous systems that schedule the outer parallelism with higher priority (*e.g.*, [41, 83]). "Cilk" is the memory requirement using the space-efficient system Cilk [25], and "ADF" is the memory requirement using algorithm *AsyncDF*. Results for other benchmarks can be found in Chapter 5.

rates significantly as the thread granularity is reduced. Both the space-efficient schedulers create fewer simultaneously active threads compared to the FIFO scheduler, thereby conserving expensive resources such as thread stacks. The benchmark inputs and thread granularities used in these experiments, along with other experimental results, are described in detail in Chapters 5 and 7.

Space-Time tradeoff

Recall that both the scheduling algorithms, *AsyncDF* and *DFDeques*, utilize a memory threshold K, so that threads can be preempted before they allocate too much (*i.e.*, more that K) space. This parameter is designed to be specified by the user at the command line. A bigger value of K leads to a lower running time in practice because it allows threads to run longer without preemption and reduces scheduling costs. However, a larger value of K results in a higher space bound. Therefore, adjusting the value of the memory threshold K provides a trade-off between the running time and the memory requirement of a parallel computation. For example, Figure 1.7 experimentally demonstrates this trade-off for a parallel benchmark using algorithm *AsyncDF* to schedule threads on the SGI Power Challenge.

In *DFDeques*, increasing the memory threshold K also allows more threads close together in the computation graph to execute consecutively on the same processor, leading to lower scheduling overheads and better locality. We will refer to the number of threads executed on a processor from its local subqueue (without accessing the global queue of subqueues) as the scheduling granularity; a higher scheduling granularity typically results in better locality and lower scheduling overheads. As shown in figure 1.8, adjusting the value of the memory threshold K in *DFDeques* provides a trade-off between the memory requirement and scheduling granularity (and running time).

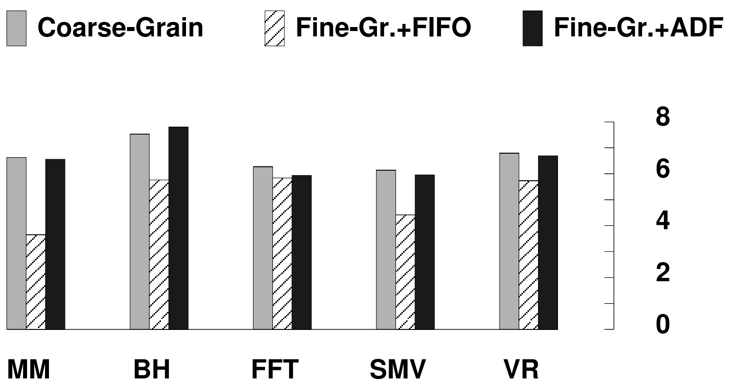


Figure 1.5: Speedups on 8 processors of an Enterprise 5000 SMP for 5 Pthreads-based applications. The speedup for each benchmark is with respect to its serial, C counterpart. The coarse-grained version is the original, unmodified benchmark, run with one thread per processor. The fine-grained version was rewritten to dynamically create and destroy large numbers of lightweight Pthreads. This version was executed using both the Pthread implementation's original FIFO scheduler, and the new *AsyncDF* scheduling algorithm. See Chapter 5 for further details, along with other experimental results.

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	189	5.8	4.9	4.6	4.83	4.85	5.39

Figure 1.6: Summary of results for Pthreads benchmarks when the thread granularities are further reduced (*i.e.*, when the threads are made more fine grained) compared to the fine-grained version in Figure 1.5. The thread granularities are adjusted to be the finest granularities for which the basic thread overheads (such as creation and synchronization) are within 5% of the running time. For each scheduling technique, we show the maximum number of simultaneously active threads (each of which requires a min. 8KB stack) created by the scheduler, the L2 cache misses rates (%), and the speedups on an 8-processor Enterprise 5000 SMP. "FIFO" is the original Pthreads scheduler that uses a FIFO queue, "ADF" algorithm *AsyncDF*, and "DFD" is algorithm *DFDeques*. Further details can be found in Chapter 7.

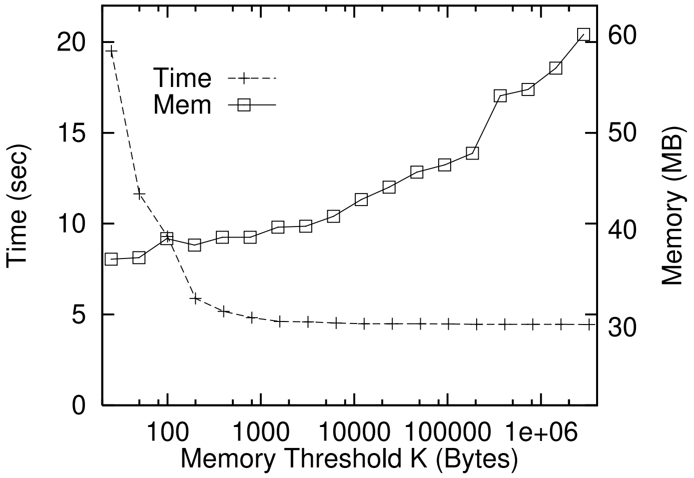


Figure 1.7: The variation of running time and memory usage with the memory threshold K (in bytes) in *AsyncDF*, for multiplying two 1024×1024 matrices using blocked recursive matrix multiplication on 8 processors of a Power Challenge. K=500–2000 bytes results in both good performance and low memory usage.

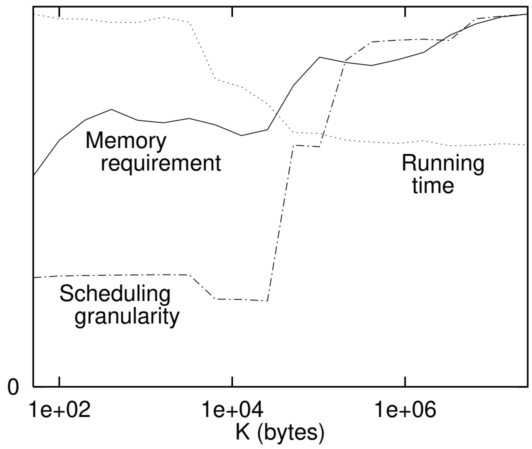


Figure 1.8: The variation of running time, scheduling granularity and memory usage with the memory threshold K (in bytes) for matrix multiplication using *DFDeques* on 8 processors of an Enterprise 5000. The units for the three quantities are shown in separate graphs in Chapter 5, along with similar results for synthetic benchmarks.

1.2 Limitations of the Dissertation

The limiting aspects of this dissertation are listed below.

(a) Machine model. The scheduling algorithms presented in this dissertation are designed for shared memory machines. I demonstrate their effectiveness in practice only on single SMPs. Unlike distributed memory machines, such machines have per-processor caches of limited size, and share common main memory modules. Provided a thread runs on a processor long enough to make good use of its cache, which processor it gets scheduled on has limited impact on performance. In contrast, in distributed machines, each processor has its own sizable memory module, and the local memory bus typically has higher bandwidth that the processor interconnects. On such machines, scheduling a thread that accesses data resident in a processor's memory module on that processor becomes more important. In Chapter 8, I speculate on how my scheduling algorithms may be extended to clusters of SMPs.

(b) Programming model. The programming model assumed by the scheduling algorithms is pure ***nested parallelism***; we define it formally in Chapter 2. Parallel languages that provide nested parallelism include data-parallel languages such as NESL [17] and HPF [60], as well as control-parallel languages such as Cilk [25] and Proteus [109]. Elsewhere, we have extended our space-efficient scheduler to the more general model of synchronization variables [19]; a description of those results is beyond the scope of this dissertation. In practice, the schedulers presented in this dissertation can be easily extended to execute programs with arbitrary, blocking synchronizations (see Chapters 5 and 7). Chapter 8 briefly speculates on how space bounds for more general programming models could also be analyzed in theory.

(c) Analysis. I analyze the running time and space requirement for parallel programs assuming a constant-time fetch-and-add instruction; the analysis does not reflect the effects of processors contending on synchronizations or memory accesses. Also, I do not analyze the total communication (in the form of cache misses) for the programs.

(d) Scalability of implementations. My implementations and experiments were carried out on single bus-based SMPs, with up to 16 processors. Each of my implementations therefore used a serialized scheduler, which does not appear to be a major bottleneck for up to 16 processors. I describe and analyze parallelized versions of the schedulers, which I expect would be more effective on a machine with a larger number of processors. However, mainly due to the lack of convenient access to such a machine, I do not experimentally validate this claim.

(e) Benchmarks. Since the algorithms are targeted towards nested parallel programs, the benchmarks I use to evaluate them predominantly use nested parallelism. The multithreaded runtime system on the Power Challenge handles multi-way forks that are common in data parallelism. However, the Pthreads interface allows only a binary fork, and all the Pthreads benchmarks are written using nested fork-join constructs. Some of the benchmarks also make limited use of Pthread mutexes. However, the Pthreads library itself makes extensive use of the mutexes and condition variables provided by the Pthreads API. The set of benchmarks includes compute-intensive and memory-intensive codes, but no I/O-intensive applications.

1.3 Organization of the Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 begins by presenting an overview of previous work on representing computations with graphs. I then motivate and explain the dynamic dags model used throughout this dissertation. I also formally define important properties of parallel programs and their parallel executions, such as space requirement, in terms of the dags. The chapter then differentiates between different models of parallelism, including nested parallelism, and ends with an overview of related work on dynamic scheduling techniques.

Chapter 3 describes algorithm *AsyncDF*, the first of the two asynchronous scheduling algorithms presented in this dissertation. I analyze the space and time bounds for a program executed using algorithm *AsyncDF*. I then describe how the scheduler itself can be parallelized, and analyze the space and time bounds including the overheads of the parallelized scheduler.

Chapter 4 describes the implementation of a multithreading runtime system that uses the *AsyncDF* scheduling algorithm. It also evaluates and compares the performance of the scheduler with previous scheduling techniques using a set of parallel benchmarks.

Chapter 5 explains how a slight variation of the *AsyncDF* scheduling algorithm can be added to a lightweight Pthreads implementation, making it space efficient. I show that the new scheduler can result in good space and time performance for fine-grained versions of a variety of Pthreads-based parallel benchmarks.

Chapter 6 presents the second space-efficient scheduling algorithm, namely, algorithm *DFDeques*. It also analyses the space and time requirements for executing a nested-parallel benchmark using the *DFDeques*.

Chapter 7 describes the implementation of *DFDeques* in the context of a Pthreads library. I experimentally compare the performance of *DFDeques* with the original Pthreads scheduler and with algorithm *AsyncDF*, by using the Pthreads-based benchmarks introduced in Chapter 5. I also present simulation results for synthetic benchmarks to compare the performance of *DFDeques* with both algorithm *AsyncDF*, and with another space-efficient (work-stealing) scheduler.

Chapter 8 summarizes the results presented in this dissertation, and describes future directions for research that emerge from this work.

Chapter 2

Background and Related Work

Dynamically generated graphs are used throughout this dissertation to represent computations with dynamic parallelism. This chapter begins with a description of previous work on using graphs to model parallel computations (Section 2.1.1), followed by a definition of the directed acyclic graph (dag) model used in this dissertation (Section 2.1.2). Sections 2.1.3 and 2.1.4 then define some properties of the computations (or their dags) that will be referred to in the remainder of this dissertation. Next, Section 2.2 describes some of the models of parallelism that can be implemented by fine-grained, lightweight threads. For each model, I give examples of existing thread systems or parallel languages that implement the model. The chapter ends with a summary in Section 2.3 of previous work on dynamic scheduling of lightweight threads for either good locality or space efficiency, along with an explanation of where this dissertation research fits in.

2.1 Modeling a Parallel Computation as a Graph

This section begins by listing some previous work on modeling parallel computations with different types of graphs, followed by a description of the particular graph model used in this dissertation. The section ends with definitions of important properties of parallel computations or their graph representations.

2.1.1 Background

Marimont [104], Prosser [130], Ianov [86], and Karp [93] introduced directed graphs to represent sequential programs. Since then, several researchers have used graphs to model both sequential and parallel computations for a variety of purposes.

Graphs have been used to represent programs in the functional programming community, so that the programs can be executed using graph reduction [157]. Internal nodes are ***application*** nodes while the leaves are either primitive operators or data values. The left child of an application node reduces to an operator, while its right child represents the operand for the left child. Graph reduction has been applied widely in the context of both sequential [29, 57, 94] and parallel [14, 125] machines.

Representing computations as graphs allows relationships between their space and time requirements to be studied using pebbling games. Graphs used in pebbling are typically directed, acyclic graphs in which nodes represent operations, and an edge from one node to another indicates that the result of one operation is an operand of another. Pippenger [127] summarizes early results in pebbling for sequential programs. Pebbling has subsequently been applied to generate parallel schedules for FFT graphs [141], and to characterize parallel complexity classes [160].

Dennis [51] introduced static dataflow graphs, which were subsequently augmented with dynamic tags that increase the available parallelism [4]. A dataflow graph is a directed graph in which nodes represent operators (or instructions) and the edges represent data dependences between nodes. Dataflow graphs have been used in the simulation of computer systems [68, 149]. As intermediate representations for dataflow languages, they are also used to execute dataflow programs at a fine-grained level, where individual operations are dynamically scheduled on special processors [6, 74]. Alternatively, the macro-dataflow approach involves compile-time partitioning of the graph nodes into coarser tasks and scheduling them onto more conventional multiprocessors [139, 140].

Computation graphs were introduced by Karp and Miller [91] as a model to represent parallelism in simple, repetitive processes. Each node of a computation graph is an operation and each edge represents a first-in first-out queue of data directed from one node to another. The model was subsequently extended to include features like conditional branching [2, 7, 52, 103, 106, 135]. These graphical models have a number of different types of nodes, including arithmetic operators and control flow decisions, and have been used to represent, analyze, manipulate and execute parallel computations.

A significant amount of work has focused on static scheduling of task graphs on parallel computers. A task graph is a directed graph in which each node represents a variable-length task (subcomputation). An edge connecting two tasks represents dependences between them, with weights denoting communication costs. The communication cost is incurred only if the two tasks are scheduled on separate processors. Although the problem of finding an optimal schedule was shown to be NP-complete even without communication costs [100], scheduling algorithms that produce schedules within a factor of 2 of optimal have been presented for various cases [28, 71, 121]. Several researchers have analyzed the trade-off between time and communication for different types of task graphs [89, 120, 122, 154]. A number of heuristics have been suggested to statically partition and schedule task graphs to minimize execution time [55, 69, 85, 107]. Task graphs have also been used to predict the performance of parallel computations [12, 98, 110, 144, 152].

Dynamic dags

All the above approaches use graphs as explicit, static representations of programs. In general, to statically construct and schedule such graphs, the costs for each runtime instance of a task (node) or data communication (edge) must be known at compile time. These quantities are difficult to statically estimate for programs with dynamic parallelism, for example, programs with data-dependent spawning and parallel recursion, or parallel loops with statically unknown bounds and iteration execution times, etc. Therefore, instead of using a static graph representation, programs with dynamic parallelism are modeled using dynamic graphs in this dissertation. A dynamic graph is a directed graph that corresponds to a single execution of a program, and unfolds (is revealed incrementally) as the execution proceeds. Each node corresponds to a separate execution instance of some task. Thus, a sequential loop would be represented as a linear sequence of nodes, created as the loop executes, instead of a cycle as in static task graphs. The dynamic graphs considered in this dissertation are assumed to be acyclic[1]. The complete, directed acyclic graph or ***dag*** is available only at the end of an execution, and need not be explicitly created by the scheduler. We simply view it as a trace of that execution.

Although static dags were initially used to represent arithmetic expressions [27, 28], researchers have more recently used dynamic dags to model parallel computations with dynamic parallelism. Burton *et al.* [30, 143], Blumofe and Leiserson [24, 26], and Blelloch *et al.* [21] modeled parallel computations as dynamic dags to generate efficient schedules. Leighton *et al.* [99] studied efficient, online embeddings in hypercubes, of dags that represent a tree of dynamically spawned tasks. Dynamic dag representations have also been used to define relaxed models of memory consistency [22], or for proving correctness of algorithms that detect race conditions [40, 58]. Bertsekas and Tsitsiklis [16] present a good introduction to the dynamic dag model for parallel programs. They explain how parallel executions can be represented by schedules for the corresponding dags, and they define the running time for a parallel execution in terms of these schedules.

2.1.2 Dynamic dags used in this dissertation

Dynamic dags are used as an abstract model to represent parallel computations in this dissertation. This has a number of advantages.

- A dag is a simple and intuitive representation for a program execution, independent of the parallel language or underlying parallel machine.
- The problem of dynamically generating a schedule for the program reduces to the problem of generating an online schedule for the dynamic dag (see Section 2.1.4).
- The properties of a parallel computation, such as the total number of operations or the critical path length, can be defined in terms of properties of the dag (see Sections 2.1.3). Similarly, the costs of a particular schedule can be analyzed in terms of properties of the dag.
- Different models of dynamic parallelism can be compared in terms of the structural properties of the dags generated by the parallel computations (see Section 2.2).

In the dag model used in this dissertation, each node represents a unit of computation or an ***action*** in a thread; therefore, I will use the terms "node" and "action" interchangeably. Each action must be executed serially, takes exactly a single timestep (clock cycle) to be executed on a processor. A single action may allocate or deallocate space. Since machine instructions do not necessarily complete in a single timestep, one machine instruction may translate to a series of multiple actions.

Each edge in the dag represents a dependence between two actions. Thus, if the dag has an edge (u, v), then the node v must be executed after node u. One or more new threads are created by executing a ***fork*** instruction. When a thread t_a forks a thread t_b, thread t_a is called the ***parent*** of thread t_b, and thread t_b a ***child*** of thread t_a. Figure 2.1 shows an example dag for a simple parallel computation. The ***fork*** edges, shown as dashed lines in the figure, are revealed when a

[1]A graph representing the execution of a computation with arbitrary synchronizations may contain a cycle, implying that the execution represented by the graph deadlocks. Therefore, we do not consider such computations.

thread forks a child thread. The fork edge goes from the current node of the forking thread to the initial node of the child thread. The dotted ***synch*** edges represent a synchronization between two threads, while each solid and vertical ***continue*** edge represents a sequential dependence between a pair of consecutive actions within a single thread. For every edge (u, v) in the dag, node u is called a ***parent*** of node v, and node v a child of node u.

Each node v in the dag is assigned an integer weight $m(v)$ to represent a memory allocation or deallocation. Thus, a node (action) v that allocates M units of memory is assigned a label $m(v) = M$. If the node v deallocates M bytes, then $m(v) = -M$; if it performs no allocation or deallocation, $m(v) = 0$. We assume that a single node (action) does not perform both an allocation and a deallocation.

If the programming model supports multi-way forks, a node can have a high out-degree. In contrast, in systems that only allow binary forks, the outdegree of a node can be at most two. As a convention, when a thread forks a child thread, I will draw the child thread to the immediate left of the parent thread. If multi-way forks are permitted, then the children are shown arranged from left-to-right in the forking (or program text) order.

Dag-determinism

Each dag represents a single execution of a parallel computation. Therefore, a single parallel computation may result in different dags for each parallel execution of the computation. The analysis in this dissertation focuses on parallel computations that are ***dag-deterministic***, that is, the structure of the dag (including the weights on its nodes) is the same for every execution of the computation. Thus, the dag representing a dag-deterministic computation is independent of factors such as the scheduling technique, the number of processors, or the relative order of execution of actions that can be executed in parallel. A dag-deterministic parallel computation may be non-deterministic in the traditional sense, that is, it may produce different results during different executions. However, to analyze the space and time costs, we simply require that the dag for each execution be the same. As an example of a computation that is not dag-deterministic, consider a program with speculative parallelism. It may perform more (or less) work when executed on multiple processors compared to its serial execution. Since the dags for the two executions differ, the computation is not dag-deterministic. Similarly, a search program with pruning may result in different dags for each execution, and is therefore not dag-deterministic.

The reason for focusing on dag-determinism in this dissertation is that the space and time costs for a parallel schedule are defined in terms of properties of the dag. For example, I define the parallel space requirement S_p of a computation as a property of the dag that represents the parallel execution of that computation on p processors. S_p is then bound in terms of the space requirement S_1 for a serial schedule of the *same* dag. Now this bound makes sense only if the serial execution of the computation actually produces the same dag. Therefore, this dissertation restricts its focus to dag-deterministic computations to allow meaningful analyses of the scheduling algorithms. The schedulers presented in this dissertation are, nonetheless, perfectly useful for executing computations that are not purely dag-deterministic. However, for such computations, the space and time costs would be bound in terms of the worst-case space and time costs for all possible dags, which may be difficult to predict or compute offline. This issue is discussed briefly in Chapter 8.

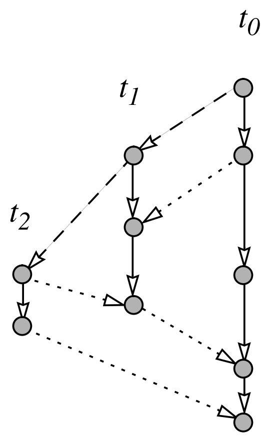


Figure 2.1: An example dag for a parallel computation. Each dashed edge represents a fork, and each dotted edge represents a synchronization of a child thread with its parent. Solid, vertical edges represent sequential dependences within threads. Here t_0, the initial (root) thread, forks a child thread t_1, which forks a child thread t_2.

2.1.3 Work and depth

The ***work*** of a dag-deterministic parallel computation is the total number of actions executed in it. Since each action requires a single timestep to be executed, the work of a parallel computation also equals the time taken to execute it serially (assuming zero scheduling overheads). A computation's ***depth*** is the length of the critical path in it, that is, the time required to execute the computation on an infinite number of processors (assuming no scheduling or communication overheads). Thus, the work of a dag-deterministic parallel computation equals the total number of nodes in its dag, while its depth equals the length of the longest path in the dag. For example, the dag in Figure 2.1 has 10 units (timesteps) of work, and a depth of 6 timesteps; thread t_0 performs 5 units of work.

2.1.4 Schedules: space and time requirement

For analyzing the space and time requirements of a parallel computation, this dissertation assumes that the clocks (timesteps) of the processors are synchronized. Therefore, although we are modeling asynchronous parallel computations, the schedules are represented as sets of nodes executed in discrete timesteps. With this assumption, the parallel (or serial) execution of a computation on p processors can be represented by a ***p-schedule*** $s_p = V_1, V_2, \ldots, V_\tau$, where V_i is the set of nodes executed at timestep i. Since each processor can execute at most one node in any timestep, each set V_i contains at most p nodes. Thus, for a serial schedule or 1-schedule, each V_i consists of at most a single node. A p-schedule must obey the dependency edges, that is, a node may appear in a set V_i only if all its parent nodes appear in previous sets.

The ***length*** of a p-schedule is the time required to execute the p-schedule, that is, the number of timesteps in it. Thus, a p-schedule $s_p = V_1, V_2, \ldots, V_\tau$ has length τ. The space required by s_p is $S_p = n + max_{j=1,\ldots,\tau}\left(\sum_{i=1}^{j}\sum_{v\in V_i} m(v)\right)$, where n is the size of the input data, and $m(v)$ is the amount of memory allocated by node v.

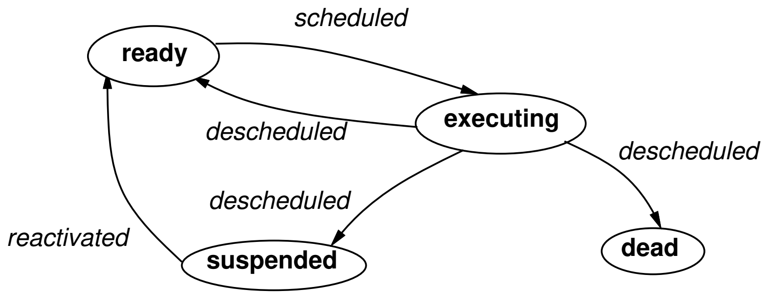


Figure 2.2: The state transition diagram for threads. An active thread that is either executing, ready or suspended.

A node in a thread is ***ready*** to execute when all its parents have been executed, but the node itself has not yet been executed. A thread that has been created but has not yet terminated is called ***active***. A thread is ***dead*** once it has terminated. At any time during the execution, the first unexecuted node (or action) in an active thread is called the thread's ***current*** node. An active thread is ***executing*** if it is currently being executed on some processor. Otherwise, an active thread is either ready or suspended. The thread is ***ready*** if its current node is ready; otherwise, it is said to be ***suspended***. Figure 2.2 shows the state transition diagram for threads. A suspended thread is ***reawakened*** when its current node becomes ready. We say a thread is ***scheduled*** when its state changes from ready to executing. When it is subsequently ***descheduled***, it leaves the executing state.

In this dissertation, a thread's ***granularity*** refers to the average number of actions executed by the thread. Thus, threads that perform only a few operations are ***fine grained***, while threads that perform a large number of operations are ***coarse grained***. For example, a SPMD-style computation that creates one thread per processor, independent of the problem size, has coarse-grained threads. I also (informally) define ***scheduling granularity*** to be the average number of actions executed consecutively on a single processor, from threads close together in the dag. Figure 2.3 shows two different mappings of threads from a parallel computation to processors that result in different scheduling granularities.

Depth-first schedules and serial space

As demonstrated by the examples in Chapter 1, the schedule generated for a parallel computation depends on the algorithm used to schedule the threads. Thus, several different serial schedules (with different space requirements) may exist for a single dag. A serial implementation that maintains a LIFO (last-in-first-out) stack of ready threads results in a depth-first schedule. A ***depth-first schedule*** is a 1-schedule in which, at every step, we pick the most recently executed node v that has a ready child node, and execute any child of v. Since the node v may have two or more children, and we do not specify which child node to pick first, a dag may have several depth-first schedules. For example, Figure 2.4 shows two possible depth-first schedules for the dag from Figure 2.1.

A specific depth-first schedule called a 1DF-***schedule*** can be defined for computations by introducing priorities on the threads—a total order where $t_a \succ t_b$ means that t_a has a higher priority

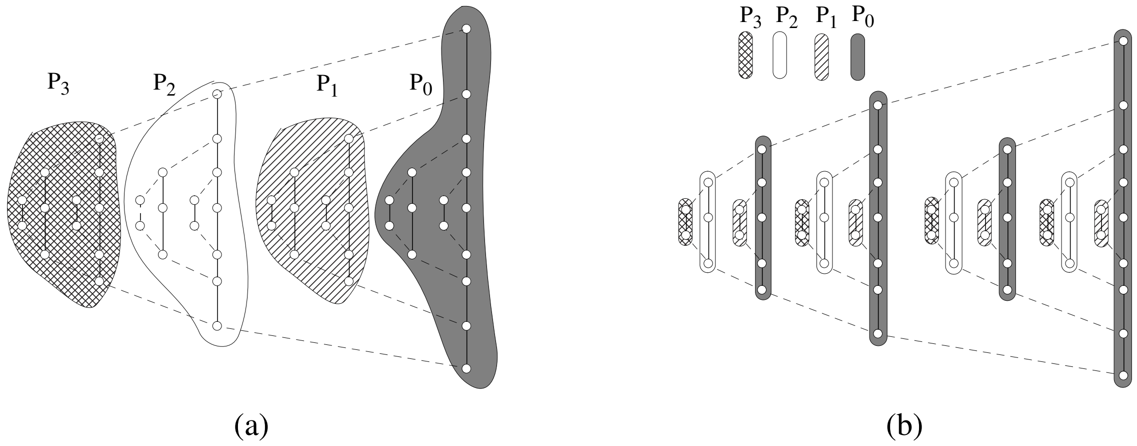


Figure 2.3: Two different mappings (a) and (b) of threads in a dag onto processors $P_0, \ldots, P_3$. (a) has higher scheduling granularity than (b).

than t_b. To derive this ordering we say that whenever a thread t_0 forks a thread t_1, the forked thread will have a higher priority than the forking thread ($t_1 \succ t_0$) but the same priority relative to all other threads currently existing in the system. If multi-way forks are permitted, priorities are assigned to the threads in the program text order; thus, if thread t_0 forks n threads $t_1, \ldots, t_n$ in a single timestep, then $t_1 \succ t_2 \succ \ldots \succ t_n \succ t_0$. A 1DF-schedule is the unique 1-schedule generated by always executing the current action of the highest-priority ready thread. The order in which nodes are executed in a 1DF-schedule determines their ***1DF-numbers***. For example, Figure 2.4 (b) shows the 1DF-numbering of the dag from Figure 2.1. Recall that my convention is to draw a newly forked child thread to the immediate left of its parent. Therefore, all my example dags are drawn such that, if $t_a \succ t_b$, then t_a appears to the left of t_b. Thus, the 1DF-schedule is the unique, *left-to-right* depth-first schedule.

Serial implementations of many languages execute a 1DF-schedule. In fact, if the thread creations (forks) were to be replaced by simple function calls, then the serial schedule executed by any stack-based language such as C is identical to a 1DF-schedule for most parallel computations. There exist some parallel computations (such as those with right-to-left synch edges), for which a 1DF-schedule need not be the most natural serial execution order. However, for the class of computations that this dissertation focuses on, the 1DF-schedule is indeed the most natural serial execution order, and therefore, its space requirement will be referred to as the serial space requirement S_1 for a parallel computation.

2.2 Models of Parallelism

Researchers in the past have compared different models for parallel machines; for example, Traff [155] presents a good overview of the PRAM model and its variants. There has also been some work in classifying parallel languages or models of parallel programming. Bal and Tanenbaum [8] compare the data-sharing semantics of parallel languages for distributed memory machines. Sipelstein and Blelloch [147] review and compare a set of parallel languages that provide

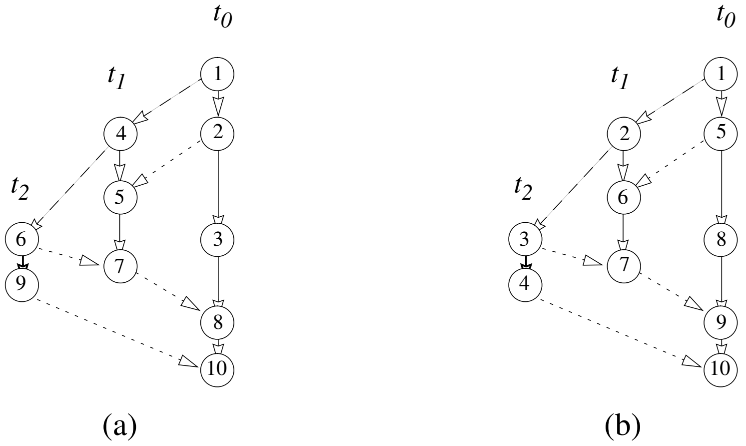


Figure 2.4: Two different depth-first schedules (a) and (b) for the example dag from Figure 2.1; each node is labelled with its execution order. A node cannot be executed until all the nodes that have edges into it have been executed. (b) is a 1DF-schedule, and therefore each node's label is its 1DF-number. To obtain a 1DF-schedule, priorities are assigned to the threads such that $t_2 \succ t_1 \succ t_0$.

data aggregates and operations to manipulate them as language primitives. Dekeyser and Marquet [50] classify some data-parallel languages according to the level of abstraction they provide to the programmer. Skillicorn and Talia classify a larger set of parallel languages into six categories based on similar criteria. At one end of their spectrum are languages with implicit parallelism, such as Haskell [82] and UNITY [38], for which the programming is relatively simple but the performance relies heavily on the language implementation. At the other extreme are languages such as MPI [61] in which the parallelism, partitioning, scheduling, communication and synchronization are all specified by the programmer. This dissertation research is applicable to the intermediate class of languages in which the parallelism is explicit and dynamic (in the form of lightweight threads), while the partitioning, scheduling, communication and synchronization are all implicit, and left to the language (or machine) implementation. I am not aware of any previous work that further differentiates between parallel languages in this specific class, based on the model of parallel programming supported by each language. Therefore, this section is an attempt to make such a differentiation. In particular, I describe the different models of parallelism based on the structure of the dags resulting from the parallel computations, and provide examples of parallel languages or computations that support each model.

The model of parallelism supported by a language (or thread library) determines the style in which lightweight threads may be created or synchronized in the language. Imposing no restrictions on the programming model leads to computations that are represented by arbitrary, unstructured dags, and executing them efficiently can be difficult. Therefore, a number of parallel languages or thread libraries provide a set of primitives that lead to computations with well-structured dags. Here I describe five models in roughly increasing order of flexibility or expressiveness. I only cover models that allow dynamic creation of threads, as opposed to the SPMD style where one thread is created per processor at the start of the computation (*e.g.,* in MPI or Split-C [49]). This list is by no means comprehensive, but covers the models implemented by a large set of parallel languages.

1. Flat parallelism

In the flat model of parallelism, the original (root) thread is allowed to fork child threads. All the child threads must synchronize with the root thread at a single point in the program and terminate; the child threads cannot fork more threads themselves or communicate with each other. Once the threads synchronize, the parent can once again fork new child threads. Figure 2.5 (a) shows a sample dag for a program with flat parallelism. Flat, data-parallel languages like High Performance Fortran [60] support this style of parallelism over elements of dense arrays. Parallelizing compilers also often generate an output program in this style. For example, when the compiler encounters a loop nest, it may parallelize one of the nested loops by creating one child thread per processor, and partitioning the iterations of that loop among the child threads. Alternatively, the compiler may create one child thread per iteration of the chosen loop, and use a static or dynamic scheme [84, 97, 128, 158] to partition the child threads among the processors. The child threads then synchronize with the parent at the end of the parallelized loop.

2. Nested parallelism

As with flat parallelism, all the child threads in a nested parallel computation synchronize with the parent at a single point and then terminate. No other synchronizations are permitted, and therefore, as with flat parallelism, child threads may not communicate with each other. However, a child thread is allowed to fork its own child threads, and the forks must be perfectly nested (see Figure 2.5 (b)). The dags that represent nested parallel computations are fairly well structured, and are called series-parallel dags [21]. Data-parallel languages such as NESL [17] and control-parallel languages such as Proteus [109] implement nested parallelism. A number of parallel programs, including divide-and-conquer computations, branch-and-bound searches, functional expression evaluations, parallel loop nests, and octree-based codes for graphics or dynamics applications, make use of nested parallelism. The examples in Chapter 1 are nested parallel programs.

3. Strictness

Strict computations [24] extend nested parallel computations by allowing one or more synchronization edges from a thread to its ancestor threads. In functional programs, this restriction implies that a function cannot be invoked until all its arguments are available, although the arguments may be computed in parallel. ***Fully strict*** computations are a subset of strict computations that only allow synchronization edges from a thread to its parent. For example, the dag in Figure 2.5 (c) represents a fully strict computation. The Cilk programming language [25] implements a subset of fully strict computations in which, all the children currently forked by a thread must synchronize with the parent at a single point; this subset is equivalent to nested parallel computations in its expressiveness.

4. Synchronization variables

A synchronization variable is a write-once variable, and any thread that performs a read on a synchronization variable before it has been written to, must suspend. Once a write is performed on the variable, it can be read multiple times. Therefore, there is a synchronization edge from the node

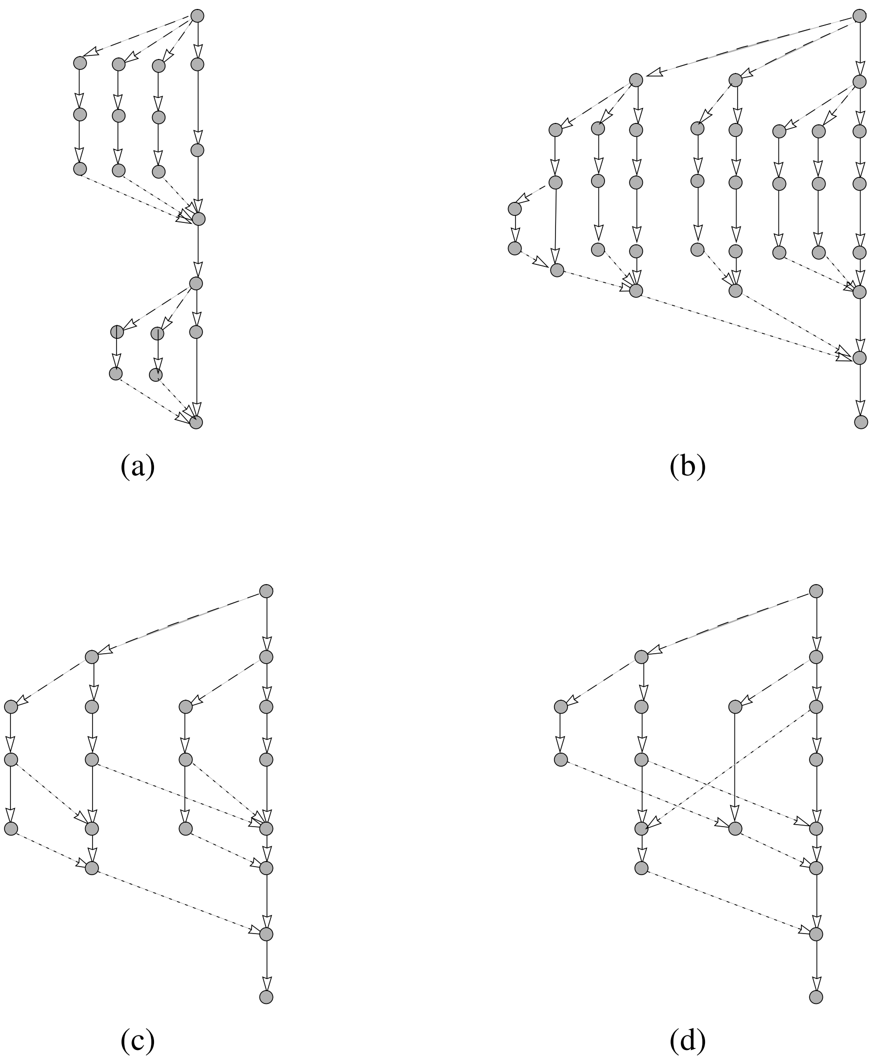


Figure 2.5 : Example dags for the different models of dynamic parallelism. (a) A computation with flat parallelism, where the root thread may fork child threads. (b) A nested parallel computation, in which all forks and joins are nested. (c) A strict computation, which allows a thread to synchronize with its ancestors (this example is also fully strict because threads only synchronize with their parents). (d) A computation with synchronization variables, where synchronization edges go from nodes that perform writes to synchronization variables, to nodes that read them.

that performs a write on a synchronization variable, to every node that reads it. There is no restriction on the pattern of synchronization edges, that is, a synchronization edge may go from a thread to any other thread, as long as the resulting graph is a dag (has no cycles). As with the above models, new threads are created by forking. Figure 2.5 (d) shows an example dag representing a parallel computation with synchronization variables. Synchronization variables can be used to implement futures in such languages as Multilisp [77], Mul-T [95], Cool [36] and OLDEN [34]; I-structures in Id [5]; events in PCF [148]; streams in SISAL [59]; and are likely to be helpful in implementing the user-specified synchronization constraints in Jade [134]. Computations with futures are a special case, in which all the synch edges in the dag go from left to right; the 1DF-schedule is a natural serial execution order for such computations.

4. Other synchronization primitives

Computations that use some of the other synchronization primitives, such as mutexes, semaphores, or condition variables, are often not dag-deterministic. For example, consider a program in which the root thread forks a child thread, and both then execute some critical section protected by a mutex. If the root thread acquires the mutex first, there is a synchronization edge from the node in the root thread that unlocks the mutex to the node in the child thread that locks it. In contrast, if the child acquires the mutex first, the synchronization edge goes from the unlocking node in the child thread to the locking node in the parent. Thus, although individual executions have unique dags to represent them, the computation is not dag-deterministic, *i.e.*, it cannot be represented with a single dag that is independent of the individual executions. A number of general-purpose thread libraries provide such arbitrary synchronization primitives [15, 35, 44, 46, 88, 116, 150].

The scheduling algorithms presented in this dissertation are analyzed for nested parallel computations. However, the Pthreads-based implementations described in Chapters 5 and 7 use the schedulers to execute computations with arbitrary synchronization primitives, including mutexes and condition variables. The experimental results indicate that the schedulers work well in practice for computations with a moderate amount of such synchronizations.

2.3 Previous work on lightweight thread systems

A variety of systems have been developed to schedule lightweight, dynamic threads [13, 25, 34, 36, 47, 75, 76, 90, 102, 105, 111, 115, 126, 137, 159, 161, 162, 164]. Although the main goals have been to achieve low runtime overheads, good load balancing, and high locality, a significant body of work has also focused on developing scheduling techniques to conserve memory requirements. This section presents an brief overview of previous work on dynamic thread scheduling for good locality, followed by a description of space-efficient schedulers.

2.3.1 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 do not apply to

SMPs. However, in many parallel programs, 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, increasing scheduling granularity typically provides good locality, and also reduces scheduling contention (that is, contention between processors while accessing shared scheduling data structures). For example, if the i^{th} thread (going from left to right) in Figure 2.3 accesses the i^{th} block or element of an array, then scheduling consecutive threads on the same processor, as in Figure 2.3 (a), provides better locality. Hence many parallel implementations of fine-grained threads use per-processor data structures to store ready threads [66, 77, 90, 92, 101, 145, 159]. Threads created on a processor are stored locally and moved only when required, so that most of the threads created on one processor are executed on that processor. This technique effectively increases scheduling granularity, and can be applied to a variety of programs that dynamically create threads, including those with nested parallelism and synchronization variables. For programs with parallel loops, techniques that statically or dynamically chunk loop iterations were proposed to achieve similar goals [84, 97, 128, 158].

An alternate approach to scheduling for locality has focused on taking advantage of user-supplied annotations or hints regarding the data access patterns of the threads. For example, if threads are annotated with the memory locations that they access, then threads accessing the same or nearby locations can be scheduled in close succession on the same processor [36, 105, 126]. Similarly, if hints regarding the sharing of data between threads can be provided by the programmer, an estimate of the data footprint sizes of threads in a processor's cache can be computed and used by the scheduler [161]. However, such annotations can be cumbersome for the user to provide in complex programs, and are often specific to a certain language or library interface. Therefore, instead of relying on user-supplied hints, the algorithm presented in this dissertation (Chapter 7) uses the heuristic of scheduling threads close in the dag on the same processor to obtain good locality.

2.3.2 Scheduling for space-efficiency

The initial approaches to conserving memory were based on heuristics that work well for some applications, but do not provide guaranteed bounds on space [13, 32, 48, 56, 67, 70, 73, 77, 102, 111, 118, 119, 124, 138]. For example, Ruggiero and Sargeant introduced throttling [138] to avoid the creation of too many tasks in the Manchester dataflow machine [74]. Their technique involves switching between FIFO (first-in-first-out) and LIFO (last-in-first-out) scheduling on the basis of the system load. Peyton-Jones *et al.* [124] present a method to control parallelism in parallel graph reduction, based on spawning new parallel tasks only when the system load is low. Burton [32], Epstein [56], and Osborne [118] suggested techniques to control speculative parallelism by assigning priorities to tasks. Culler and Arvind [48] proposed loop-bounding techniques to control the excess parallelism in a dataflow language Id [5]. Multilisp [77], a flavor of Lisp that supports parallelism through the "future" construct (and therefore supports computations with write-once synchronization variables), uses per-processor stacks of ready threads to limit the parallelism. Each processor adds or removes threads from the top of its stack, while idle processors steal from the bottom of some other processor's stack. A similar method was subsequently implemented in the context of Qlisp [123]. Lazy allocation of a thread stack involves allocating the stack when the thread is first executed, instead of when it is created, thereby reducing thread creation time and conserving memory [13]. Lazy thread creation [70, 90, 111] avoids allocating resources for a thread unless it is executed in parallel. Filaments [102], a package that supports fine-grained fork-join or loop parallelism using stateless threads, conserves memory and reduces overheads by coarsening and pruning excess parallelism. Nested data parallel languages that are implemented using flattening [18] may require large amounts of memory; runtime serialization can reduce memory requirements for some programs [119]. Grunwald and Neves use compile-time analysis of the whole program to conserve the stack space required for lightweight threads [73]; they calculate the exact size of the stack required for each function, and insert checks for stack overflow in recursive functions. Fu and Yang reduce memory usage for volatile objects in a distributed objects model, by clustering and scheduling tasks based on knowing which objects they access [67].

Recent work has resulted in provably efficient scheduling techniques that guarantee upper bounds on the space required by the parallel computation [21, 24, 26, 30, 143]. For example, Blumofe and Leiserson [24] showed that randomized work stealing guarantees an upper bound of $p \cdot S_1$ for the space required on p processors, where S_1 is the serial (depth-first) space requirement[2]. In their algorithm, each processor maintains its own list of ready threads, which it treats like a FIFO stack; threads are pushed on or popped off the top of this ready stack by the processor. When a processor runs out of threads on its own stack, it picks another processor at random, and steals from the *bottom* of its stack. Their model applies to fully-strict computations with binary forks, and allows memory allocations on a thread stack, but not on the heap[3]. They also bound the time and communication requirements of the parallel computations. The Cilk runtime system [25] uses this randomized work stealing algorithm to efficiently execute multithreaded programs. Various other systems use similar work stealing strategies [76, 111, 115, 159] to control the parallelism. This approach also typically provides good locality, since threads close together in the dag are scheduled on the same processor.

The scheduling algorithm proposed by Burton and Simpson [30, 143] also guarantees an upper bound of $p \cdot S_1$. Their model allows dag-deterministic computations with arbitrary dags and is therefore more general than fully strict parallelism. However, their definition of S_1 is the maximum space required over all possible depth-first serial executions, rather than just the left-to-right depth-first execution. There may not exist an efficient algorithm to compute this value of S_1 for arbitrary dags.

A recent scheduling algorithm by Blelloch *et al.* [21] improved the previous space bounds from a multiplicative factor on the number of processors to an additive factor for nested parallel computations. The algorithm generates a schedule that uses only $S_1 + O(p \cdot D)$ space, where D is the depth of the parallel computation. This bound is asymptotically lower than the previous bound of $p \cdot S_1$ when $D = o(S_1)$, which is true for parallel computations that have a high degree of parallelism, such as all programs in the class NC [42]. For example, a simple algorithm to multiply two $n \times n$ matrices has depth $D = \Theta(\log n)$ and serial space $S_1 = \Theta(n^2)$, giving space bounds of $O(n^2 + p \log n)$ instead of $O(n^2 p)$ on previous systems. The low space bound of $S_1 +$

[2]More recent work provides a stronger upper bound than $p \cdot S_1$ for space requirements of regular divide-and-conquer algorithms using randomized work stealing [23].

[3]Their model does not allow any 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.

$O(p \cdot D)$ is achieved by ensuring that the parallel execution follows an order that is as close as possible to the serial execution. However, the algorithm has scheduling overheads that are too high for it to be practical. Since it is synchronous, threads need to be rescheduled after every unit computation to guarantee the space bounds. Moreover, it ignores the issue of locality—a thread may be moved from processor to processor at every timestep, and threads close together in the dag may get scheduled on different processors.

This dissertation presents two asynchronous scheduling algorithms *AsyncDF* and *DFDeques*, that maintain the same asymptotic space bound of $S_1 + O(p \cdot D)$ as the previous scheduling algorithm [21], but overcome the above problems. The main goal in the design of the two algorithms was to allow threads to execute nonpreemptively and asynchronously, allowing for better locality and lower scheduling overheads. The low space bound is achieved by prioritizing threads according to their execution order in the 1DF-schedule (*i.e.,* the 1DF-numbering of their current nodes), and preempting them when they run out of a preallocated quota of memory. In addition, threads that perform big memory allocations are delayed by lowering their priority. To ensure scalability, I present parallelized versions of the scheduler for both algorithms. I show that, including the costs of the parallelized scheduler, both algorithms execute parallel computations with W work in $O(W/p + D \cdot \log p)$ time and $S_1 + O(p \cdot \log p \cdot D)$ space.

The algorithms *AsyncDF* and *DFDeques* are designed and analyzed for nested parallel computations on shared memory machines. Our model allows memory to be allocated on the shared heap as well as the thread stacks. Algorithm *AsyncDF* allows multi-way forks, while algorithm *DFDeques* allows only binary forks. Algorithm *AsyncDF* uses a single, globally ordered priority queue of ready threads, with input and output buffers that provide concurrent, non-blocking access to the queue. However, this results in poor locality when threads are fine grained, since, although a single thread may execute on the same processor, threads close together in the dag may be scheduled on different processors. For example, the mapping in Figure 2.3 (a) could be a possible mapping of threads to processors using work stealing, while algorithm *AsyncDF* may result in the mapping shown in Figure 2.3 (b).

Algorithm *DFDeques* improves upon algorithm *AsyncDF* by allowing processors to use separate ready queues, which are globally ordered. This results in better locality (by scheduling fine-grained threads close in a dag on the same processor) and lower contention during scheduling, at the cost of a slight increase in space requirement. It is more efficient in practice compared to algorithm *AsyncDF* when the threads are finer grained. I present experimental results to show that both the algorithms achieve good performance in terms of both memory and time for a variety of parallel benchmarks. Although I analyze the algorithms for nested parallel computations, experimental results indicate that they are useful in practice for scheduling parallel computations with more general synchronization primitives, such as locks or condition variables (see Chapters 5 and 7).

Algorithm *AsyncDF* was recently extended to execute computations with synchronization variables in a provably-efficient manner [19]. This extension allows a computation with W work, σ synchronizations, D depth and S_1 sequential space, to run in $O(W/p+\sigma\cdot\log(p\cdot D)/p+D\cdot\log(p\cdot D))$ time and $S_1 + O(p \cdot D \cdot \log(p \cdot D))$ space on p processors. The bounds include all the space and time costs for the scheduler. Lower bounds for the special case where the computation graph is planar were also provided. A detailed description of the scheduling algorithm is beyond the scope of this dissertation.

Chapter 3

AsyncDF: An asynchronous, Space-Efficient Scheduler

This chapter describes and analyzes algorithm *AsyncDF*, the first of the two space-efficient scheduling algorithms presented in this dissertation. The chapter begins by listing the basic ideas behind the algorithm that make it space and time efficient (Section 3.1). Section 3.2 describes the data structures used by the scheduler, and then presents the pseudocode for algorithm *AsyncDF*. Section 3.3 then analyzes the space and time requirements of a parallel computation executed using algorithm *AsyncDF*. I show that a parallel computation with depth D, work W, and serial space requirement S_1, is executed on p processors by algorithm *AsyncDF* using $S_1 + O(K \cdot p \cdot D)$ space (including scheduler space); here, K is the value of the memory threshold used by *AsyncDF*. Further, if S_a is the total space allocated in the computation, I show that the execution requires $O(W/p + S_a/(K \cdot p) + D)$ timesteps. This time bound does not include the scheduling overheads; in Section 3.4 I describe the implementation of a parallelized scheduler and analyze the space and time bounds including scheduler overheads. The contents of this chapter are summarized in Section 3.5.

3.1 Basic Concepts

Algorithm *AsyncDF* is designed for the model of nested parallelism with multi-way forks. The algorithm is based on the following concepts; the first four ensure space efficiency, while the last two are aimed at obtaining good time performance.

- **Depth-first priorities**. As with the work of Blelloch *et al.* [21], threads are stored in a global queue prioritized according to their depth-first execution order, that is, according to the 1DF-numbering of their current nodes. This limits the number of nodes that execute "out-of-order" with respect to a 1DF-schedule, thereby bounding the amount of space allocated in excess of the serial space requirement S_1 of the 1DF-schedule.

- **Fixed memory quota**. Every time a thread is scheduled on a processor, that is, every time it moves from the ready state to the executing state, it is assigned a fixed memory quota of K bytes. The user-adjustable value K is called the ***memory threshold*** of the scheduler. When the thread allocates space on its stack or on the shared heap, the available memory quota is appropriately reduced. Once the thread reaches an instruction that requires more memory than the remaining quota, the thread is preempted, and the processor finds the next ready thread to execute.

- **Delays before large allocations**. When a thread reaches an action that allocates a large amount[1] of space, the thread is stalled by inserting dummy threads before the large allocation. In the meantime, if other threads with a higher priority become ready, they get scheduled before the stalled thread.
- **Lazy forks**. When a thread reaches a fork instruction, it is preempted and added to the scheduling queue. Its child threads are created lazily, that is, only when they are selected to be scheduled. Until then, the parent thread acts as a representative for all its child threads that are yet to be created.
- **Uninterrupted execution**. Threads are allowed to execute without interruption on the same processor until they fork, suspend, terminate, or run out of their memory quota. This improves upon previous work [21] by providing better locality and lower scheduling overheads.
- **Asynchronous scheduling**. The scheduling is overlapped with the computation, and the processors take turns performing the task of scheduling. The processors execute asynchronously with respect to each other, that is, they do not synchronize after every timestep[2].

Any single thread may be scheduled several times. Every time it is scheduled, a sequence of its actions is executed non-preemptively. We will refer to each such sequence as a ***batch***. In algorithm *AsyncDF*, all the nodes in a batch are part of a single thread. Thus, algorithm *AsyncDF* effectively splits a thread into one or more batches. The term batch will be used subsequently in this chapter to analyze the space requirement.

3.2 The *AsyncDF* Algorithm

This section presents the *AsyncDF* scheduling algorithm. It starts by listing the data structures required for scheduling, followed by pseudocode for the algorithm itself.

3.2.1 Scheduling data structures

As mentioned in the Section 3.1, algorithm *AsyncDF* prioritizes threads according to the 1DF-numbers of their current nodes. By scheduling threads according to these priorities, the algorithm ensures that their execution order in the parallel schedule is close to that in the 1DF-schedule.

To maintain the relative thread priorities, algorithm *AsyncDF* uses a shared priority queue $\mathcal{R}$ to store ready threads, suspended threads, and ***stubs*** that act as place-holders for threads that are currently being executed. Threads in $\mathcal{R}$ are stored from, say, left to right, in increasing order of

[1] Here, a "large" allocation is an allocation exceeding the memory quota K.

[2] The analysis, however, assumes that the clocks on the processors are synchronized.

the 1DF-numbers of their current nodes. The lower the 1DF-number of a thread's current node, the higher is the thread's priority.

Storing threads in order of their priorities requires $\mathcal{R}$ to support operations such as inserting or deleting from the middle of the queue. Implementing these operations in a concurrent and efficient manner on a shared queue is difficult. Therefore, two additional FIFO queues, Q_{in} and Q_{out}, are used instead to provide concurrent accesses. Q_{in} and Q_{out} act as input and output buffers, respectively, to store threads that are to be inserted into or that have been removed from $\mathcal{R}$ (see Figure 3.1). Thus, processors can perform fast, non-blocking accesses to the two FIFO buffer queues, instead of directly accessing $\mathcal{R}$. The scheduler described in this chapter serializes the transfer of threads between the buffer queues and $\mathcal{R}$; Section 3.4 describes how this transfer can be parallelized.

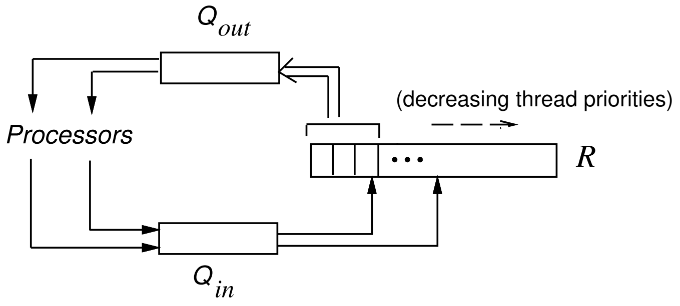


Figure 3.1 : The movement of threads between the processors and the scheduling queues. Q_{in} and Q_{out} are FIFOs, whereas $\mathcal{R}$ allows insertions and deletions of intermediate nodes. Threads in $\mathcal{R}$ are always stored from left to right in decreasing order of priorities.

3.2.2 Algorithm description

Figure 3.2 shows the pseudocode for the *AsyncDF*(K) scheduling algorithm; K is the user-adjustable memory threshold for the scheduler. The processors normally act as ***workers***. A worker processor takes a thread from Q_{out}, executes it until it terminates, preempts itself, or suspends (waiting for one or more child threads to terminate). The worker then returns the thread to Q_{in} and picks the next thread from Q_{out}. Every time a thread is picked from Q_{out}, it is allotted a memory quota of K bytes; it may subsequently use the quota to allocate heap or stack space. When the thread exhausts its memory quota and reaches an action that requires additional memory to be allocated, it must preempt itself before performing the allocation. A thread that reaches a fork instruction also preempts itself.

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

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

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

Figure 3.2: The *AsyncDF* scheduling algorithm. *currT* is the current thread executing on a processor. The thread preempts itself when it reaches a fork instruction, or when it needs to allocate memory after exhausting its memory quota. When the scheduler creates new child threads, it inserts them into $\mathcal{R}$ to the immediate left of their parent thread. This maintains the invariant that the threads in the ready queue are always in increasing order of the 1DF-numbers of their current nodes. Child threads are forked only when they are to be added to Q_{out}, that is, when they are among the leftmost p ready threads in $\mathcal{R}$.

A processor that executes a scheduling step starts by putting a new scheduling thread on Q_{out}. Next, it moves all the threads from Q_{in} to $\mathcal{R}$. Each thread has a pointer to a stub that marks its original position relative to the other threads in $\mathcal{R}$; it is inserted back in that position. The scheduler then compacts the ready queue by removing threads that have terminated. If a thread is the last among its siblings to terminate, and the scheduler finds its parent thread suspended, the parent thread is reactivated. All threads that were preempted due to a fork or an exhaustion of their memory quota, are returned to $\mathcal{R}$ in the ready state. If a thread has reached a fork instruction, its child threads are created and inserted to its immediate left by the scheduler. The child threads are placed in $\mathcal{R}$ in order of the 1DF-numbers of their current nodes. Finally, the scheduler moves the *leftmost* p ready threads from $\mathcal{R}$ to Q_{out}, leaving behind stubs to mark their positions in $\mathcal{R}$. If $\mathcal{R}$ has less than p ready threads, the scheduler moves them all to Q_{out}. The scheduling thread then completes the scheduling step, and the processor resumes the task of a worker.

To limit the number of threads in $\mathcal{R}$, the child threads of a forking thread are created *lazily*. A child thread is not explicitly created until it is to be moved to Q_{out}, that is, when it is among the leftmost p threads represented in $\mathcal{R}$. Until then, the parent thread implicitly represents the child thread. A single parent may represent several child threads. This optimization ensures that a thread does not have an entry in $\mathcal{R}$ until it has been scheduled at least once before, or is in (or about to be inserted into) Q_{out}. If a thread T is ready to fork child threads, all its child threads will be forked (created) and scheduled before either T, or any other threads in $\mathcal{R}$ to the right of T, can be scheduled.

Handling large allocations of space. In algorithm *AsyncDF(*K*)*, every time a thread is scheduled, its memory quota is reset to K bytes (the memory threshold). This does not allow any single action within a thread to allocate more than K bytes. I now explain how such allocations are handled, similar to the technique suggested in previous work [21]. The key idea is to delay the big allocations, so that if threads with lower 1DF-numbers become ready in the meantime, they will be executed instead. Consider a thread with a node that allocates m units of space in the original dag, where $m > K$. We transform the dag by inserting a fork of m/K parallel threads called ***dummy*** threads, before the memory allocation (see Figure 3.3). These dummy threads perform a single action (a no-op) each, but do not allocate any space. Every time a processor executes the no-op in a dummy thread, the thread terminates, and the processor gets a new thread from Q_{out}. Once all the dummy threads have been executed, the original thread can proceed with the allocation of m space. Because the dummy threads, which are added using an m/K-way fork, may execute in parallel, this transformation increases the depth of the dag by at most a constant factor. If S_a is the total space allocated in the program (not counting the deallocations), the number of nodes in the transformed dag is at most $W + S_a/K$. This transformation takes place at runtime, and the on-line *AsyncDF* algorithm generates a schedule for this transformed dag. This ensures that the space requirement of the generated schedule does not exceed our space bounds, as proved in Section 3.3.

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

Lemma 3.1 *The AsyncDF scheduling algorithm always maintains the threads in $\mathcal{R}$ from left to right in an increasing order of the* 1DF*-numbers of their current nodes.*

Proof: This lemma can be proved by induction. When the execution begins, $\mathcal{R}$ contains just the root thread, and therefore it is ordered by the 1DF-numbers. Assume that at the start of some

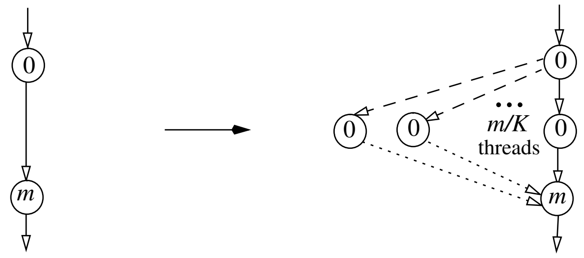


Figure 3.3: A transformation of the dag to handle a large allocation of space at a node without violating the space bound for algorithm *AsyncDF*(K). Each node is labeled with the amount of memory its action allocates. When a thread needs to allocate m space ($m > K$), m/K dummy child threads are forked in parallel before the allocation. Each dummy thread consists of a no-op action that does not allocate any space. After these dummy threads complete execution, the original thread may perform the allocation and continue with its execution.

scheduling step, the threads in $\mathcal{R}$ from left to right are in increasing order of the 1DF-numbers of their current nodes. For a thread that forks, inserting its child threads to its immediate left in the order of their 1DF-numbers maintains the ordering by 1DF-numbers. A thread that suspends due to a memory allocation is returned to its original position. Its new current node has the same 1DF-number as its previous node, *relative* to current nodes of other threads. Deleting threads from $\mathcal{R}$ does not affect their ordering. Therefore the ordering of threads in $\mathcal{R}$ by 1DF-numbers is preserved after every operation performed by the scheduler. ∎

Lemma 3.1 implies that when the scheduler moves the leftmost p ready threads from $\mathcal{R}$ to Q_{out}, their current nodes have the lowest 1DF-numbers among all the ready nodes in $\mathcal{R}$. We will use this fact to prove the space bound of the schedule generated by algorithm *AsyncDF*.

3.3 Analysis of Algorithm *AsyncDF*

In this section, I analyze the space and time required by a parallel computation executed using algorithm *AsyncDF*. The space bound includes the space required by the scheduling data structures in the general case. Recall that the scheduler in algorithm *AsyncDF* is serialized; therefore, the time bound in this section is presented only for the special case when the scheduler does not become a bottleneck. In general, this bottleneck can be avoided by parallelizing the scheduler. For a detailed analysis of the space and time bounds including the overheads of a parallelized scheduler in the general case, see Section 3.4.2.

This section first states the cost model assumed by the analysis. I then show that a parallel computation with depth D and work W, which requires S_1 space to execute on one processor, is executed by the *AsyncDF* scheduling algorithm on p processors using a memory threshold K in $S_1 + O(K \cdot p \cdot D)$ space (including scheduler space). Next, I show that the execution requires $O(W/p + S_a/(K \cdot p) + D)$ timesteps, where S_a is the total space allocated in the computation.

Recall that according to the dag model defined in Section 2.1.2, the depth D is defined in terms of actions, each of which requires a unit timestep (clock cycle) to be executed. Since the granularity of a clock cycle is somewhat arbitrary, especially considering highly pipelined processors with multiple functional units, this would seem to make the exact value of the depth D somewhat arbitrary. For asymptotic bounds this is not problematic since the granularity will only make constant factor differences. In Appendix A, however, I modify the space bound to be independent of the granularity of actions, making it possible to bound the space requirement within tighter constant factors.

3.3.1 Cost model

The following timing assumptions are made in the space and time analysis presented here. However, these assumptions are not required to ensure the correctness of the scheduler itself.

As explained in Section 2.1.2, the timesteps are synchronized across all the processors. At the start of each timestep, we assume that a worker processor is either busy executing a thread, or is accessing the queues Q_{out} or Q_{in}. An idle processor always busy waits for threads to appear in Q_{out}. We assume a constant time, atomic fetch-and-add operation in our system. Therefore, using the algorithms described in Appendix C, all worker processors can access Q_{in} and Q_{out} in constant time. Thus, at any timestep, if Q_{out} has n threads, and p_i processors are idle, then $\min(n, p_i)$ of the p_i idle processors are guaranteed to succeed in picking a thread from Q_{out} within a constant number of timesteps. We do not need to limit the duration of each scheduling step to prove the space bound; we simply assume that it takes at least one timestep to execute.

3.3.2 Space bound

As introduced in Section 3.1, each sequence of nodes in a thread that is executed non-preemptively is called a batch. We will call the first node to be executed in a batch a ***heavy*** node; it is the current node of the thread when it is scheduled. The remaining nodes in a batch are called ***light*** nodes. Thus, a batch is a heavy node followed by a sequence of light nodes. Note that the splitting of a thread into batches, and hence the classification of nodes as heavy or light, depends on the value of the memory threshold being used by the scheduler.

We attribute all the memory allocated by light nodes in a batch to the heavy node of the batch, and assume that light nodes do not allocate any space themselves (although they may deallocate space). This is a conservative view of space allocation, that is, the total space allocation analyzed in this section using this assumption is equal to or more than the actual space allocation. The basic idea in the analysis is to bound the number of heavy nodes that execute out of order with respect to the 1DF-schedule.

When a thread is inserted into any of the queues Q_{in}, Q_{out} or $\mathcal{R}$, we will say its current (and heavy) node has been inserted into the queue. A heavy node may get executed several timesteps after it becomes ready and after it is put into Q_{out}. However, a light node is executed in the timestep it becomes ready, since a processor executes consecutive light nodes in a batch nonpreemptively.

Let $s_p = V_1, \ldots, V_\tau$ be the parallel schedule of the dag generated by algorithm *AsyncDF*(K). Here V_i is the set of nodes that are executed at timestep i. Let s_1 be the 1DF-schedule for the same dag. At any timestep during the execution of s_p, all the nodes executed so far form a ***prefix*** of s_p. For $j = 1, \ldots, \tau$, a ***j-prefix*** of s_p is defined as the set of all nodes executed during the first j timesteps of s_p, that is, the set $\bigcup_{i=1}^{j} V_i$.

Consider an arbitrary j-prefix σ_p of s_p, for any $j = 1, \ldots, \tau$. 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 3.4 shows a simple dag with a parallel prefix σ_p for an arbitrary p-schedule s_p, and its corresponding serial prefix σ_1. Since all the premature nodes appear in s_1 after all the non-premature nodes, they have higher 1DF-numbers than the non-premature nodes.

The parallel execution has higher memory requirements because of the space allocated by the actions corresponding to the premature nodes. Hence we need to bound the space allocated by the premature nodes in σ_p. To get this bound, we need to consider only the heavy premature nodes, since the light nodes do not allocate any space. We will assume for now that no single node in the dag allocates more than K bytes. Later we will relax this assumption to cover bigger allocations. We first prove the following bound on the number of heavy premature nodes that get executed in any prefix of the parallel schedule.

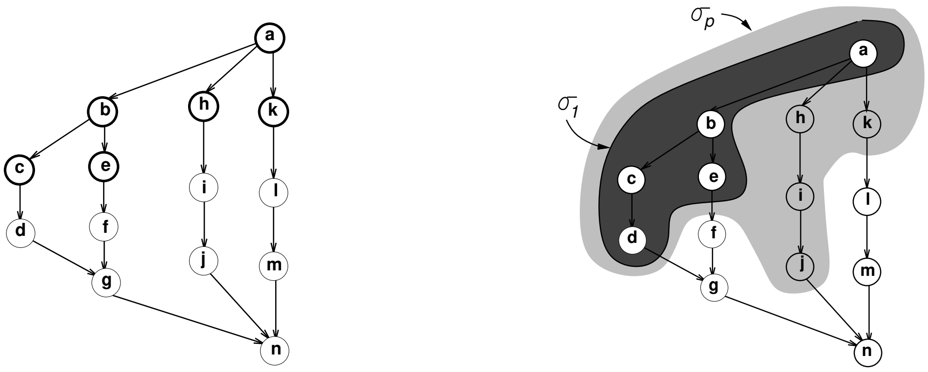
Lemma 3.2 *Let G be a dag of W nodes and depth D. Let s_1 be the 1DF-schedule for G, and let s_p be a parallel schedule for G executed by the AsyncDF algorithm on p processors. Then the number of heavy premature nodes in any prefix of s_p with respect to the corresponding prefix of s_1 is at most $O(p \cdot D)$.*

Proof: Consider an arbitrary prefix σ_p of s_p, and let σ_1 be the corresponding prefix of s_1. Let v be the last non-premature node to be executed in the prefix σ_p; if there are two or more such nodes, pick any one of them. Let P be a path in the dag from the root to v constructed such that, for every edge (u, u') along P, u is the last parent (or any one of the last parents) of u' to be executed. (For example, for the σ_p shown in Figure 3.4, v is the node labelled e, and the path P is (a, b, e).) Since v is non-premature, all the nodes in P are non-premature.

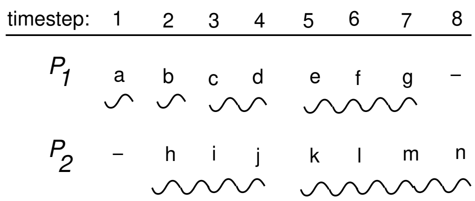
Let u_i be the node on the path P at depth i; then u_1 is the root, and u_δ is the node v, where δ is the depth of v. Let t_i be the timestep in which u_i is executed; let $t_{\delta+1}$ be the last timestep in σ_p. For $i = 1, \ldots, \delta$, let I_i be the interval $\{t_i + 1, \ldots, t_{i+1}\}$.

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

Consider the case when u_{i+1} is a heavy node. After timestep t_i, Q_{out} may contain p nodes. Further, because access to Q_{in} requires constant time, the thread τ that contains u_i must be inserted into Q_{in} within a constant number of timesteps after t_i. During these timesteps, a constant number of scheduling steps may be executed, adding another $O(p)$ threads into Q_{out}. Thus, because Q_{out} is a FIFO, a total of $O(p)$ heavy nodes may be picked from Q_{out} and executed before u_{i+1}; all of these heavy nodes may be premature. However, once the thread τ is inserted into Q_{in}, the next scheduling step must find it in Q_{in}; since u_i is the last parent of u_{i+1} to execute, this scheduling step makes u_{i+1} available for scheduling. Thus, this scheduling step or any subsequent scheduling step must put u_{i+1} on Q_{out} before it puts any more premature nodes, because u_{i+1} has a lower 1DF-number. When u_{i+1} is picked from Q_{out} and executed by a worker processor, another $p - 1$ heavy premature nodes may get executed by the remaining worker processors in the same timestep, which, by definition, is timestep t_{i+1}. Thus, a total of $O(p)$ heavy premature nodes may be executed



(a) (b)



(c)

Figure 3.4 : (a) A simple program dag; all edges are shown here as solid lines for clarity. The 1DF-schedule for this dag is $s_1 = [a, b, c, d, e, f, g, h, i, j, k, l, m, n]$. For $p = 2$, a possible parallel schedule on processors P_1 and P_2 is shown in (c), and is represented as $s_p = [\{a\}, \{b, h\}, \{c, i\}, \{d, j\}, \{e, k\}, \{f, l\}, \{m, g\}, \{n\}]$. In this schedule, each batch is indicated as a curly line below the nodes in (c). The first node in each batch is a heavy node; thus, the heavy nodes are a, b, c, e, h and k, and are also shown with bold outlines in (a). For $j = 5$, the j-prefix of s_p is $\sigma_p = \{a, b, h, c, i, d, j, e, k\}$, and the corresponding serial prefix of s_1 is $\sigma_1 = \{a, b, c, d, e\}$. σ_p and σ_1 are shown in (b). Therefore, the premature nodes in σ_p, that is, the nodes in $\sigma_p - \sigma_1$, are h, i, j, and k. Of these, the heavy premature nodes are h and k.

in interval I_i. Similarly, since $v = u_\delta$ is the last non-premature node in σ_p, at most $O(p)$ heavy premature nodes get executed in the last interval I_δ. Because $\delta \leq D$, σ_p contains a total of $O(p \cdot D)$ heavy premature nodes. ∎

We can state the following lemma bounding the space allocation of a parallel schedule.

Lemma 3.3 *Let G be a program dag with depth D, in which every node allocates at most K space. If the* 1DF*-schedule for the dag requires S_1 space, then algorithm AsyncDF(K) generates a schedule s_p on p processors that requires at most $S_1 + O(K \cdot p \cdot D)$ space.*

Proof: Consider the j-prefix σ_p of s_p for any $j = 1, \ldots, \tau$, where τ is the length of s_p. According to Lemma 3.2, σ_p has $O(p \cdot D)$ heavy premature nodes. Recall that all the memory allocations are attributed to heavy nodes, and each thread is allotted a quota of K bytes in *AsyncDF(K)* every time it is scheduled. Since no single node in the dag allocates more than K bytes of memory, every batch in the thread accounts for at most K memory. Thus, every heavy node allocates at most K bytes and light nodes allocate no memory. Therefore, the total memory allocated by all the premature nodes is at most $O(K \cdot p \cdot D)$. Further, the total memory allocated by non-premature nodes cannot exceed the serial space requirement S_1, and therefore, the total memory allocated by all the nodes in σ_p cannot exceed $S_1 + O(K \cdot p \cdot D)$. Thus, we have bound the space allocated in the first j timesteps of s_p for any $j = 1, \ldots, \tau$. ∎

Handling allocations bigger than K. A transformation the program dag to handle allocations bigger than the memory threshold K was described in Section 3.2.2. Consider any heavy premature node v that allocates $m > K$ space. The m/K dummy threads inserted before it are executed before the allocation takes place. Having no-op nodes, they do not actually allocate any space, but are entitled to allocate a total of m space (K units each) according to algorithm *AsyncDF(K)*. Hence v can allocate these m units without exceeding the space bound in Lemma 3.3. With this transformation, a parallel computation with W work and D depth that allocates a total of S_a units of memory results in a dag with at most $W + S_a/K$ nodes and $O(D)$ depth. Therefore, using Lemma 3.3, we can state the following lemma.

Lemma 3.4 *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by the AsyncDF algorithm using $S_1 + O(K \cdot p \cdot D)$ space.*
∎

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

Lemma 3.5 *The space required by the scheduler is $O(p \cdot D)$.*

Proof: When a processor starts executing a scheduling step, it first empties Q_{in}. At this time, there can be at most $p - 1$ threads running on the other processors, and Q_{out} can have another p threads in it. The scheduler adds at most another p threads (plus one scheduling thread) to Q_{out}, and no more threads are added to Q_{out} until the next scheduling step. Since all the threads executing on the processors can end up in Q_{in}, Q_{in} and Q_{out} can have a total of at most $3p$ threads between them at any time.

Finally, we bound the number of entries in $\mathcal{R}$, which has one entry for each active thread. At any stage during the execution, the number of active threads is at most the number of premature nodes executed, plus the maximum number of threads in Q_{out} (which is $2p+1$), plus the maximum number of active threads in the 1DF-schedule. Any step of the 1DF-schedule can have at most D active threads, since it executes threads in a depth-first manner. Since the number of premature nodes is at most $O(p \cdot D)$, $\mathcal{R}$ has at most $O(p \cdot D) + (2p+1) + D = O(p \cdot D)$ threads. Since each thread's state can be stored using a constant c units of memory[3], the total space required by the three scheduling queues is $O(c \cdot p \cdot D) = O(p \cdot D)$. ∎

Using Lemmas 3.4 and 3.5, we can now state the following bound on the total space requirement of the parallel computation.

Theorem 3.6 *A computation of depth D and work W, which requires S_1 space to execute on one processor, is executed on p processors by the AsyncDF(K) algorithm using $S_1 + O(K \cdot p \cdot D)$ space (including scheduler space).* ∎

When the memory threshold K is a constant, algorithm *AsyncDF(K)* executes the above computation on p processors using $S_1 + O(p \cdot D)$ space.

3.3.3 Time bound

Finally, we bound the time required to execute the parallel schedule generated by algorithm *AsyncDF(K)* for a special case; Section 3.4.2 analyzes the time bound in the general case. In this special case, we assume that the worker processors never have to wait for the scheduler to add ready threads to Q_{out}. Thus, when there are r ready threads in the system, and n processors are idle, Q_{out} has at least $\min(r, n)$ ready threads. Then $\min(r, n)$ of the idle processors are guaranteed to pick ready threads from Q_{out} within a constant number of timesteps. We can show that the time required for such an execution is within a constant factor of the time required to execute a greedy schedule. A ***greedy*** schedule is one in which at every timestep, if n nodes are ready, $\min(n, p)$ of them get executed. Previous results have shown that greedy schedules for dags with W nodes and D depth require at most $W/p + D$ timesteps to execute [26]. Our transformed dag has $W + S_a/K$ nodes and $O(D)$ depth. Therefore, we can show that our scheduler requires $O(W/p + S_a/pK + D)$ timesteps to execute on p processors. When the allocated space S_a is $O(W)$, the number of timesteps required is $O(W/p + D)$. For a more in-depth analysis of the running time that includes the cost of a parallelized scheduler in the general case, see Section 3.4.2.

3.4 A Parallelized Scheduler

The time bound proved in Section 3.3.3 does not include scheduling overheads. The scheduler in the *AsyncDF* algorithm is a serial scheduler, that is, only one processor can be executing the

[3] Recall that a thread allocates stack and heap data from the global pool of memory that is assigned to it every time it is scheduled; this data is hence accounted for in the space bound proved in Lemma 3.4. Therefore, the thread's state here refers simply to its register contents.

scheduler() procedure at a given time. However, the amount of work involved in a scheduling step, and hence the time required to execute the scheduler() procedure, increases with the number of processors p. This can cause idle worker processors to wait longer for ready threads to appear Q_{out}. Therefore, the scheduler must be parallelized to scale with the number of processors. In this section, I describe a parallel implementation of the scheduler and analyze its space and time costs. I show that a computation with W work and D depth can be executed in $O(W/p+S_a/Kp+D\cdot\log p)$ time and $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space on p processors; these bounds include the overheads of the parallelized scheduler. The additional $\log p$ term is due to the parallel prefix operations executed by the scheduler.

Only a theoretical description of a parallelized scheduler is presented in this dissertation. The experimental results presented in subsequent chapters have been obtained using a serial scheduler; the results demonstrate that a serial scheduler provides good performance on a moderate number of processors.

3.4.1 Parallel implementation of a lazy scheduler

Instead of using scheduler threads to periodically (and serially) execute the *scheduler* procedure as shown in Figure 3.2, we devote a constant fraction αp of the p processors to it (where $0 < \alpha < 1$). The remaining $(1-\alpha)p$ processors always execute as workers. To amortize the cost of the scheduler, we place a larger number of threads (up to $p \log p$ instead of p) into Q_{out}. $\mathcal{R}$ is implemented as an array of threads, stored in decreasing order of priorities from left to right.

As described in Section 3.2.2, threads are forked lazily; when a thread reaches a fork instruction, it is simply marked as a ***seed*** thread. At a later time, when its child threads are to be scheduled, they are created from the seed and placed to the immediate left of the seed in order of their priorities. Each child thread has a pointer to its parent. When all child threads have been created, the parent (seed) thread once again becomes a regular thread. All the child threads get created from the seed before the parent thread is next scheduled, since they have higher priorities than the parent. Thread deletions are also performed lazily: every thread that terminates is simply marked in $\mathcal{R}$ as a ***dead*** thread, to be deleted in some subsequent timestep.

The synchronization (join) between child threads of a forking thread is implemented using a fetch-and-decrement operation on a synchronization counter associated with the fork. Each child that terminates marks itself as dead and decrements the counter by one. The last child thread among its siblings to terminate (the one that decrements the counter to zero) switches on a special ***last-child*** flag in its entry.

Unlike the serial scheduler described in Section 3.2, suspended threads (threads waiting to synchronize with their child threads) are not stored in $\mathcal{R}$. Every other active thread in this implementation has an entry in $\mathcal{R}$, as do dead threads that are yet to be deleted. Suspended threads found in Q_{in} are not put back into $\mathcal{R}$; they are stored elsewhere[4], and their entries in $\mathcal{R}$ are marked as dead. A suspended thread is subsequently inserted back into $\mathcal{R}$ in place of the child thread that has the last-child flag set. If a parent reaches the point of synchronization with its child threads after all the child threads have terminated, the parent does not suspend itself, and retains its entry in $\mathcal{R}$.

[4]A suspended threads is not stored in any data structure; it just exists as a thread data structure allocated off the heap. It is subsequently accessed through the parent pointer in one of its child threads.

For every ready (or seed) thread T that has an entry in $\mathcal{R}$, we use a nonnegative integer $c(T)$ to denote the number of ready threads T represents. Then for every seed T, $c(T)$ is the number of child threads still to be created from it, plus one (for itself). For every other ready thread in $\mathcal{R}$, $c(T) = 1$, since it only represents itself.

The scheduler() procedure from Figure 3.2 can now be replaced by a while loop that runs until the entire computation has been executed. Each iteration of this loop, which we call a ***scheduling iteration***, is executed in parallel by only the αp scheduler processors. Therefore, it need not be protected by a scheduler lock as in Figure 3.2. Let r be the total number of ready threads represented in $\mathcal{R}$ after threads from Q_{in} are moved to $\mathcal{R}$ at the beginning of the iteration, and let $|Q_{out}|$ be the number of threads currently in Q_{out} when the iteration starts. The scheduling iteration of a lazy scheduler is then described in Figure 3.5.

begin scheduling iteration
Let $q_o = min(r, \, p \log p - |Q_{out}|)$.

1. Remove the set $\mathcal{T}$ of all threads in Q_{in} from Q_{in}. Except for suspended threads, move all threads in $\mathcal{T}$ to $\mathcal{R}$, that is, update their states in $\mathcal{R}$. For each suspended thread in $\mathcal{T}$, mark its entry in $\mathcal{R}$ as dead.
2. For every dead thread in $\mathcal{T}$, if its last-child flag is set, and its parent is suspended, replace its entry in $\mathcal{R}$ with the parent thread and reactivate the parent (mark it as ready).
3. Delete all the dead threads up to the leftmost $(q_o + 1)$ ready or seed threads.
4. Perform a prefix-sums computation on the $c(T)$ values of the leftmost q_o ready or seed threads to find the set C of the leftmost q_o ready threads represented by these threads. For every thread in C that is implicitly represented by a seed, create an explicit entry for the thread in $\mathcal{R}$, marking it as a ready thread.
5. Move the threads in the set C from $\mathcal{R}$ to Q_{out}, leaving stubs in $\mathcal{R}$ to mark their positions.

end scheduling iteration

Figure 3.5: The sequence of steps executed in each scheduling iteration. The αp scheduling processors continuously execute scheduling iterations until the execution is completed.

Consider a child thread T that is the last to terminate amongst its siblings. If T is not the rightmost (lowest priority) thread amongst its siblings, then some of T's siblings, which have already terminated, may be represented as dead threads to its right in $\mathcal{R}$. If T is replaced by its parent thread, the parent has a lower priority than these dead siblings of T to its immediate right. Thus, due to lazy deletions, active threads in $\mathcal{R}$ may be out of order with respect to one or more dead threads to their immediate right. However, the scheduler deletes all dead threads up to the first $(q_o + 1)$ ready or seed threads. This ensures that all dead threads to the immediate right of any ready thread (or seed representing a ready thread) T are deleted before T is scheduled. Therefore, no descendents of a thread may be created until all dead threads out of order with respect to the thread are deleted. Thus, at any time during the execution, a thread may remain out of order in $\mathcal{R}$ with respect to only the dead threads to its immediate right.

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

Lemma 3.7 *For any $0 < \alpha < 1$, a scheduling iteration that deletes n dead threads runs in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ time on αp processors.*

Proof: Let $q_o = min(r, p \log p - |Q_{out}|)$ be the number of threads the scheduling iteration must move to Q_{out}; then, $q_o \leq p \log p$. We analyze the time required for each step of the scheduling iteration described in Figure 3.5. Recall that $\mathcal{T}$ is the set of all threads in Q_{in} at the start of the scheduling iteration.

1. At the beginning of the scheduling iteration, Q_{in} contains at most $2p \log p + (1-\alpha)p$ threads, that is, $|\mathcal{T}| \leq 2p \log p + (1-\alpha)p$. Since each of these threads has a pointer to its stub in $\mathcal{R}$, αp processors can move the threads in $\mathcal{T}$ to $\mathcal{R}$ and update their states in $O(\frac{\log p}{\alpha})$ time.

2. αp processors can replace dead threads in $\mathcal{T}$ that have their last-child flag set with their suspended parents in $O(\frac{\log p}{\alpha})$ time.

3. Let T be the $(q_o + 1)^{th}$ ready or seed thread in $\mathcal{R}$ (starting from the left end). The scheduler needs to delete all dead threads to the left of T. In the worst case, all the stubs are also to the left of T in $\mathcal{R}$. However, the number of stubs in $\mathcal{R}$ is at most $2p \log p + (1-\alpha)p$, that is, one for each live thread. Since there are n dead threads to the left of T, there are a total of at most $n + 2p \log p + (1-\alpha)p$ threads to the left of T. Therefore, the n threads can be deleted from $n + 2p \log p + (1-\alpha)p$ threads in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ timesteps on αp processors.

4. After the deletions, the leftmost $q_o \leq p \log p$ ready threads are among the first $3p \log p + (1-\alpha)p$ threads in $\mathcal{R}$; therefore the prefix-sums computation will require $O(\frac{\log p}{\alpha})$ time.

5. Finally, q_o new child threads can be created and added in order to the left end of $\mathcal{R}$ in $O(\frac{\log p}{\alpha})$ time.

All deletions and additions are performed near the left end of $\mathcal{R}$, which are simple parallel operations in an array[5]. Thus, the entire scheduling iteration runs in $O(\frac{n}{\alpha p} + \frac{\log p}{\alpha})$ time. ∎

3.4.2 Space and time bounds using the parallelized scheduler

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

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

Theorem 3.8 *Let S_1 be the space required by a* 1DF*-schedule for a computation with work W and depth D, and let S_a be the total space allocated in the computation. The parallelized scheduler with a memory threshold of K units, generates a schedule on p processors that requires $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space and $O(W/p + S_a/pK + D \cdot \log p)$ time to execute.* ∎

These time and space bounds include scheduling overheads. The time bound is derived by counting the total number of timesteps during which the worker processors may be either idle, or busy executing actions. The space bound is proved using an approach similar to that used in Section 3.3.2. When the total space allocated $S_a = O(W)$, the time bound reduces to $O(W/p + D \cdot \log p)$.

3.5 Summary

In this chapter, I have presented *AsyncDF*, an asynchronous, space-efficient scheduling algorithm. The algorithm prioritizes threads at runtime by their serial execution order, and preempts threads before they allocate too much memory. I first described a simple version of *AsyncDF* which serializes the scheduling. This version was shown to execute a nested-parallel computation with a depth of D and a serial space requirement of S_1 on p processors using $S_1 + O(K \cdot p \cdot D)$ space; here, K is the value of the memory threshold used by the scheduler. (This version is used in the runtime system described in Chapter 4.) This chapter then presented a version of algorithm *AsyncDF* in which the scheduling operations are fully parallelized. A parallel computation can be implemented with such a scheduler using $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space and $O(W/p + S_a/p + D \log p)$ time (including scheduler overheads); here W is the work of the computation, and S_a is the total space allocated. The next two chapters describe experiments with implementing and evaluating the *AsyncDF* algorithm in the context of two runtime systems.

Chapter 4

Implementation and Evaluation of Algorithm *AsyncDF*

Chapter 3 presented algorithm *AsyncDF*, a provably space and time efficient scheduling algorithm for lightweight threads. However, the theoretical analysis only provides upper (worst-case) bounds on the space and time requirements of a multithreaded computation. The actual performance for real parallel programs can be significantly better than these worst-case bounds. Therefore, other space-efficient schedulers with asymptotically higher space bounds than algorithm *AsyncDF*, may still result in lower space requirements in practice. Further, the performance of algorithm *AsyncDF* on a real parallel machine may be affected by factors not accounted for in the assumptions made during its analysis[1]. Therefore, an experimental evaluation is required to test whether algorithm *AsyncDF* is indeed more space efficient than previous approaches, and whether it does result in good time performance. This chapter describes the implementation and evaluation of a space-efficient runtime system that uses algorithm *AsyncDF* (as described in Figure 3.2) to schedule lightweight threads. The prototype runtime system was built specifically to evaluate the scheduling algorithm, and to compare its performance with previous space-efficient approaches.

The chapter begins by presenting an overview of the runtime system implementation (Section 4.1), followed by a brief description of each parallel benchmark used to evaluate the system (Section 4.2). Sections 4.3 and 4.4 then present experimental results measuring the running times and the memory requirements of the benchmarks. For each benchmark, I compare its space and time performance on my runtime system with its performance on Cilk [25], a previously existing space-efficient system. The results indicate that algorithm *AsyncDF* is more effective in controlling memory requirements, without compromising time performance. This chapter then describes the experimental trade-off between running time and memory requirement in algorithm *AsyncDF*(K), which can be adjusted by varying the value of the commandline parameter (*i.e.*, the memory threshold) K. Finally, Section 4.5 summarizes the results presented in this chapter.

[1] In particular, the analysis ignores the effects of the memory hierarchies found on real machines.

4.1 The Runtime system

The runtime system has been implemented on a 16-processor SGI Power Challenge, which has a shared memory architecture with processors and memory connected via a fast shared-bus interconnect. The bus has a bandwidth of 1.2 GB per second with a 256-bit wide data bus and a separate 40-bit wide address bus. Each processor is a 195 MHz R10000 with a 2 MB secondary cache. Since the number of processors on this architecture is not very large, a serial implementation of the scheduler as described in Section 3.2 does not create a bottleneck in the runtime system. The set $\mathcal{R}$, which is accessed only by a single scheduling thread at any time, is implemented as a simple, doubly-linked list to allow insertions and deletions of intermediate nodes. The FIFO queues Q_{in} and Q_{out}, which are accessed by the scheduler and the workers, are required to support concurrent enqueue and dequeue operations. They are implemented using variants of lock-free algorithms based on atomic fetch-and-Φ primitives [108].

The parallel programs executed on the runtime system have been explicitly hand-coded in the continuation-passing style, similar to the code generated by the Cilk-1 preprocessor[2] [25]. Each continuation points to a C function representing the next computation of a thread, and a structure containing all its arguments. These continuations are created dynamically and moved between the queues. A thread that has reached a fork instruction for a parallel loop holds a pointer to the function representing the loop body, and the index of the next child thread to be forked. The scheduler lazily creates separate entries for each child thread when they are to be inserted into Q_{out}. The entries are placed in $\mathcal{R}$ to the immediate left of the parent thread. As described in Chapter 3, a synchronization counter is used to implement a join between child threads.

One kernel thread is forked per processor at the start of the computation to execute the task of a worker. Each worker takes a continuation off Q_{out}, and simply applies the function pointed to by the continuation, to its arguments. The high-level program is manually broken into such functions at points where it executes a parallel fork, a recursive call, or a memory allocation; thus, each function corresponds to a batch of instructions (nodes) in a thread. When a worker finds a scheduling thread on Q_{out}, it executes one scheduling step, as described in Section 3.2.

For nested parallel loops, iterations of the innermost loop are grouped by hand into equal-sized chunks (the chunk size can be specified at runtime), provided it does not contain calls to any recursive functions. It should be possible to automate such coarsening statically with compiler support, or at runtime using a dynamic loop scheduling scheme [84, 128, 158, 97]. Scheduling a chunk at a time improves performance by reducing scheduling overheads and providing good locality, especially for fine-grained iterations.

Recall that algorithm *AsyncDF*(K) prescribes adding m/K dummy threads in parallel before a large memory allocation of m bytes. In this implementation, I instead added a ***delay counter*** of value m/K units before the memory allocation to represent the dummy threads. Thus, when the scheduler encounters a thread among the leftmost p ready threads in $\mathcal{R}$ with a non-zero value of the delay counter, it simply decrements the value of the counter by the appropriate value. Each decrement by one unit represents the scheduling of one dummy thread, and accordingly, fewer real threads are added to Q_{out}. For example, on p processors, if the scheduler finds that the leftmost ready thread T in $\mathcal{R}$ has a delay counter with value greater than or equal to p, it decrements the

[2] A preprocessor-generated version of a program on this system is expected to have similar efficiency as the straightforward hand-coded version.

counter by p units and does not add any ready threads to Q_{out} in that scheduling step, even if $\mathcal{R}$ contains ready threads to the right of T. Since the scheduler decrements delay counters by up to p units, this corresponds to the dummy threads being forked and executed in parallel by the worker processors. When the delay counter of a ready thread reaches zero, the thread may be moved to Q_{out}, and the memory allocation, if needed, is performed as soon as the thread is next scheduled on a processor.

4.2 Benchmark programs

The runtime system was designed to efficiently execute programs with dynamic or irregular parallelism. I implemented five such parallel programs to evaluate the runtime system. This section briefly describes the implementation of these programs, along with the problem sizes and thread granularities used in the experiments.

1. Blocked recursive matrix multiply (Rec MM)

This program multiplies two dense $n \times n$ matrices using a simple recursive divide-and-conquer method, and performs $O(n^3)$ work. The recursion stops when the blocks are down to the size of 64×64, after which a standard, row-column matrix multiply is executed serially. Alternatively, a fast, machine-specific routine such as BLAS3 [54] could be utilized at the leafs of the recursion tree. The code is similar to the pseudocode shown in Figure 1.1. This algorithm significantly outperforms the row-column matrix multiply for large matrices (e.g., by a factor of over 4 for 1024×1024 matrices) because its use of blocks results in better cache locality. At each stage of the recursion, temporary storage is first allocated to store intermediate results, which are generated during subsequent recursive calls. Then a new child thread is forked to execute each of the eight recursive calls. When the child threads terminate, the results in the temporary storage are added to the results in the output matrix, and then the temporary storage is deallocated. Note that the allocation of temporary storage can be avoided, but this leads to either significantly increased code complexity or reduced parallelism. The results reported are for the multiplication of two 1024×1024 matrices of double-precision floats. Although the code is restricted to matrix sizes that are powers of 2 (or, more precisely, $2^n \times b$ for some integer n, where b is the block size), it can be extended to efficiently multiply matrices of other sizes [63].

2. Strassen's matrix multiply (Str MM)

The DAG for this algorithm is very similar to that of the blocked recursive matrix multiply (Rec MM), but performs only $O(n^{2.807})$ work and makes seven recursive calls at each step [151]. As with Rec MM, a new thread is forked in parallel for each recursive call, and the parallel recursion is terminated when the matrix size is reduced to 64×64. A simple, serial algorithm is used to multiply the 64×64 matrices at the leafs of the recursion tree. This version of Strassen's matrix multiply, which dynamically forks lightweight threads, results in significantly simpler code compared to when the work is statically partitioned among the processors. The sizes of the input matrices multiplied were the same as for Rec MM. However, unlike in Rec MM, instead of using a separate output matrix for the results, the results are stored in one of the input matrices.

3. Fast multipole method (FMM)

FMM is an N-body algorithm that calculates the forces between N bodies in $O(N)$ work [72]. I have implemented the most time-consuming phases of the algorithm, which are a bottom-up traversal of the octree followed by a top-down traversal[3]. In the top-down traversal, for each level of the octree, the forces on the cells in that level due to their neighboring cells are calculated in parallel. For each cell, the forces over all its neighbors are also calculated in parallel, for which temporary storage needs to be allocated. This storage is freed when the forces over the neighbors have been summed up to get the resulting force on that cell. With two levels of parallelism, the structure of this code looks very similar to the pseudocode described in Chapter 1 (Figure 1.3). The experiments were conducted on a uniform octree with 4 levels (8^3 leaves), using 5 multipole terms for force calculation. Each thread in the bottom-up phase computes the multipole expansion of one cell from its child cells. In the top-down phase, each thread handles interactions of a cell with up to 25 of its neighbors.

4. Sparse matrix-vector multiply (Sparse MV)

This program multiplies an $m \times n$ sparse matrix with a $n \times 1$ dense vector. The dot product of each row of the matrix with the vector is computed to get the corresponding element of the resulting vector. There are two levels of parallelism: over each row of the matrix, and over the elements of each row multiplied with the corresponding elements of the vector to calculate the dot product. Once the elements of a row are multiplied with the corresponding elements of the dense vector, the results are summed in parallel using a tree-based algorithm. In my experiments, I used $m = 20$ and $n = 1500000$, and 30% of the elements were non-zeroes. Using a large value of n provides sufficient parallelism within a row, but using large values of m leads to a very large size of the input matrix, making the amount of dynamic memory allocated in the program negligible in comparison[4]. Each thread computed the products of up to 10,000 pairs of elements in each row-vector dot product.

5. ID3 decision tree builder

This algorithm by Quinlan [132] builds a decision tree from a set of training examples with discrete attributes in a top-down manner, using a recursive divide-and-conquer strategy. At the root node, the attribute that best classifies the training data is picked, and recursive calls are made to build subtrees, with each subtree using only the training examples with a particular value of that attribute. Each recursive call is made in parallel, and the computation of picking the best attribute at a node, which involves counting the number of examples in each class for different values for each attribute, is also parallelized. Temporary space is allocated to store the subset of training examples used to build each subtree, and is freed once the subtree is built. I built a tree from 4 million test examples, each with 4 multi-valued attributes[5]. The parallel recursion was terminated when the

[3] The Pthreads version described in Chapter 5 includes all the phases in the algorithm.

[4] The input matrix was randomly generated; the Pthreads version of the algorithm described in Chapter 5 uses input generated from a real finite elements problem.

[5] As with Sparse MV, the input was randomly generated; the Pthreads version of this algorithm described in Chapter 5 uses a real input data set, and is extended significantly to handle continuous attributes.

number of examples fell under 10,000, after which the recursion was implemented serially.

4.3 Time performance

This section presents the results of experiments that measured and compared the running times of the five parallel benchmarks on my runtime system, with the same benchmarks written in version 5.0 of the Cilk programming language [25]. Cilk is a space-efficient multithreading system that guarantees a space bound of $p \cdot S_1$ for programs with a serial space requirement of S_1 on p processors. The Cilk programs were written at a high level with simple, fork-join style parallelism using the Cilk instructions spawn and sync. Cilk uses a Cilk-to-C translator to perform data flow analysis and convert the high-level programs into a form equivalent to the continuation passing style that I use to write programs for my runtime system. Although Cilk supports more fine grained threads, to make a valid comparison, the thread granularities in the Cilk programs were set to be identical to those in my original programs.

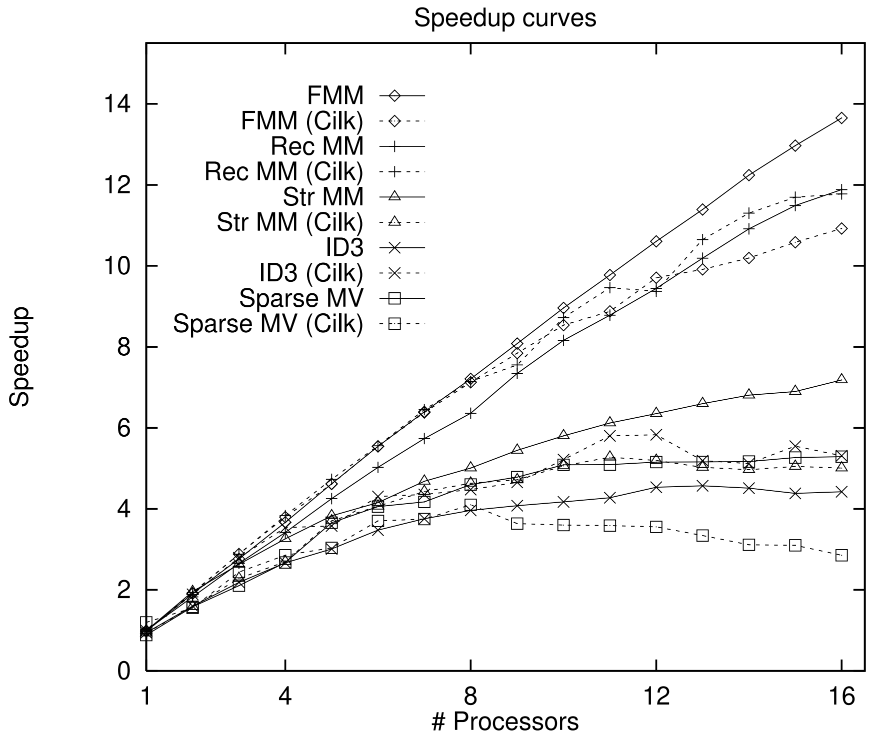


Figure 4.1 : The speedups achieved on up to 16 R10000 processors of a Power Challenge machine, using a value of K=1000 bytes. The speedup on p processors is the time taken for the serial C version of the program divided by the time for the multithreaded version to run on p processors. For each application, the solid line represents the speedup using my system, while the dashed line represents the speedup using the Cilk system. All programs were compiled using `gcc -O2 -mips2`.

Figure 4.1 shows the speedups for the five benchmarks on up to 16 processors. The speedup for each program is with respect to its efficient serial C version, which does not use my runtime system. Since the serial C program runs faster than my runtime system on a single processor, the speedup shown for one processor is less than 1. However, for all the programs, it is close to 1, implying that the overheads in the system are low. The timings on my system include the overheads due to delays introduced before large allocations. A value of $K = 1000$ bytes was used for all the benchmarks in these experiments. Figure 4.1 also shows the speedups for the same programs running on Cilk. The timings show that the performance on my system is comparable with that on Cilk. The memory-intensive programs such as sparse matrix-vector multiply do not scale well on either system beyond 12 processors; their performance is probably affected by bus contention as the number of processors increases.

Figure 4.2 shows the breakdown of the running time for one of the programs, blocked recursive matrix multiplication. The results show that the percentage of time spent waiting for threads to appear in Q_{out} increases as the number of processors increases (since I use a serial scheduler). A parallel implementation of the scheduler, such as one described in Chapter 3 (Section 3.4), will be more efficient for a larger number of processors.

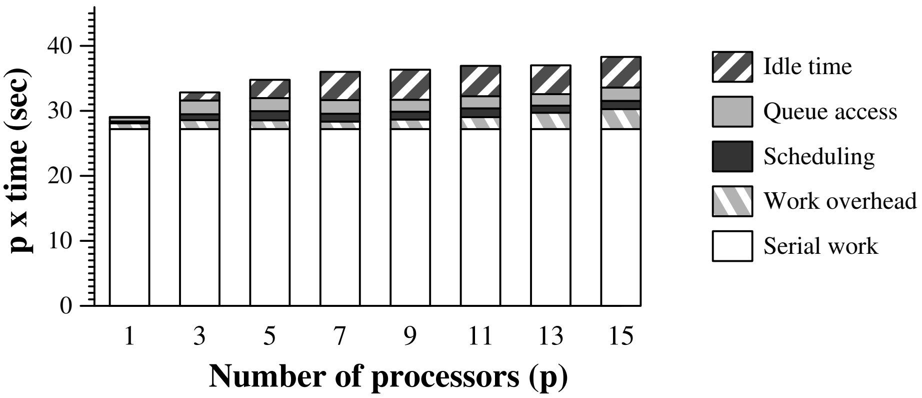


Figure 4.2 : The total processor time (the running time multiplied by the number of processors p) for blocked recursive matrix multiplication. "Serial work" is the time taken by a single processor executing the equivalent serial C program. For ideal speedups, all the other components would be zero. The other components are overheads of the parallel execution and the runtime system. "Idle time" is the total time spent waiting for threads to appear in Q_{out}; "queue access" is the total time spent by the worker processors inserting threads into Q_{in} and removing them from Q_{out}. "Scheduling" is the total time spent as the scheduler, and "work overhead" includes overheads of creating continuations, building structures to hold arguments, and (de)allocating memory from a shared pool of memory, as well as the effects of the delay counters, cache misses and bus contention.

4.4 Space performance

This section presents results of experiments that evaluate the space efficiency of algorithm *AsyncDF*, followed by an empirical demonstration of the space-time trade-off that exists in the algorithm.

The memory usage of each application was measured as the high water mark of memory allocation during the execution of the program. Figure 4.3 shows the memory usage for the five applications. Here three implementations for each program are compared—one in Cilk and the other two using my runtime system. Of the two implementations on my system, one inserts delays

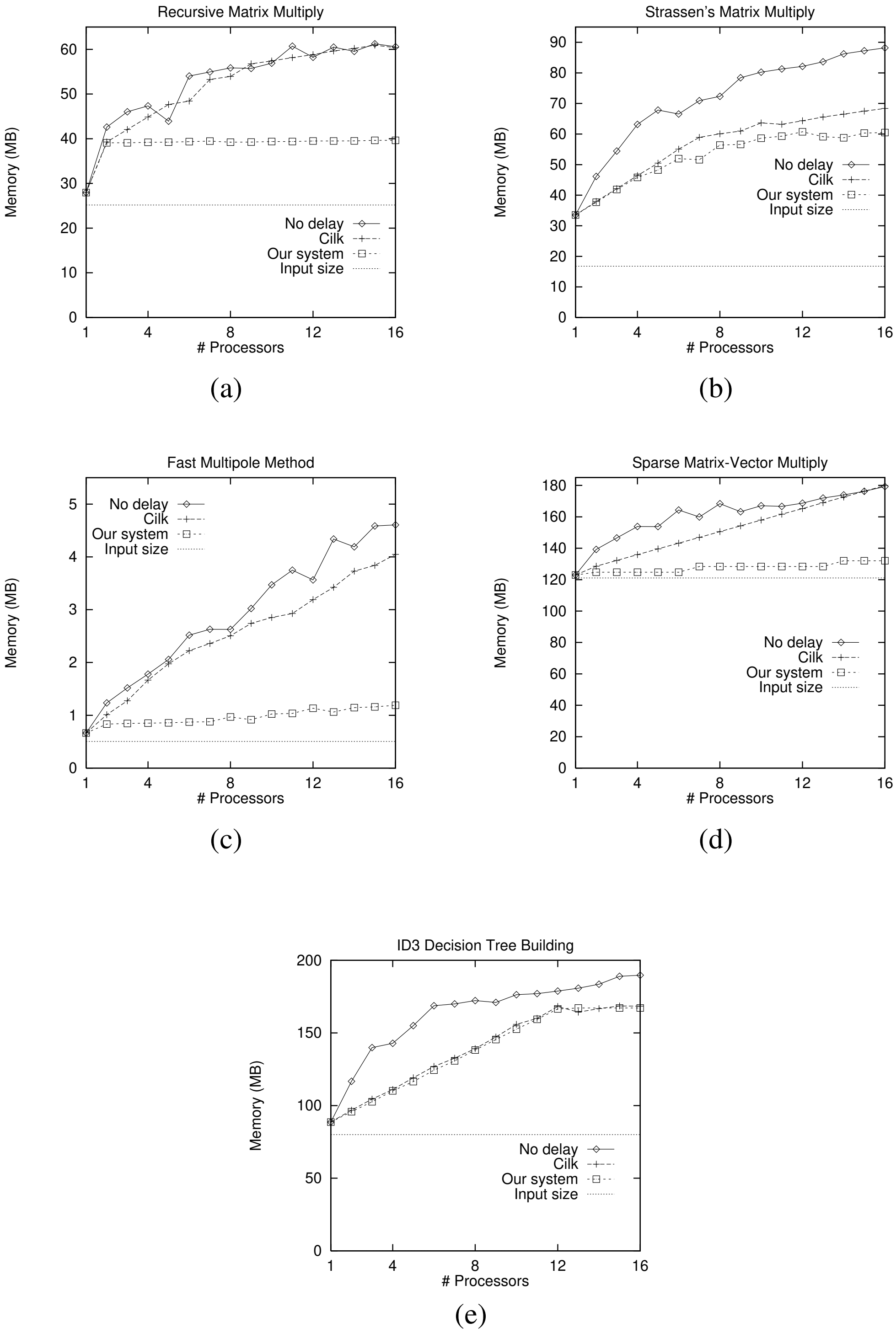


Figure 4.3: The memory requirements of the parallel programs. For $p = 1$ the memory usage shown is for the serial C version. The memory usage of each program when the big memory allocations are delayed (with $K = 1000$), is compared with when they are allowed to proceed without any delay, as well as with the memory usage on Cilk. The version without the delay on my system (labeled "No delay") is an estimate of the memory usage resulting from previous scheduling techniques. These results show that delaying big allocations significantly changes the order of execution of the threads, and results in much lower memory usage, especially as the number of processors increases.

before large allocations, while the other does not. For all the five programs implemented, the version without the delay results in approximately the same space requirements as would result from scheduling the outermost level of parallelism. For example, in Strassen's matrix multiplication, algorithm *AsyncDF* without the delay would allocate temporary space required for p branches at the top level of the recursion tree before reverting to the execution of the subtree under the first branch. On the other hand, scheduling the outer parallelism would allocate space for the p branches at the top level, with each processor executing a subtree serially. Hence algorithm *AsyncDF* without the delay is used here to estimate the memory requirements of previous techniques [41, 83], which schedule the outer parallelism with higher priority. Cilk uses less memory than this estimate due to its use of randomization: an idle processor steals the topmost thread (representing the outermost parallelism) from the private queue of a randomly picked processor; this thread may not represent the outermost parallelism in the entire computation. A number of systems [30, 70, 77, 92, 111, 115, 143, 145, 159] use a work stealing strategy similar to that of Cilk. The results show that when big allocations are delayed, algorithm *AsyncDF* results in a significantly lower memory usage, particularly as the number of processors increases. A notable exception is the ID3 benchmark, for which our scheduler results in a similar space requirement as that of Cilk. This is because the value of K ($= 1000$ bytes) is too large to sufficiently delay the large allocations of space until higher priority threads become ready[6]. In these graphs, I have not compared *AsyncDF* to naive scheduling techniques such as breadth-first schedules, which have much higher memory requirements.

Space-Time Tradeoff

The space bound for algorithm *AsyncDF*(K) derived in Chapter 3 (Section 3.3) increases with K, the value of the memory quota alloted to each thread every time it is scheduled. At the same time, the number of dummy threads (*i.e.*, the value of the delay counter) introduced before a large allocation depends on the value of K. Further, the smaller the value of K, the more often is a thread preempted due to exhaustion of its memory quota, resulting in lower locality and higher scheduling overheads. Hence there exists a trade-off between memory usage and running time, that can be adjusted by varying the value of K. For example, Figure 4.4 shows how the running time and memory usage for blocked recursive matrix multiplication are affected by K. For small K, many dummy threads (*i.e.*, large delays) are introduced, and threads are preempted often. This results in poor locality and high scheduling overheads, resulting in a high running time. However, the execution order is close to a 1DF-schedule, and therefore, the memory requirement is low. In contrast, for large K, very few dummy threads are inserted, and threads are preempted less often. This results in low running time, but higher memory requirements. K=500-2000 bytes results in both good performance and low memory usage.

For all the five programs implemented, the trade-off curves looked similar; however, they may vary for other parallel programs. A default value of $K = 1000$ bytes resulted in a good balance between space and time performance for the five test programs, although in practice it might be

[6]This problem arose because some inherently parallel operations in the algorithm had been coded serially; this serialization, although not necessary, simplified the writing of the code in the continuation-passing style required by the system. Consequently, the depth of the program increased, and there was often not a sufficient amount of inner parallelism to keep processors busy, forcing them to execute the outer parallelism instead.

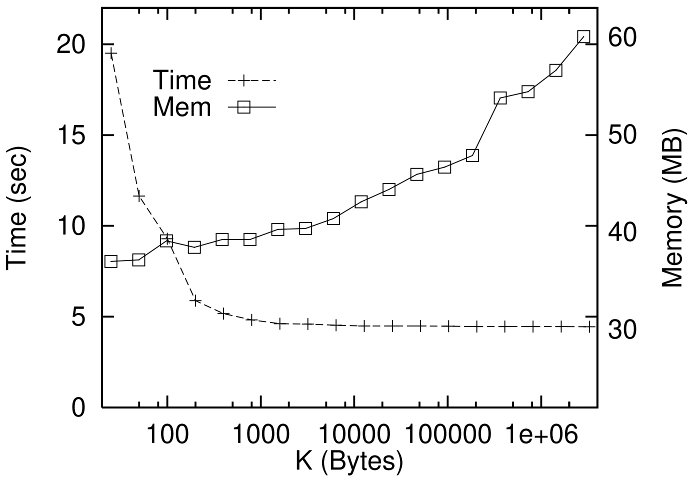


Figure 4.4 : The variation of running time and memory usage with K (in bytes) for multiplying two 1024×1024 matrices using blocked recursive matrix multiplication on 8 processors. For very small K, running time is high while the memory requirement is low. In contrast, for large K, the running time is low while the memory requirement is high. K=500-2000 bytes results in both good performance and low memory usage.

useful to allow users to tune the parameter for their needs.

4.5 Summary

In this chapter, I have described the implementation of a simple, multithreading runtime system for the SGI Power Challenge SMP. The implementation uses algorithm *AsyncDF* (presented in Chapter 3) to schedule lightweight threads. Experimental results for nested-parallel benchmarks on this system indicate that, compared to previous schedulers (including a provably space-efficient scheduler), *AsyncDF* typically leads to a lower space requirement. When innermost, fine-grained loops (or recursive calls) in the benchmarks were grouped into chunks, the time performance of algorithm *AsyncDF* was shown to be comparable to that of an efficient thread scheduler. I also experimentally demonstrated that the algorithm provides a trade-off between running time and memory requirement, by adjusting the value of the memory threshold. The next chapter describes the implementation of algorithm *AsyncDF* in the context of another runtime system, namely, a commercial Pthreads package.

Chapter 5

A Space-Efficient Implementation of Pthreads

The experimental results presented in Chapter 4 indicate that algorithm *AsyncDF* does indeed provide good space and time performance in practice. The runtime system described in that chapter was built specifically to implement algorithm *AsyncDF*, and is restricted to purely nested parallel computations. This chapter describes experiments with implementing a slight variation of algorithm *AsyncDF* in the context of a more general threads library, namely, Posix standard threads or Pthreads [88].

The chapter begins by motivating the choice of Pthreads as a medium for my experiments with space efficient schedulers (Section 5.1). Section 5.2 describes a particular Pthreads implementation, namely the native Pthreads implementation on Solaris 2.5, which I use in all my subsequent experiments. The section also presents experimental results of implementing a simple parallel benchmark, namely, dense matrix multiply, using the existing Pthreads implementation; the algorithm used involves dynamic creation of a large number of Pthreads, and dynamic memory allocation. The benchmark exhibits poor space and time performance, mainly due to the use of a space-inefficient scheduler. Section 5.3 then lists the modifications I made to the existing Pthreads implementation to make it space-efficient, presenting the performance of the benchmark after each modification. The final version of the implementation uses a variation of algorithm *AsyncDF* to schedule the threads. This chapter also describes and presents experimental results for the other parallel benchmarks; I use them to evaluate and compare the performance of the original Pthreads implementation with the modified Pthreads implementation (Section 5.4). All the benchmarks dynamically create and destroy large numbers of lightweight Pthreads. For a majority of these benchmarks, I started with a pre-existing, publicly available, coarse-grained version, which I modified to use large numbers of dynamic Pthreads. For such benchmarks, I also compare the performance of the rewritten version with the original, coarse-grained version. The coarse-grained versions are typically hand-coded to carefully partition and balance the load for good performance. The chapter ends with a brief discussion on selecting the appropriate thread granularity in Section 5.5. I show that algorithm *AsyncDF* does not handle finer thread granularities very efficiently, thereby motivating the need for using algorithm *DFDeques* to handle finer-grained threads. Finally, Section 5.6 summarizes this chapter.

The Pthread-based benchmarks used to evaluate the new space-efficient scheduler predominantly use a nested, fork-join style of parallelism; however, some of the benchmarks also make limited use of mutexes. The Pthread implementation itself makes extensive use of both Pthread mutexes and condition variables. Because the Pthreads library supports synchronization primitives like mutexes and condition variables, the class of computations that can be expressed using Pthreads are often not dag-deterministic. Therefore, defining the serial space requirement S_1 precisely for Pthread-based computations may not be possible. Nevertheless, the resulting space requirement for the parallel execution can be bounded in terms of the serial space requirement of the dag that represents the particular parallel execution. The experimental results presented in this chapter indicate that, in practice, the new scheduler does effectively limit the space requirement of the parallel benchmarks.

5.1 Motivation

The space-efficient scheduling algorithms presented in this dissertation could be applied in the context of several multithreaded systems besides Pthreads. There are, however, several reasons for choosing Pthreads as the platform to implement algorithm *AsyncDF* (as well as the *DFDeques* algorithm described in Chapter 7):

- The Pthread interface is a Posix standard that has recently been adopted by the vendors of a number of operating systems [53, 79, 87, 142, 153]. Pthread implementations are also freely available for a wide variety of platforms [131]. Therefore, Pthreads have now become a common medium for writing portable multithreaded code in general, and, in particular, they are popular for shared memory parallel programming.

- By selecting a Pthreads library as the medium for my experiments, I can directly use publicly available Pthread-based benchmarks to evaluate my schedulers. Similarly, the benchmarks that I code in a lightweight threaded style may be executed directly on any platform supporting a Pthreads implementation[1].

- The interface provides a fairly broad range of functionality. For example, Pthreads may synchronize using mutexes, condition variables, or joins. Therefore, my schedulers may be applied to a class of applications broader than those that are purely nested parallel.

- Most platforms that support Pthreads today are releasing lightweight, user-level implementations of Pthreads that allow thousands of threads to be expressed in the program. Examples of current platforms that schedule lightweight, user-level threads on top of kernel threads (to take advantage of multiple processors) include Digital UNIX, Solaris, IRIX, and AIX. In particular, I had access to the source code for the popular, native Pthreads implementation on Solaris. To my knowledge, this is one of the most efficient implementations of user-level Pthreads today.

- As with MPI, Pthreads adopts a library approach, that is, the use of Pthreads does not require adopting a new parallel programming language (and compiler front end). Therefore, Pthreads are gaining acceptance among the parallel programming community.

[1] For these benchmarks to execute efficiently, the Pthreads implementation must be lightweight (user-level) and must use a space-efficient scheduler.

Pthreads have two main disadvantages: the inability to support extremely lightweight thread operations, and the lack of syntactic sugar to simplify the thread interface. Multithreading languages with restricted functionality, or thread systems that can take advantage of a preprocessor or specialized compiler, are able to support extremely lightweight threads and higher-level thread interfaces [66, 70, 102]. The broad functionality and the library approach of Pthreads make an extremely lightweight implementation of Pthreads impossible, and results in a somewhat large and complex interface. For example, creating a user-level Pthread is typically two orders of magnitude more expensive than the cost of a function call, and requires the programmer to explicitly pack the thread arguments into a contiguous chunk of memory. Further, there are no interface functions to express multi-way forks or parallel loops; Pthreads only support a binary fork. However, I feel that these drawbacks are a small price to pay for the other advantages offered by Pthreads. Syntactic sugar to ease the task of forking Pthreads, or for providing multi-way forks can always be added through a preprocessor.

Despite several lightweight, user-level implementations of Pthreads, most programmers still write parallel programs in a coarse-grained style, with one Pthread per processor. The main reason is probably the lack of good schedulers that can efficiently execute a program with large numbers of lightweight, dynamic Pthreads. In particular, as I show in the next section, a typical Pthreads scheduler creates too many simultaneously active threads in such a program. These threads may all contend for stack and heap space, leading to poor space and time performance. However, this chapter subsequently shows that by using a space-efficient thread scheduler, an existing Pthreads implementation can efficiently execute programs with dynamic, lightweight threads.

5.2 The Solaris Pthreads Implementation

This section describes the native Pthreads implementation on Solaris 2.5, followed by some experiments measuring the performance of a parallel matrix multiply benchmark that uses Pthreads on Solaris. The experiments with the remaining benchmarks are described in Section 5.4.

The Solaris operating system contains kernel support for multiple threads within a single process address space [129]. One of the goals of the Solaris Pthreads implementation is to make the threads sufficiently lightweight so that thousands of them can be present within a process. The threads are therefore implemented by a user-level threads library so that common thread operations such as creation, destruction, synchronization and context switching can be performed efficiently without entering the kernel.

Lightweight, user-level Pthreads on Solaris are multiplexed on top of kernel-supported threads called LWPs. This assignment of the lightweight threads to LWPs is controlled by the user-level Pthreads implementation [150]. A thread may be either bound to an LWP (to schedule it on a system-wide basis) or may be multiplexed along with other unbound threads of the process on top of one or more LWPs. LWPs are scheduled by the kernel onto the available CPUs according to their scheduling class and priority, and may run in parallel on a multiprocessor. Figure 5.1 shows how threads and LWPs in a simple Solaris process may be scheduled. Process 1 has one thread bound to an LWP, and two other threads multiplexed on another LWP, while process 2 has three threads multiplexed on two LWPs. This chapter focuses on the policy used to schedule unbound Pthreads at a given priority level on top of LWPs.

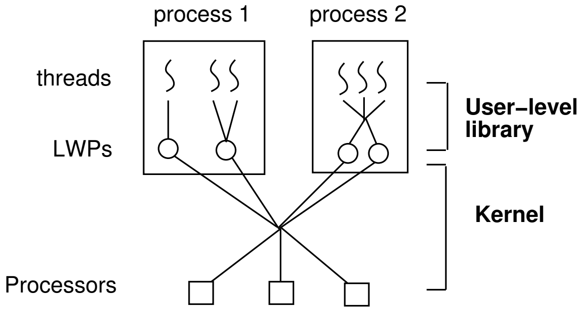


Figure 5.1: Scheduling of lightweight Pthreads and kernel-level LWPs in Solaris. Threads are multiplexed on top of LWPs at the user level, while LWPs are scheduled on processors by the kernel.

Operation	Create	Context switch	Join	Semaphore sync.
Unbound thread	20.5	9	6	19
Bound thread	170	11	8.5	55

Figure 5.2: Uniprocessor timings in microseconds for Solaris threads operations on a 167 MHz Ultra-SPARC running Solaris 2.5. Creation time is with a preallocated stack, and does not include any context switch. (Creation of a bound or unbound thread without a preallocated stack incurs an additional overhead $200\mu s$ for the smallest stack size of a page(8KB). This overhead increases to $260\mu s$ for a 1MB stack.) Join is the time to join with a thread that has already exited. Semaphore synchronization time is the time for two threads to synchronize using a semaphore, and includes the time for one context switch.

Since Solaris Pthreads are created, destroyed and synchronized within a user-level library without kernel intervention, these operations are cheaper than the corresponding operations on kernel threads. Figure 5.2 shows the overheads for some Pthread operations on a 167 MHz UltraSPARC processor. Operations on bound threads involve operations on LWPs and require kernel intervention; they are hence more expensive than user-level operations on unbound threads. Note, however, that the user-level thread operations are still significantly more expensive than function calls; *e.g.,* the thread creation time of $20.5\mu s$ corresponds to over 3400 cycles on the 167 MHz UltraSPARC. The Pthreads implementation incurs this overhead for every thread expressed in the program, and does not attempt to automatically coarsen the parallelism by combining threads. Therefore, the basic thread overheads limit how fine-grained a task may be expressed using Pthreads without significantly affecting performance. It is left to the programmer to select the finest granularity for the threads such that the overheads remain insignificant, while maintaining portability, simplicity and load balance (see Section 5.5 for a discussion of thread granularity).

Although more expensive than function calls, the thread overheads are low enough to allow the creation of many more threads than the number of processors during the execution of a parallel program, so that the job of scheduling these threads and balancing the load across processors may be left to the threads implementation. Thus, this implementation of Pthreads is well-suited to express medium-grained threads, resulting in simple and efficient code, particularly for programs with dynamic parallelism. For example, Figure 5.3 shows the pseudocode for a block-based, divide-and-conquer algorithm for matrix multiplication using dynamic parallelism: each parallel, recursive call is executed by forking a new thread. (This code was introduced as a programming example in Chapter 1.) To ensure that the total overhead of thread operations is not significant, the parallel recursion on a 167 MHz UltraSPARC is terminated once the matrix size is reduced to 64×64 elements; beyond that point, an efficient serial algorithm is used to perform the multiplication[2]. The total time to multiply two 1024×1024 matrices with this algorithm on a single 167 MHz UltraSPARC processor, using a LIFO scheduling queue and assuming a preallocated stack for every thread created, is $17.6s$; of this, the thread overheads are no more than $0.2s$. The more complex but asymptotically faster Strassen's matrix multiply can also be implemented in a similar divide-and-conquer fashion with a few extra lines of code; coding it with static partitioning is significantly more difficult. Further, efficient, serial, machine-specific library routines can be easily plugged in to multiply the 64×64 submatrices at the base of the recursion tree. Temporary space is allocated at the start of each recursive call to store intermediate results. Although the allocation of this temporary space can be avoided, but this would significantly add to the complexity of the code or reduce the parallelism. The resulting dag for this matrix multiply program is shown in figure 5.4. The dag is similar to the one in Figure 1.3(b), except that threads here are forked in a tree instead of the flat loop, and memory allocations decrease down the tree. Dense matrix multiply is a fairly regular application and can be statically partitioned. Nonetheless, I am using it here as a simple example to represent a broad class of divide-and-conquer algorithms with similar dags but irregular, data-dependent structures. (Examples of such programs include sorts, data classifiers, computational geometry codes, etc.) The recursive matrix multiply algorithm discussed here generates large numbers of threads of varying granularities and memory requirements, and is therefore a good test for the Pthread scheduler.

Performance of matrix multiply using the native Pthreads implementation

I implemented the algorithm in Figure 5.3 on an 8-processor Sun Enterprise 5000 SMP running Solaris with 2 GB of main memory. Each processor is a 167 MHz UltraSPARC with a 512KB L2 cache. Memory access latency on the Enterprise is 300 ns (50 cycles), while obtaining a line from another processor's L2 cache requires 480 ns (80 cycles). Figure 5.5 (a) shows the speedup of the program with respect to the serial C version written with function calls instead of forks. The speedup was unexpectedly poor for a compute-intensive parallel program like matrix multiply. Further, as shown in Figure 5.5 (b), the maximum memory allocated by the program during its execution (*e.g.*, 115 MB on 8 processors) significantly exceeded the memory allocated by the corresponding serial program (25 MB).

To detect the cause for the poor performance of the program, I used a profiling tool (Sun Workshop version 4.0) to obtain a breakdown of the execution time, as shown in Figure 5.6. The processors spend a significant portion of the time in the kernel making system calls. The most time-consuming system calls were those involved in memory allocation. I also measured the maximum number of threads active during the execution of the program: the Pthreads implementation creates more than 4500 active threads during execution on a single processor. In contrast, a simple, serial, depth-first execution of the program (in which a child preempts its parent as soon as it is forked) on a single processor should result in just 10 threads being simultaneously active. Both these measures indicate that the native Pthreads implementation creates a large number of active threads, which

[2]The matrix multiply code was adapted from an example Cilk program available with the Cilk distribution [25].

```
begin Matrix_Mult(A, B, C, n)
    if (n ≤ Leaf_Size)
        serial_mult(A, B, C, n);
    else
        T := mem_alloc(n × n);
        initialize smaller matrices as quadrants of A, B, C, and T;
        fork Matrix_Mult(A11, B11, C11, n/2);
        fork Matrix_Mult(A11, B12, C12, n/2);
        fork Matrix_Mult(A21, B12, C22, n/2);
        fork Matrix_Mult(A21, B11, C21, n/2);
        fork Matrix_Mult(A12, B21, T11, n/2);
        fork Matrix_Mult(A12, B22, T12, n/2);
        fork Matrix_Mult(A22, B22, T22, n/2);
        fork Matrix_Mult(A22, B21, T21, n/2);
        join with all forked child threads;
        Matrix_Add(T, C);
        mem_free(T);
end Matrix_Mult
```

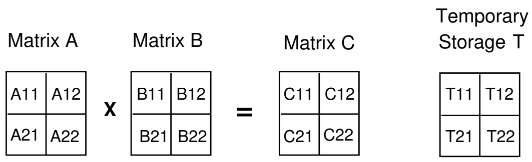


Figure 5.3: Pseudocode to multiply two $n \times n$ matrices A and B and storing the final result in C using a divide-and-conquer algorithm. The `Matrix_Add` function is implemented similarly using a parallel divide-and-conquer algorithm. The constant Leaf_Size to check for the base condition of the recursion is set to 64 on a 167 MHz UltraSPARC.

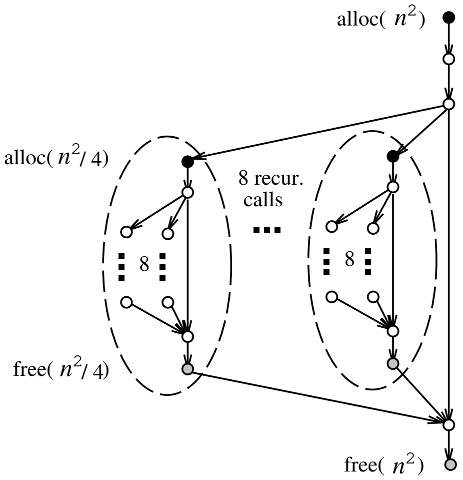
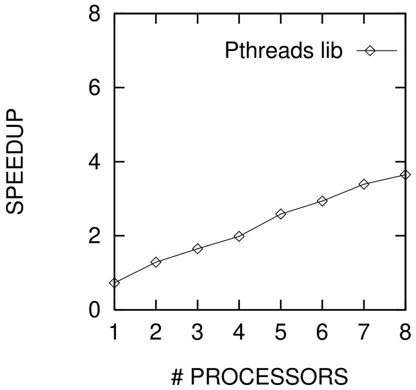
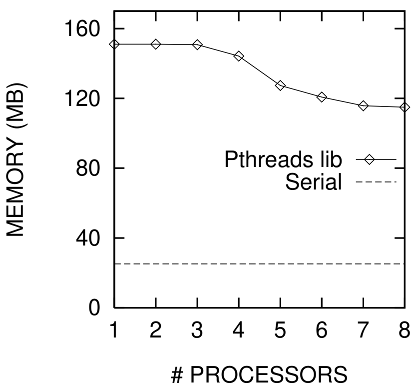


Figure 5.4: Program dag for recursive matrix multiply; the oval outlines demarcate the recursive calls at the top level of the recursion. Shaded nodes denote allocation (black) and deallocation (grey) of temporary space. The additional threads that perform matrix addition in parallel (in matrix_add) are not shown here for brevity.



(a)



(b)

Figure 5.5: Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using the native Pthreads implementation: (a) speedup with respect to a serial C version; (b) high water mark of total heap memory allocation during the execution of the program. "Serial" is the space requirement of the serial program, and equals the size of the input and output (A, B, and C) matrices.

all contend for allocation of stack and heap space, as well as for scheduler locks, resulting in poor speedup and high memory allocation. Even if a parallel program exhibits good speedups for a given problem size, it is important to minimize its memory requirement; otherwise, as the problem size increases, the performance soon begins to suffer due to excessive TLB and page misses.

The Solaris Pthreads implementation creates a very large number of active threads because it uses a FIFO queue. Further, when a parent thread forks a child thread, the child thread is added to the queue rather than being scheduled immediately. Consequently, the program dag is executed in a breadth-first manner, resulting in almost all the 4096 threads at the lowest level of the recursion tree being simultaneously active.

To improve the time and space performance of multithreaded applications a scheduling technique that creates fewer active threads, as well as limits their memory allocation, is necessary. I next describe the modifications I made to the existing Pthreads implementation to make it space-efficient.

5.3 Improvements to the Pthreads Implementation

This section lists the modifications I made to the user-level Pthreads implementation on Solaris 2.5 to improve the performance of the matrix multiply algorithm described in Section 5.2. The effect of each modification on the program's space and time performance is shown in Figure 5.7. All the speedups in Figure 5.7(a) are with respect to the serial C version of matrix multiply. The curves marked "Original" in the figure are for the original Pthreads implementation (with the deadlock-avoidance feature of automatic creation of new LWPs [150] disabled to get consistent timings[3]).

All threads were created at the same priority level, and the term "scheduling queue" used

[3]Disabling this feature stabilizes and marginally improves the performance of the application using the original Pthreads scheduler.

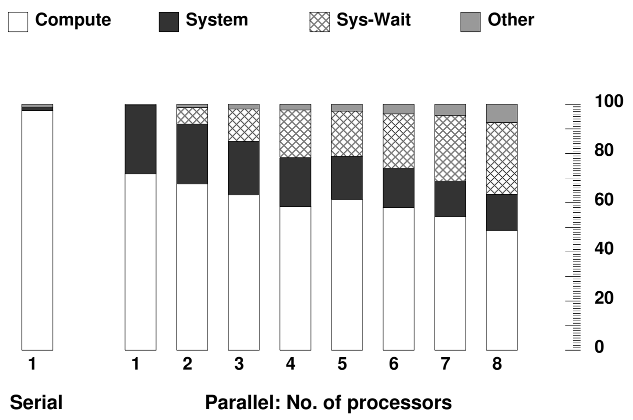


Figure 5.6: A breakdown of execution times on up to 8 processors for matrix multiply. "Compute" is the percentage of time doing useful computation, "system" is the percentage of time spent in system calls, and "sys-wait" is the percentage of time spent waiting in the kernel. "Other" includes idle time, the time spent waiting on user-level locks, and the time spent faulting in text and data pages.

below refers to the set of all threads at that priority level. In the original Pthreads implementation, this scheduling queue is implemented as a FIFO queue. To prevent multiple processors from simultaneously accessing the queue, it is protected by a common scheduler lock. I retained the use of this lock in all my modifications.

1. **LIFO scheduler.** I first modified the scheduling queue to be last-in-first-out (LIFO) instead of FIFO. A FIFO queue executes the threads in a breadth-first order, while a LIFO queue results in execution that is closer to a depth-first order. As expected, this reduces the memory requirement. However, the memory requirement still increases with the number of processors. The speedup improves significantly (see curve labeled as "LIFO" in Figure 5.7).

2. ***AsyncDF*: a space-efficient scheduler.** Next, I implemented a variation of algorithm *AsyncDF* (as shown in Figure 3.2). In this variation, instead of using the queues Q_{in} and Q_{out}, all processors directly access the scheduling queue ($\mathcal{R}$) using the existing scheduler lock. Therefore, no special scheduler threads are required. The main difference between this variation of *AsyncDF* and the LIFO scheduler is that threads in the scheduling queue at each priority level are always maintained in their serial, depth-first execution order. As described in Chapter 3, maintaining this order at runtime is simple and inexpensive. The final scheduling algorithm can be described as follows.

   - There is an entry in the scheduling queue for every thread that has been created but that has not yet exited. Thus threads represented in the queue may be either ready, blocked, or executing. These entries serve as placeholders for blocked or executing threads.
   - When a thread is preempted, it is returned to the scheduling queue in the *same* position (relative to the other threads) that it was in when it was last selected for execution. This position was marked by the thread's entry in the queue.

- When a parent thread forks a child thread, the parent is preempted immediately and the processor starts executing the child thread. A newly forked thread is placed to the immediate left of its parent in the scheduling queue.
- Every time a thread is scheduled, it is allocated a memory quota (implemented as a simple integer counter) initialized to a constant K bytes (the memory threshold). When it allocates m bytes of memory, the counter is decremented by m units. When a thread reaches an action that needs to allocate more memory than the current value of its counter, the thread is preempted. If a thread contains an action that allocates $m > K$ bytes, δ ***dummy*** threads (threads that perform a no-op and exit) are inserted in parallel[4] by the new scheduler before the allocation, where δ is proportional to m/K.

I modified the `malloc` and `free` library functions to keep track of a thread's memory quota, and fork dummy threads when necessary. The curve labelled "ADF" in Figure 5.7(a) shows that the speedup improves with the space-efficient scheduler. Further, the memory requirement (see Figure 5.7(b)) is significantly lower, and no longer increases rapidly with the number of processors. The memory threshold K was set to 50,000 bytes in these experiments[5].

3. **Reduced default stack size.** The Pthread library on Solaris specifies a stack size of 1MB for threads created with default attributes, and caches stacks of this default size for reuse. However, for applications that dynamically create and destroy a large number of threads, where each thread requires a more modest stack size, the default size of 1MB is too large. Therefore, to avoid requiring the programmer to supply and cache thread stacks in each application, I changed the default stack size to be a page (8KB). This reduces the time spent allocating the stacks. The improved performance curves are marked as "LIFO + small stk" with the LIFO scheduler, and "ADF + small stk" with the new, space-efficient scheduler. The final version ("ADF + small stk") yields good absolute performance for matrix multiplication; it runs within 2% of the hand-optimized, machine-specific BLAS3 routine for matrix multiplication on Solaris 2.5. The breakdown of running times in this final version is shown in Figure 5.8.

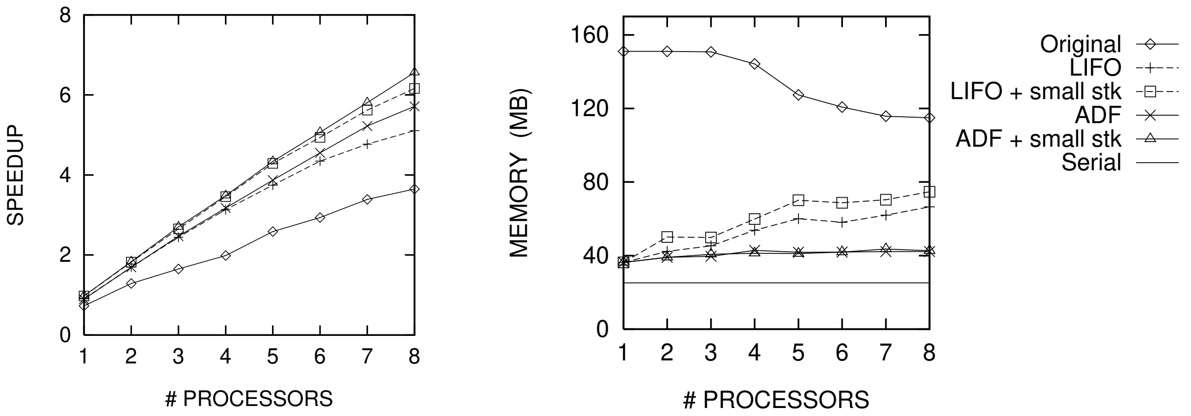
   The improvements indicate that allowing the user to determine the default thread stack size may be useful. However, predicting the required stack size can be difficult for some applications. In such cases, instead of conservatively allocating an extremely large stack, a technique such as stacklets [70] or whole program optimization [73] could be used to dynamically and efficiently extend stacks.

5.4 Other Parallel Benchmarks

In this section, I briefly describe experiments with 6 additional Pthreads-based parallel programs. The majority of them were originally written to use one thread per processor. I rewrote the pro-

[4] Since the Pthreads interface allows only a binary fork, these δ threads are forked as a binary tree instead of a δ-way fork.

[5] Compared to the threads described in Chapter 4, Pthreads are more expensive to create and destroy. Therefore, a fewer number of Pthreads (and hence a larger K) are sufficient to delay large allocations.



(a) (b)

Figure 5.7 : Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using variations of the native Pthreads implementation: (a) speedup with respect to a serial C version; (b) high water mark of heap memory allocation during the execution of the program. The results were averaged over 3 consecutive runs of the program. "Original" is with the original Pthreads implementation, "LIFO" uses a LIFO scheduler, "LIFO + small stk" stands for the LIFO scheduler with a reduced default stack size, "ADF" uses a variation of algorithm *AsyncDF*, and "ADF + small stk" uses the variation of *AsyncDF* with a reduced default stack size.

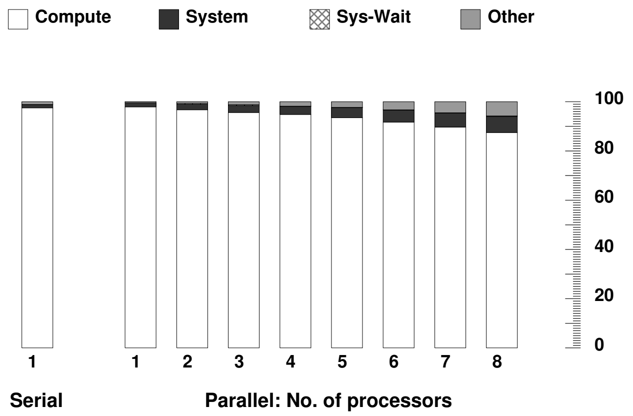


Figure 5.8: A breakdown of execution times on up to 8 processors for matrix multiply using the space-efficient scheduler based on algorithm *AsyncDF*. The Pthread library's default stack size was set at 8KB. The different components are the same as the components in Figure 5.6.

Benchmark	Problem Size	Coarse gr.	Fine gr. + orig. lib		Fine gr. + new lib	
		Speedup	Speedup	Threads	Speedup	Threads
Matrix Mult.	1024×1024	6.63	3.65	1977	6.56	59
Barnes Hut	$N = 100K$, Plmr. model	7.53	5.76	860	7.80	34
FMM	$N = 10K$, 5 terms	—	4.90	4348	7.45	24
Decision Tree	133,999 instances	—	5.23	94	5.25	70
FFTW	$N = 2^{22}$	6.27	5.84	224	5.94	14
Sparse Matrix	30K nodes, 151K edges	6.14	4.41	55	5.96	32
Vol. Rend.	256^3 volume, 375^2 image	6.79	5.73	131	6.69	25

Figure 5.9: Speedups on 8 processors over the corresponding serial C programs for the 7 parallel benchmarks. Three versions of each benchmark are listed here: the original coarse-grained version (BLAS3 for Matrix Multiply, and none for FMM or Decision Tree), the fine-grained version that uses a large number of threads with the original Solaris Pthreads implementation, and the fine-grained version with the modified Pthreads implementation (that uses the space-efficient scheduler and an 8KB default stack size). "Threads" is the maximum number of active threads during the 8-processor execution.

grams to use a large number of Pthreads, and compared the performance of the original, coarse-grained program with the rewritten, fine-grained version. The performance of the fine-grained version was measured using both the original Pthreads implementation and the implementation with the space-efficient scheduler based on algorithm *AsyncDF* (using a reduced 8KB default stack size). I refer to the latter setting as the ***new*** Pthreads scheduler (or as the new version of the Pthreads implementation). Since Pthreads are significantly more expensive than function calls, I coarsened some of the natural parallelism available in the program. This simply involved setting the chunking size for parallel loops or the termination condition for parallel recursive calls. The coarsening amortizes thread operation costs and also provides good locality within a thread, but still allows a large number of threads to be expressed. All threads were created requesting the smallest stack size of one page (8KB). The experiments were performed on the 8-processor Enterprise 5000 described in Section 5.2. The memory threshold K was set to 50,000 bytes. All programs were compiled using Sun's Workshop compiler (`cc`) 4.2, with the optimization flags `-fast -xarch=v8plusa -xchip=ultra -xtarget=native -xO4`.

5.4.1 The parallel benchmarks

Each benchmark is described below with its experimental results; Figure 5.9 summarizes the results for all the benchmarks.

1. Barnes Hut

This program simulates the interactions in a system of N bodies over several timesteps using the Barnes-Hut algorithm[11]. Each timestep has three phases: an octree is first built from the set of bodies, the force on each body is then calculated by traversing this octree, and finally, the position and velocity of each body is updated. I used the "Barnes" application code from the SPLASH-2 benchmark suite [163] in our experiments.

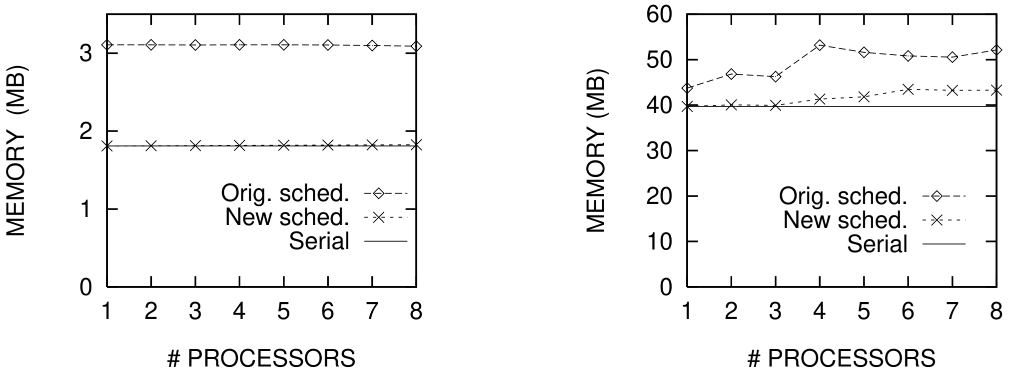
In the SPLASH-2 Barnes code, one Pthread is created for each processor at the beginning of the execution; the threads (processors) synchronize using a barrier after each phase within a timestep. Once the tree is constructed, the bodies are partitioned among the processors. Each processor traverses the octree to calculate the forces on the bodies in its partition, and then updates the positions and velocities of those bodies. It also uses its partition of bodies to construct the octree in the next timestep. Since the distribution of bodies in space may be highly non-uniform, the work involved for the bodies may vary to a large extent, and a uniform partition of bodies across processors leads to load imbalance. The Barnes code therefore uses a *costzones* partitioning scheme to partition the bodies among processors [146]. This scheme tries to assign to each processor a set of bodies that involve roughly the same amount of work, and are located close to each other in the tree to get better locality.

I modified the Barnes code so that, instead of partitioning the work across the processors, a new Pthread is created to execute each small, constant-sized unit of work. For example, in the force calculation phase, starting from the root of the octree, a new Pthread was recursively forked to compute the force on the set of particles in each subtree. The recursion was terminated when the subtree had (on average) under 8 leaves. Since each leaf holds multiple bodies, this granularity is sufficient to amortize the cost of thread overheads and to provide good locality within a thread. Thus, we do not need any partitioning scheme in my code, since the large number of threads in each phase are automatically load balanced by the Pthreads implementation. Further, no per-processor data structures were required in my code, and the final version was significantly simpler than the original code.

The simulation was run on a system of 100,000 bodies generated under the Plummer model [1] for four timesteps (as with the default Splash-2 settings, the first two timesteps were not timed). Figure 5.9 shows that the simpler, fine-grained approach achieves the same high performance as the original code. However, the Pthread scheduler needs to be carefully implemented to achieve this performance. When the thread granularity is coarsened and therefore the number of threads is reduced, the performance of the original scheduler also improves significantly. However, as the problem size scales, unless the number of threads increases, the scheduler cannot balance the load effectively. Besides forks and joins, this application uses Pthread mutexes in the tree building phase to synchronize modifications to the partially built octree.

2. Fast Multipole Method

This application executes the Fast Multipole Method or FMM [72], another N-Body algorithm. The FMM in three dimensions, although more complex, has been shown to perform fewer computations than the Barnes-Hut algorithm for simulations requiring high accuracy, such as electrostatic systems [20]. The main work in FMM involves the computation of local and multipole expansion series that describe the potential field within and outside a cell, respectively. I first wrote the serial C version for the uniform FMM algorithm, and then parallelized it using Pthreads. The parallel version is written to use a large number of threads, and I do not compare it here to any preexisting version written with one thread per processor. The program was executed on 10,000 uniformly distributed particles by constructing a tree with 4 levels and using 5 terms in the multipole and local expansions.



(a) FMM (b) Decision Tree

Figure 5.10 : Memory requirements for the FMM and Decision Tree benchmarks. "Orig. lib" uses the native Pthreads implementation, while "New lib" uses the implementation modified to use the new scheduler based on *AsyncDF*. "Serial" is the space requirement of the serial program.

Each phase of the force calculation and its parallelization is described below.

1. Multipole expansions for leaf cells are calculated from the positions and masses of their bodies; a separate thread is created for each leaf cell.

2. Multipole expansions of interior cells are calculated from their children in a bottom-up phase; a separate thread is created for each interior (parent) cell.

3. In a top-down phase, the local expansion for each cell is calculated from its parent cell and from its well-separated neighbors; since each cell can have a large number of neighbors (up to 875), we created a separate thread to compute interactions with up to constant number (25) of a cell's neighbors. Threads are forked as a binary tree.

4. The forces on bodies are calculated from the local expansions of their leafs and from direct interactions with neighboring bodies; a separate thread is created for each leaf cell.

Since this algorithm involves dynamic memory allocation (in phase 3), I measured its space requirement with the original and new versions of the Pthreads library implementation (see Figure 5.10(a)). As with matrix multiply, the new scheduling technique (based on *AsyncDF*) results in lower space requirement. The speedups with respect to the serial C version are included in Figure 5.9.

3. Decision Tree Builder

Classification is an important data mining problem. I implemented a decision tree builder to classify instances with continuous attributes. The algorithm used is similar to ID3 [132], with C4.5-like additions to handle continuous attributes [133]. The algorithm builds the decision tree in a top-down, divide-and-conquer fashion, by choosing a split along the continuous-valued attributes based on the best gain ratio at each stage. The instances are sorted by each attribute to calculate the

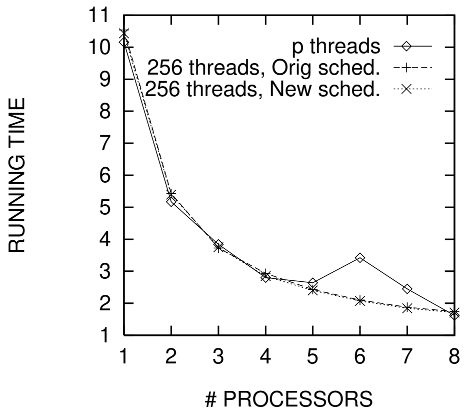


Figure 5.11: Running times for three versions of the multithreaded, one-dimensional DFT from the FFTW library on p processors: (1) using p threads, (2) using 256 threads with the original Pthreads scheduler, (3) using 256 threads with the modified Pthreads scheduler.

optimal split. The resulting divide-and-conquer computation graph is highly irregular and data dependent, where each stage of the recursion itself involves a parallel divide-and-conquer quicksort to split the instances. I used a speech recognition dataset [78] with 133,999 instances, each with 4 continuous attributes and a true/false classification as the input. A thread is forked for each recursive call in the tree builder, as well as for each recursive call in quicksort. In both cases, a switch to serial recursion is made once the number of instances is reduced to 2000. Since a coarse-grained implementation of this algorithm would be highly complex, requiring explicit load balancing, I did not implement it. The 8-processor speedups obtained with the original and new Pthreads scheduler are shown in Figure 5.9; both the schedulers result in good time performance; however, the new scheduler resulted in a lower space requirement (see Figure 5.10(b)).

4. Fast Fourier Transform

The FFTW ("Fastest Fourier Transform in the West") library [65] computes the one- and multidimensional complex discrete Fourier transform (DFT). The FFTW library is typically faster than all other publicly available DFT code, and is competitive or better than proprietary, highly optimized versions such as Sun's Performance Library code. FFTW implements the divide-and-conquer Cooley-Tukey algorithm [43]. The algorithm factors the size N of the transform into $N = N_1 \cdot N_2$, and recursively computes N_1 transforms of size N_2, followed by N_2 transforms of size N_1. The package includes a version of the code written with Pthreads, which I used in these experiments. The FFTW interface allows the programmer to specify the number of threads to be used in the DFT. The code forks a Pthread for each recursive transform, until the specified number of threads are created; after that it executes the recursion serially. The authors of the library recommend using one Pthread per processor for optimal performance.

A one-dimensional DFT of size $N = 2^{22}$ was executed in these experiments, using either p threads (where p = no. of processors) as the coarse grained version, or 256 threads as the fine grained version. Figure 5.11 shows the speedups over the serial version of the code for one to eight processors. When p is a power of two, the problem size (which is also a power of two) can be uniformly partitioned among the processors using p threads, and being a regular computation, it does not suffer from load imbalance. Therefore, for $p = 2, 4, 8$, the version with p threads runs marginally faster. However, for all other p, the version with a larger number of threads can be better load balanced by the Pthreads implementation, and therefore performs better. This example indicates that without any changes to the code, the performance becomes less sensitive to the number of processors when a large number of lightweight threads are used. The performance of this application was comparable with both the original Pthreads scheduler and the modified scheduler (see Figure 5.9).

5. Sparse Matrix Vector Product

This benchmark involves of 20 iterations of the product $w = M \cdot v$, where M is a sparse, non-symmetric matrix and v and w are dense vectors. The code is a modified version of the Spark98 kernels [117] which are written for symmetric matrices. The sparse matrix in these experiments is generated from a finite element mesh used to simulate the motion of the ground after an earthquake in the San Fernando valley [9, 10]; it has 30,169 rows and 151,239 non-zeroes. In the coarse-grained version, one thread is created for each processor at the beginning of the simulation, and the threads execute a barrier at the end of each iteration. Each processor (thread) is assigned a disjoint and contiguous set of rows of M, such that each row has roughly equal number of nonzeroes. Keeping the sets of rows disjoint allows the results to be written to the w vector without locking.

In the fine-grained version, 64 threads are created and destroyed in each iteration. The rows are partitioned equally rather than by number of nonzeroes, and the load is automatically balanced by the threads scheduler (see Figure 5.9).

6. Volume Rendering

This application from the Splash-2 benchmark suite uses a ray casting algorithm to render a 3D volume [145, 163] . The volume is represented by a cube of volume elements, and an octree data structure is used to traverse the volume quickly. The program renders a sequence of frames from changing viewpoints. For each frame, a ray is cast from the viewing position through each pixel; rays are not reflected, but may be terminated early. Parallelism is exploited across these pixels in the image plane. My experiments do not include times for the preprocessing stage which reads in the image data and builds the octree.

In the Splash-2 code, the image plane is partitioned into equal sized rectangular blocks, one for each processor. However, due to the nonuniformity of the volume data, an equal partitioning may not be load balanced. Therefore, every block is further split into tiles, which are 4×4 pixels in size. A task queue is explicitly maintained for each processor, and is initialized to contain all the tiles in the processor's block. When a processor runs out of work, it steals a tile from another processor's task queue. The program was run on a $256 \times 256 \times 256$ volume data set of a Computed Tomography head and the resulting image plane was 375×375 pixels.

In the fine-grained version, a separate Pthread was created to handle each set of 64 tiles (out of a total of 8836 tiles). Since rays cast through consecutive pixels are likely to access much of the same volume data, grouping a small set of tiles together is likely to provided better locality. However, since the number of threads created is much larger than the number of processors, the computation is load balanced across the processors by the Pthreads scheduler, and does not require

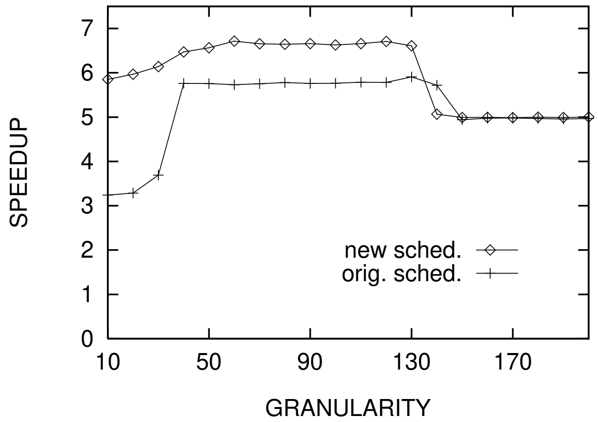


Figure 5.12: Variation of speedup with thread granularity (defined as the maximum number of 4×4 pixel tiles processed by each thread) for the volume rendering benchmark. "Orig. sched." is the speedup using the original FIFO scheduling queue, while "New sched." is the speedup using the space-efficient scheduler based on *AsyncDF*.

the explicit task queues used in the original, coarse-grained version. Figure 5.9 shows that the performance of the simpler, rewritten code (using the modified Pthread scheduler) is competitive with the performance of the original code.

5.5 Selecting the optimal thread granularity

Fine-grained threads allow for good load balancing, but may incur high thread overheads and poor locality. In the experiments described so far, the granularity of the Pthreads was adjusted to amortize the cost of basic thread operations (such as creation, deletion, and synchronization). However, since algorithm *AsyncDF* may schedule threads accessing common data on different processors, the granularity needed to be increased further for some applications to get good locality within each thread. For example, in the volume rendering application, if we create a separate thread to handle each set of 8 4x4 tiles of the image, the serial execution of the multithreaded program is slowed down by at most 2.25% due to basic thread overheads, compared to the serial C program. However, the same program on 8 processors slows down nearly 20%, probably due to poorer data locality. Tiles close together in the image are likely to access common data, and therefore the program scales well when the thread granularity is increased so that each thread handles 64 (instead of 8) tiles. This effect of thread granularity on performance is shown in Figure 5.12. The application slows down at large thread granularities (> 130 tiles per thread) due to load imbalance, while the slowdown at finer thread granularities (< 50 tiles per thread) is due to high scheduling overheads and poor locality. Ideally, we would like the performance to be maintained as the granularity is reduced.

Since basic Pthread operations cannot be avoided, the user must coarsen thread granularities to amortize their costs. However, ideally, we would not require the user to further coarsen threads for locality. Instead, the scheduling algorithm should schedule threads that are close in the computation graph on the same processor, so that good locality may be achieved. Chapter 7 presents a space-efficient scheduling algorithm, namely, algorithm *DFDeques*, that was designed for this purpose.

5.6 Summary

In this chapter, I have described a space-efficient implementation of Posix threads or Pthreads. The new scheduler added to the native Pthreads implementation on Solaris is based on algorithm *AsyncDF*. The shared scheduling queue within the threads package was essentially converted from a FIFO queue to a priority queue in which Pthreads are prioritized by their serial execution order.

The space and time performance of new scheduler was evaluated using seven Pthreads-based benchmarks: a dense matrix multiply, a sparse matrix multiply, a volume renderer, a decision tree builder, two N-body codes, and an FFT package. The experimental results indicate that the space-efficient scheduler results in better space and time performance compared to the original FIFO scheduler. In particular, it allows simpler, fine-grained code for the benchmarks to perform competitively with their hand-partitioned, coarse-grained counterparts.

One drawback of expressing a large number of lightweight, fine-grained Pthreads is having to pick the appropriate thread granularity. Since Pthreads are two orders of magnitude more expensive than function calls, the user always has to ensure that the work performed by the threads amortizes the cost of basic thread operations. However, with algorithm *AsyncDF*, further coarsening of threads may be required to ensure low scheduling contention and good data locality within each thread. The next chapter presents algorithm *DFDeques* with the goal of addressing this problem.

Chapter 6

Automatic Coarsening of Scheduling Granularity

Thread packages like Pthreads do not support very fine-grained thread operations due to their rich functionality and library approach. For example, because Pthreads cannot take advantage of lazy thread creation as in specialized multithreaded languages [70, 66, 102], every Pthread expressed by the programmer incurs the cost of a thread creation and deletion. Therefore, even for serial executions, the programmer must ensure that average thread granularities are sufficiently large to amortize the costs of basic thread operations. However, we showed in Chapter 5 that for some applications, the *AsyncDF* scheduling algorithm does not result in good parallel performance even at such threads granularities. In particular, threads have to be further coarsened to allow good locality and low scheduling overheads during a parallel execution. Ideally, however, the user should not have to further coarsen threads beyond what is required to amortize the cost of basic thread operations. Instead, the scheduling algorithm should automatically schedule fine-grained threads that are close in the computation's dag on the same processor, so that good locality and low scheduling overheads are achieved.

In this chapter, I introduce algorithm *DFDeques*, an improvement on algorithm *AsyncDF*; it is aimed at providing better time performance for finer grained threads by increasing the scheduling granularity beyond the granularities of individual threads. Like algorithm *AsyncDF*, it guarantees a space bound of $S_1 + O(K \cdot p \cdot D)$ (in the expected case) for a nested-parallel program with serial space requirement S_1 and depth D executing on p processors. As before, K is the user-adjustable memory threshold used by the scheduler. For a simplistic cost model, I show that the expected running time is $O(W/p+D)$ on p processors. However, when the scheduler in *DFDeques* is parallelized, the costs of all scheduling operations can be accounted for with a more realistic model. Then the parallel computation can be executed using $S_1 + O(K \cdot D \cdot p \cdot \log p)$ space and $O(W/p + D \cdot \log p)$ time (including scheduling overheads); such a parallelized scheduler is described in Appendix D.

This chapter is organized as follows. Section 6.1 motivates the need for improving algorithm *AsyncDF* by providing a simple example; it also explains the main differences between algorithms *AsyncDF* and *DFDeques*. Section 6.2 then describes the programming model supported by algorithm *DFDeques* and the data structures it uses, followed by the pseudocode for the algorithm itself. I analyze the space and time requirements for a parallel computation executed using al-

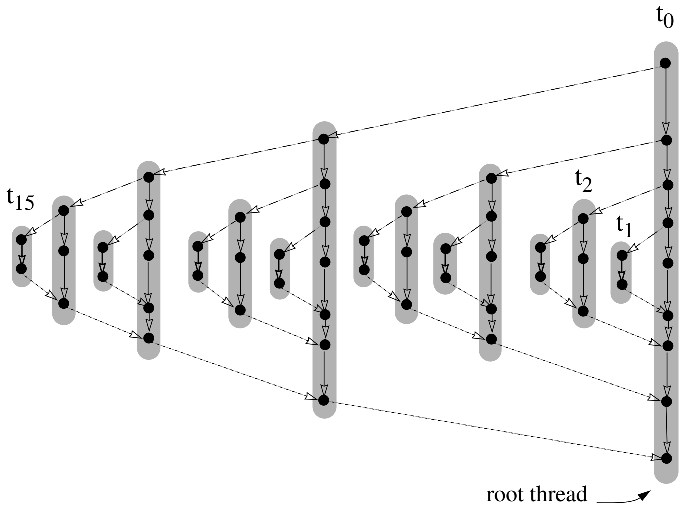


Figure 6.1: An example dag for a parallel computation. each of the 16 threads is shown as a shaded region. The threads are numbered from right-to-left; thread t_0 is the initial, root thread. In this example, thread t_i accesses i^{th} element of an array.

gorithm *DFDeques* in Section 6.3. Finally, Section 6.4 summarizes the results presented in this chapter.

6.1 Motivation and Basic Concepts

Consider the example in Figure 6.1. The dag represents a parallel loop that is forked as a binary tree of threads, with one thread per iteration. Let the i^{th} thread (going from right to left) execute the i^{th} iteration of the parallel loop; let us assume that it accesses the i^{th} element of an array. Therefore, if multiple, consecutive elements of an array fit into one cache line, then scheduling a block of consecutive threads on one processor results in good locality; Figure 6.2(a) shows one such schedule. Further, contention while accessing scheduling data structures could be reduced by storing an entire block of threads in an individual processor's scheduling queue. Then for a majority of the time, a processor has to access only its own queue. Work stealing scheduling techniques in which coarser-grained threads are typically stolen by an idle processor [77, 25, 101, 96, 159] provide such benefits, and are therefore likely to result in schedules such as the one shown in Figure 6.2(a).

Unlike work stealing, algorithm *AsyncDF* uses a single, flat, global scheduling queue. Further, high-priority threads in *AsyncDF* are often deeper in the dag and therefore more fine grained. Consequently, consecutive threads in the example may be scheduled by *AsyncDF* on different processors, as shown in Figure 6.2(b). This can result in poor locality and high scheduling overheads when the threads are fairly fine grained. Note, however, that if each thread were somewhat coarsened, so that it represents a block of iterations of the parallel loop and therefore access several consecutive memory locations, the time performance of algorithm *AsyncDF* would, in practice, be equivalent to that of a work stealing scheduler. Further, as indicated by results in Chapters 4 and 5, it would also typically result in lower memory requirements compared to previous sched-

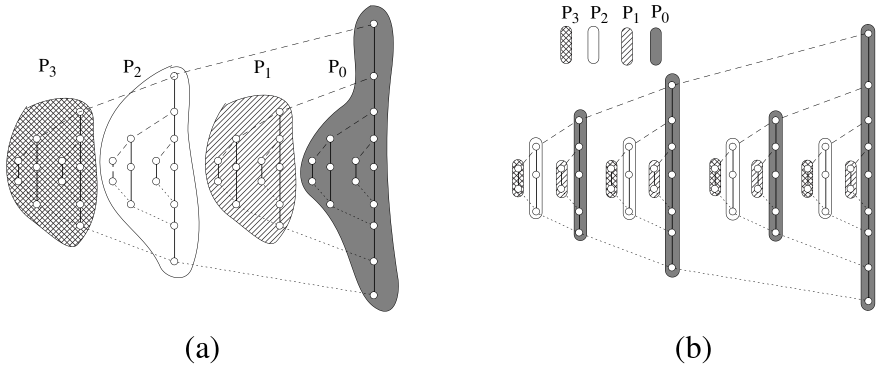


Figure 6.2: (a) and (b): Two different mappings of threads of the dag in Figure 6.1 onto processors $P_0, \ldots, P_3$. Scheduling consecutive threads on the same processor, as in (a), can provide better cache locality in this example.

ulers. Algorithm *DFDeques* was therefore designed to improve upon *AsyncDF* by automatically combining fine-grained threads close together in the dag into a single scheduling unit to provide a higher scheduling granularity. This typically results in better locality and lower scheduling contention that *AsyncDF*, particularly when threads are more fine grained. Algorithm *DFDeques* borrows ideas from work stealing to obtain this improvement in time performance, while maintaining the same asymptotic space bound as algorithm *AsyncDF*. However, as shown in Chapter 7, in practice *DFDeques* results in a marginally higher memory requirement compared to *AsyncDF*.

The basic differences between algorithms *AsyncDF* and *DFDeques* are summarized below.

- Threads in *AsyncDF* were organized in a flat priority queue ($\mathcal{R}$). In contrast, *DFDeques* uses a prioritized queue $\mathcal{R}'$, each element of which is a deque (doubly-ended queue). A deque holds multiple threads with contiguous (relative) priorities. At any time, a processor owns a separate deque (unless it is switching between deques) and treats it like a private LIFO stack to store ready threads.
- In *AsyncDF*, an idle processor always picks the next thread on Q_{out}; Q_{out} buffers the highest priority threads in $\mathcal{R}$. In contrast, an idle processor in *DFDeques* picks the top thread from its current deque. If it finds it's deque empty (or if it does not own a deque), it selects at random one of the high-priority deques, and steals a thread from the bottom of the deque. This thread is typically more coarse-grained than the thread with the absolute, highest priority.
- In algorithm *AsyncDF*, the memory threshold was used to limit the memory allocated by each individual thread between preemptions; a thread was preempted and its memory threshold was reset when it reached a fork instruction. In contrast, *DFDeques* allows the memory threshold to be used towards the memory allocation of multiple threads from the same deque. This allows multiple, fine-grained threads close in the dag to execute on the same processor (assuming they do not allocate much memory).
- In algorithm *AsyncDF*, all idle processors picked threads from the head of Q_{out}; for fine-grained threads, accessing Q_{out} can therefore become a bottleneck. *DFDeques* uses randomization to reduce contention between idle processors when they steal. I therefore provide high probability (and expected case) bounds for the space and time requirements of parallel

computations executed using algorithm *DFDeques*.

6.2 Algorithm *DFDeques*

This section begins by describing the programming model for the multithreaded computations that are executed by the algorithm *DFDeques*. I then list the data structures used in the scheduling algorithm, followed by the description of the scheduling algorithm.

6.2.1 Programming model

Algorithm *DFDeques* executes nested parallel computations with binary forks and joins (unlike the multi-way forks and joins supported by *AsyncDF*). For example, the dag in Figure 6.1 represents a nested parallel computation with binary forks and joins. As with *AsyncDF*, although I describe and analyze algorithm *DFDeques* for nested parallel computations, in practice it can be easily extended to programs with more general styles of parallelism. For example, the Pthreads scheduler described in Chapter 7 (Section 7.1) efficiently supports computations with arbitrary synchronizations, such as mutexes and condition variables.

6.2.2 Scheduling data structures

As with algorithm *AsyncDF*, threads are prioritized according to their serial execution order (*i.e.*, according to the 1DF-numbering of their current nodes). The ready threads are stored in doubly-ended queues or deques [45]. 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 total number of deques in the system may exceed the number of processors. Since the programming model allows only binary threads, threads need not be forked lazily to conserve space. Further, unlike algorithm *AsyncDF*, placeholders for currently executing threads are not required in algorithm *DFDeques*. Therefore, the deques of threads only contain ready threads, and not seed threads or placeholders.

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} deque from the left end of $\mathcal{R}'$. As we will prove in the next Section, algorithm *DFDeques* maintains threads in $\mathcal{R}'$ in decreasing order of priorities from left-to-right across deques, and from top to bottom within each deque (see Figure 6.3).

6.2.3 The scheduling algorithm

The processors execute the code in Figure 6.4 for algorithm *DFDeques* (K), where K is the memory threshold. 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 quota K is equivalent to the per-thread memory quota in algorithm *AsyncDF*. However, as noted in Section 6.1, the memory quota is assigned to a processor to execute multiple threads from a single

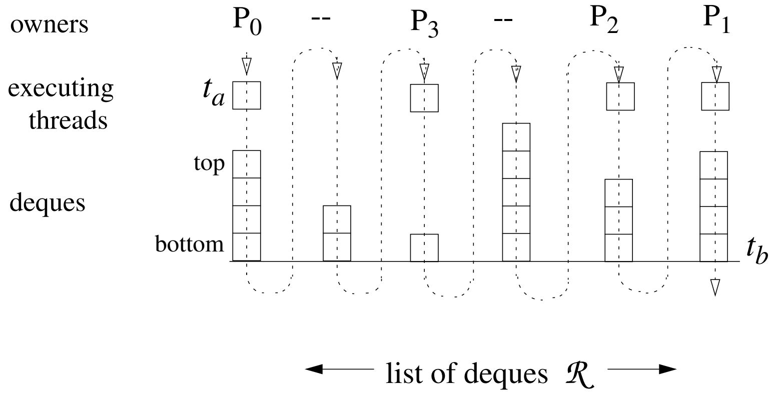


Figure 6.3 : The list $\mathcal{R}'$ of deques maintained in the system by algorithm *DFDeques*. Each deque may have one (or no) owner processor. The dotted line traces the decreasing order of priorities of the threads in the system; thus t_a in this figure has the highest priority, while t_b has the lowest priority.

deque (rather than a single thread, as in *AsyncDF*). A thread executes without preemption on a processor until one of four events takes place: (a) it forks a child thread, (b) it suspends waiting for a child to terminate, (c) the processor runs out of its memory quota, or (d) the thread terminates. When an idle processor finds its deque empty, it deletes the deque. When a processor runs out of its memory quota, or when it becomes idle and finds its deque empty, it gives up ownership of its deque and uses the `steal()` procedure to obtain a new deque. Every invocation of `steal()` resets the processor's memory quota to K bytes. Each iteration of the `while` loop in the `steal()` procedure a referred to as a ***steal attempt***.

The scheduling algorithm starts with a single deque in the system, containing the initial (root) thread. 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 $\mathcal{R}'$; this thread need not be the highest priority thread in the system. If $\mathcal{R}'$ has only $n < p$ deques, any steal attempts for $m \in [n+1, p]$ fail (that is, `pop_from_bot` returns NULL). A steal attempt may also fail if two or more processors target the same deque (as explained in Section 6.3.1), or if the deque is empty. 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 chosen deque, and starts executing the stolen thread. Otherwise, it repeats the steal attempt. 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 $m > K$, then $\lfloor m/K \rfloor$ dummy threads must be inserted before the allocation, similar to algorithm *AsyncDF*. Since the programming model supports only binary forks, these dummy threads are forked in a binary tree. We do not show this extension in Figure 6.4 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 addition of dummy threads effectively delays large allocations of space, so that higher priority threads may be scheduled instead.

Work stealing as a special case of algorithm *DFDeques*. For a nested-parallel computation,

```
while (∃ threads)
   if (currS = NULL) currS := steal();
   if (currT = NULL) currT := pop_from_top(currS);
   execute currT until it forks, suspends, terminates,
      or mem quota exhausted:
      case (fork):
         push_to_top(currT, currS);
         currT := newly forked child thread;
      case (suspend):
         currT := NULL;
      case (mem quota exhausted):
         push_to_top(currT, currS);
         currT := NULL;
         currS := NULL;    /* give up stack; */
      case (terminate):
         if currT wakes up suspended parent T'
            currT := T';
         else currT := NULL;
   if ((is_empty(currS)) and (currT= NULL))
      currS := NULL;
endwhile
```

```
procedure steal():
set memory quota to K;
while (TRUE )
   m := random number in [1 . . . p];
   S := m^th deque in R';
   T := pop_from_bot(S);
   if (T ≠ NULL)
      create new deque S' containing T
         and become its owner;
      place S' to immediate right of S in R';
      return S';
```

Figure 6.4 : Pseudocode for the *DFDeques* (K) scheduling algorithm executed by each of the p processors. K is the memory threshold, designed to be a user-adjustable command-line parameter. *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.

consider the case when we set $K = \infty$ on p processors. Then, algorithm *DFDeques* (∞) produces a schedule identical to the one produced by the provably efficient work stealing scheduler used in Cilk [25]. Since processors never give up a deque due to exhaustion of their memory quota, there are never more than p deques in the system. Maintaining the relative order of deques in $\mathcal{R}'$ becomes superfluous in this case, since now steal targets are always chosen at random from among all the deques in the system.

6.3 Analysis of Algorithm *DFDeques*

This section analyzes the time and space bounds for a parallel computation executed using algorithm *DFDeques* (K), where K is the user-assigned memory threshold. I first state the cost model assumed in the analysis, followed by definitions of some terms used in this section. I then prove the time and space bounds for a computation using algorithm *DFDeques* (K). To simplify the analysis, the cost model defined in this section assumes that deques can be inserted and deleted from $\mathcal{R}'$ at no additional cost. Appendix D explains how these operations can be performed lazily, and

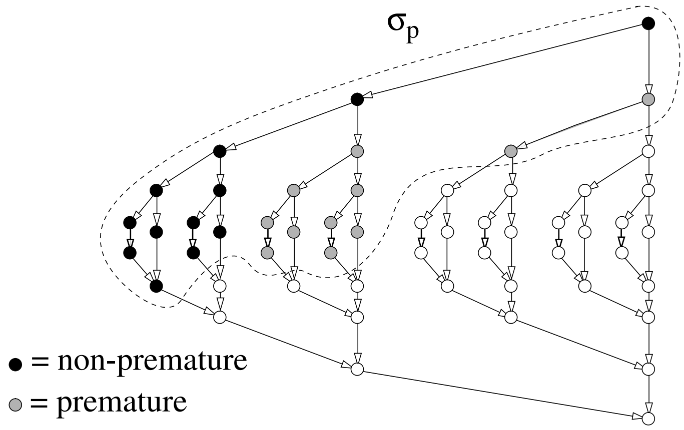


Figure 6.5: An example snapshot of a parallel schedule for the dag from Figure 6.1. The shaded nodes (the set of nodes in σ_p) have been executed, while the blank (white) nodes have not. Of the nodes in σ_p, the black nodes form the corresponding parallel prefix σ_1, while the remaining grey nodes are premature.

analyzes the running time after including their costs.

6.3.1 Cost model

As with the analysis of algorithm *AsyncDF*, we assume that the timesteps (clock cycles) are synchronized across all the processors. Recall that each action (or node in the dag) requires one timestep to be executed. We assume that an allocation of M bytes of memory (for any $M > 0$) has a depth of $\Theta(\log M)$. This is a reasonable assumption in systems with binary forks that zero out the memory as soon as it is allocated; this zeroing can be performed in parallel by forking a tree of height $\Theta(\log M)$.

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. In practice, the insertions of new deques and deletions of empty deques from $\mathcal{R}'$ can be either serialized (see Section 7.1), or performed lazily in parallel (see Appendix D). If a processor tries to remove the last thread from its deque, and another processor attempts a steal from the same deque, we assume that any one of them succeeds in removing the thread.

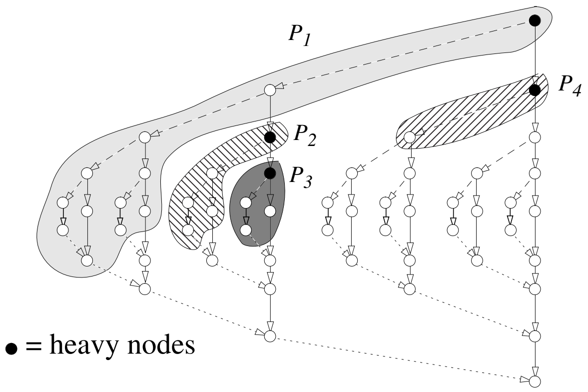


Figure 6.6 : A possible partitioning of nodes into batches for the parallel prefix σ_p from Figure 6.5. Each batch, shown with as a shaded region, is executed on one of the processors $P_0, \ldots, P_4$ in depth-first order without interruption. The heavy node in each batch is shown shaded black.

6.3.2 Definitions

We use the same definitions of prefixes and (non-)premature nodes that were defined in Chapter 3 (Section 3.3.2). Figure 6.5 shows the premature and non-premature nodes in an example dag for some arbitrary parallel prefix σ_p. However, we define a batch (of nodes) in a slightly different manner from Chapter 3. In the case of algorithm *DFDeques*, a ***batch*** now becomes the set of all nodes executed by a processor between consecutive steals. Since a processor uses its own deque to store and retrieve threads between steals, this new definition of a batch may include nodes from multiple threads instead of just one thread. As before, the first node executed from a batch is a ***heavy*** node, while all other nodes are ***light*** nodes. Thus, when a processor steals a thread from the bottom of a deque, the current node of the thread becomes a heavy node. As with the analysis of *AsyncDF*, heavy nodes are a property of the particular execution (p-schedule) rather than of the dag, since steals may take place at different times in different executions. Figure 6.6 shows a possible partitioning of nodes into batches for the parallel prefix from Figure 6.5.

We assume for now that all nodes allocate *at most* K memory; we will relax this assumption at the end of this subsection. Since the memory quota of a processor is reset to K every time it performs a steal, a processor may allocate at most K space for every heavy node it executes.

A ready thread being present in a deque is equivalent to its first unexecuted node (action) being in the deque, and we will use the two phrases interchangeably. Given a p-schedule s_p of a nested-parallel 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. The 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.

6.3.3 Space bound

We now show that using algorithm *DFDeques* (K) results in an expected space requirement of $S_1 + O(p \cdot D \cdot \min(K, S_1))$ for a nested-parallel computation with D depth and S_1 serial space requirement on p processors. Because, in practice, we use small, constant value of K, the space bound reduces to $S_1 + O(p \cdot D)$, as with algorithm *AsyncDF*.

The approach for proving the space bound of algorithm *DFDeques* is similar to that for algorithm *AsyncDF*. We first prove a lemma regarding the order of threads in $\mathcal{R}'$ maintained by algorithm *DFDeques*; this order is shown pictorially in Figure 6.3. We will then bound the *number of heavy premature nodes* that may have been executed in the parallel computation by the end of any timestep, that is, we bound the cardinality of the set $\sigma_p - \sigma_1$, for any prefix σ_p of a parallel schedule s_p executed by algorithm *DFDeques*.

Lemma 6.1 *Algorithm DFDeques maintains the following properties of the 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 exits, 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. ∎

We can now bound the number of heavy premature nodes in any snapshot (prefix) of the parallel schedule. Let s_p be the p-schedule of length T generated for G by algorithm *DFDeques* (K). For any $1 \leq \tau \leq T$, let σ_p be the prefix of s_p representing the execution after the first τ timesteps. Let $\sigma_1 \subseteq \sigma_p$ be the corresponding serial prefix of σ_p. Let v be the last non-premature node (*i.e.*, the last node from σ_1) to be executed during the first τ timesteps of s_p. If more than one such node exist, let v be any one of them. Let P be a set of nodes in the dag constructed as follows: P is initialized to $\{v\}$; for every node u in P, the last parent of u is added to P. Since the root is the only node at depth 1, it must be in P, and thus, P contains exactly all the nodes along a particular path from the root to v. Further, since v is non-premature, all the nodes in P are non-premature.

Let u_i be the node in 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 $I_{\delta+1} = \{t_\delta + 1, \ldots, \tau\}$. Since σ_p consists of all the nodes executed in the first τ timesteps, the intervals $I_1, \ldots, I_{\delta+1}$ cover the duration of execution of all nodes in σ_p.

We first prove a lemma regarding the nodes in a deque below any of the nodes in P.

Lemma 6.2 *For any $1 \leq i \leq \delta$, let u_i be the node in P at depth i. Then,*

1. *If anytime during the execution (in the first $t_i - 1$ timesteps) u_i is on some deque, then every node below it in its deque is the right child of some node in P.*

2. *When u_i is executed (at timestep t_i) on a processor, every node on the processor's deque must be the right child of some node in 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 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 6.7), 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

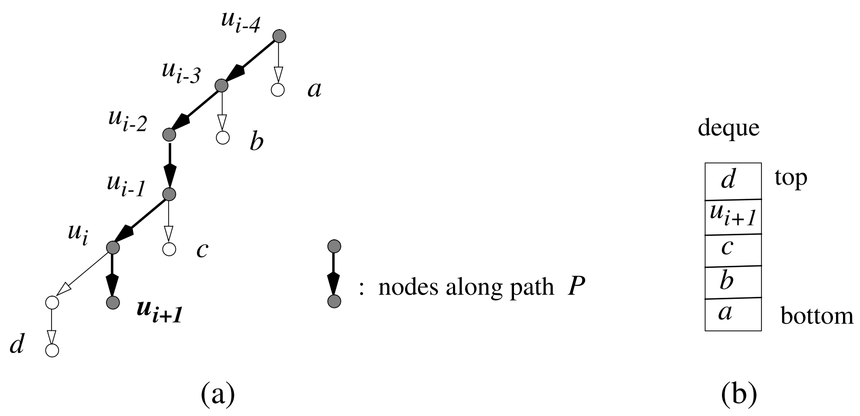


Figure 6.7 : (a) A portion of the dynamically unfolding dag during the execution. Node u_{i+1} along the path P is ready, and is currently present in some deque. The deque is shown in (b); all nodes below u_{i+1} on the deque must be right children of some nodes on P above u_{i+1}. In this example, node 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}.

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.

∎

Recall that heavy nodes are a property of the parallel schedule, while premature nodes are defined relative to a given prefix σ_p of the parallel schedule. We now prove lemmas related to the number of heavy premature nodes in σ_p.

Lemma 6.3 *Let σ_p be any parallel prefix of a p-schedule produced by algorithm DFDeques (K) for a computation with depth D. 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 interval I_i of σ_p, for $i = 1, \ldots, \delta$ (we will look at the last interval $I_{\delta+1}$ separately). By Lemma 6.1, 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 p deques in $\mathcal{R}'$, 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 steals that take place during the candidate timesteps; these are the only steals that may result in the execution of heavy premature nodes. Because a processor may attempt at most one steal in any timestep, each timestep can have at most p steals. Therefore, similar to the analysis in previous work [3], we can partition the candidate timesteps into ***phases***, such that each phase has between p and $2p-1$ steals. We call a phase in interval I_i ***successful*** if at least one of its $\Theta(p)$ steals 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 steals are chosen at random from the leftmost p deques with uniform probability, and because each phase has at least P steals, $\Pr[X_{ij}=1] \geq 1-(1-1/p)^p \geq 1-1/e > 1/2$. Thus, each phase succeeds with probability greater than $1/2$. Because u_i must get executed before or by the time x_i+1 successful steals target u_i's deque, there can be at most x_i+1 successful phases in interval I_i. Node u_i may get executed before x_i+1 steals 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 6.2, the x_i nodes below u_i are right children of nodes in P. There are $(\delta-1) < D$ nodes along P not including u_δ, and each of them may have at most one right child. Further, each successful phase in any of the first δ intervals results in at least one of these right children (or the current ready node on P) being executed. Therefore, the total number of successful phases in the first δ intervals is $\sum_{i=1}^{\delta} n_i < 2D$.

Finally, consider the final phase $I_{\delta+1}$. Let z be the ready node at the start of the interval with the highest priority. Note that $z \notin \sigma_p$ 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 6.2, 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 $\Theta(p)$ steals 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 the first τ timesteps of the parallel execution (*i.e.*, the time within which nodes from σ_p are executed) must have $< 3D$ successful phases. Each phase may result in $O(p)$ heavy premature nodes being stolen and executed. Further, for $i = 1, \ldots, \delta$, in each interval I_i, another $p-1$ heavy premature nodes may be executed in the same timestep that u_i is executed. Therefore, if σ_p has a total of N phases, the number of heavy premature nodes in σ_p is at most $(N+D)\cdot p$. Because the entire execution must have less than $3D$ successful phases, and each phase succeeds with probability $> 1/2$, the expected number of total phases before we see $3D$ successful phases is at most $6D$. Therefore, the expected number of heavy premature nodes in σ_p is at most $7D \cdot p = O(p \cdot D)$.

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 [112, Theorem 4.2] on the number of successful phases X, and setting $a = 6D + 8\ln(1/\epsilon)$, we get[1]

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

which implies that

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

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 $\Theta(p)$ steals. 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 6.4 *Let G be a dag with depth D, in which every node allocates at most K space, and for which the* 1DF*-schedule requires S_1 space. Let s_p be the p-schedule of length T 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_a. The nodes executed by P_a until its next steal, cannot allocate more than K space. Because there

[1]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.

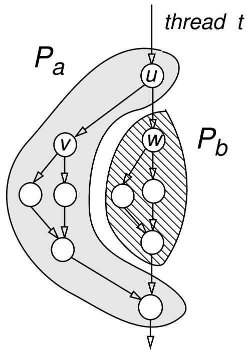


Figure 6.8: An example scenario when a processor may not execute a contiguous subsequence of nodes between steals. The shaded regions indicate the subset of nodes executed on each of the two processors, P_a and P_b. Here, processor P_a steals the thread t and executes node u. It then forks a child thread (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 (starting with node w) until it suspends. Subsequently, P_a finishes executing the child thread, 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 the 1DF-schedule.

are at most r heavy premature nodes executed, the total space allocated across all processors after i timesteps cannot exceed $S_1 + r \cdot K$.

We can now obtain a tighter bound when $K > S_1$. Consider the case when processor P_a steals a thread and executes a heavy premature node. The nodes executed by P_a 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 6.8). These intermediate nodes occur when other processors steal threads from the deque belonging to P_a, and finish executing the stolen threads before P_a finishes executing all the remaining threads in its deque. Subsequently, when P_a's deque becomes empty, the thread executing on P_a may wake up its parent, so that P_a starts executing the parent without performing another steal. Therefore, the set of nodes executed by P_a 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, 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 $S_1 + r \cdot \min(K, S_1)$. Because this bound holds for every prefix of s_p, it holds for the entire parallel execution.

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 $\min(K, S_1)$ threads with premature nodes can be forked for each heavy premature node executed. Therefore, the total number of active threads is at most $D + r \cdot \min(K, S_1)$. ∎

Each active thread requires at most a constant amount of space to be stored by the scheduler (not including stack space). Therefore, using Lemmas 6.3 and 6.4, we can state the following lemma.

Lemma 6.5 *Let G be a nested-parallel dag with depth D, and let every node in G allocate at most K space. Then the expected amount of space required to execute G on p processors using algorithm DFDeques (K), including scheduling space, is $S_1 + O(p \cdot D \cdot \min(K, S_1))$. Further, for any $\epsilon > 0$, the probability that the computation requires $S_1 + O(p \cdot (D + \ln(1/\epsilon)) \cdot \min(K, S_1))$ space is at least $1 - \epsilon$.* ∎

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 6.2. 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. This transformation takes place at runtime, and the on-line *DFDeques* algorithm generates a schedule for this transformed dag. Therefore, the bound on the space requirement of the generated schedule remains the same as the bound stated in Lemma 6.5; the final space bound is stated below.

Theorem 6.6 *Let G be a nested-parallel dag with depth D. Then the expected amount of space required to execute G on p processors using algorithm DFDeques (K), including scheduling space, is $S_1 + O(p \cdot D \cdot \min(K, S_1))$. Further, for any $\epsilon > 0$, the probability that the computation requires $S_1 + O(p \cdot (D + \ln(1/\epsilon)) \cdot \min(K, S_1))$ space is at least $1 - \epsilon$.* ∎

When K is a small constant amount of memory, the expected space requirement reduces to $S_1 + O(p \cdot D)$.

6.3.4 Lower bound on space requirement

We now show that the upper bound on space requirement, as stated in Theorem 6.6, is tight (within constant factors) in the expected case, for algorithm *DFDeques*.

Theorem 6.7 (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 the dag on p processors is $\Omega(S_1 + \min(K, S_1) \cdot p \cdot D)$.

Proof: Consider the dag shown in Figure 6.9. The black nodes denote allocations, while the grey nodes denote deallocations. The dag essentially has the a fork tree of depth $\log(p/2)$, at the leaves of which exist subgraphs[2]. 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 6.9 (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 6.9 (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 can 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 [112, Theorem 4.2] on the number of successful timesteps, we have

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

Therefore,

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

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

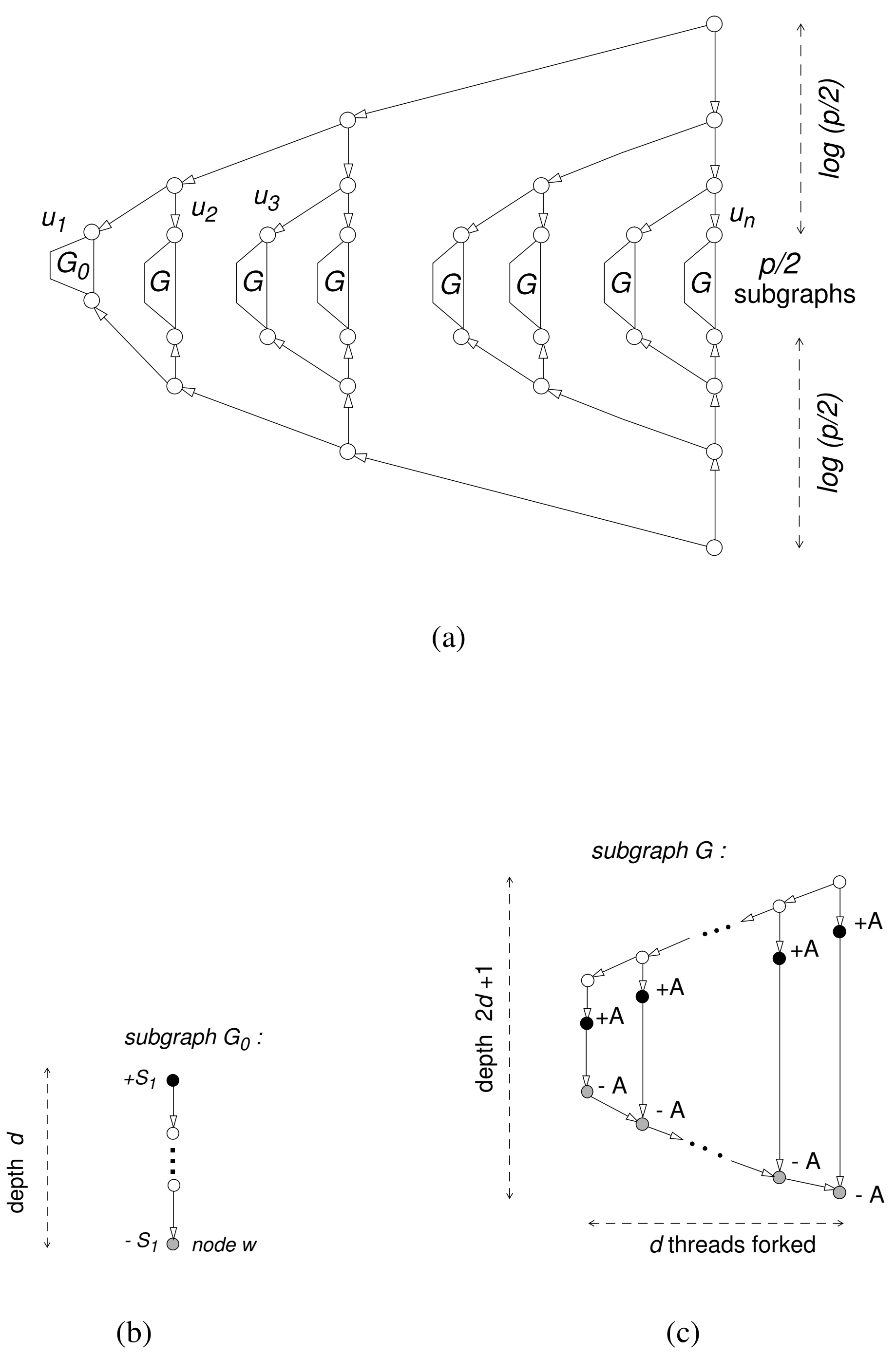


Figure 6.9 : (a) The dag for which the existential lower bound holds. (b) and (c) present the details of the subgraphs shown in (a). The black nodes denote allocations and grey nodes denote deallocations; the nodes are marked with the amount of memory (de)allocated.

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, $\Pr[\bigcup_1^n \mathcal{E}_i] \leq 2/3$. 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 (see Figure 6.9 (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 6.8 (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 [24] on p processors is $\Omega(S_1 \cdot p \cdot D)$. ∎

The corollary follows from Theorem 6.7 and the fact that algorithm *DFDeques* behaves like the space-efficient work-stealing scheduler for $K = \infty$. Blumofe and Leiserson [24] 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[3], 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 is also consistent with the upper bound of $p \cdot S$ by Simpson and Burton [143], 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.

[3]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.

6.3.5 Time bound

We now prove the time bound required for a parallel computation using algorithm *DFDeques*. This time bound do not include the scheduling costs of maintaining the relative order of the deques (*i.e.*, inserting and deleting deques in $\mathcal{R}'$), or finding the m^{th} deque. In Appendix D we describe how the scheduler can be parallelized, and then prove the time bound including these scheduling costs. We assume for now that every action allocates at most K space, for some constant K. We relax this assumption and provide the modified time bound at the end of this subsection.

Lemma 6.9 *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, $\Pr[X_j = 1] = (r/p) \cdot (1 - 1/p)^{r-1}$. Let X be the random variable representing the total number of non-empty deques that get exactly one steal request. Because there are at least r non-empty deques, the expected value of X is given by

$$\begin{aligned}
\mathrm{E}[X] &\geq \sum_{j=1}^{r} \mathrm{E}[X_j] \\
&= r \cdot \frac{r}{p} \cdot (1 - \frac{1}{p})^{r-1} \\
&\geq \frac{r^2}{p} \cdot (1 - \frac{1}{p})^{p} \\
&\geq \frac{r^2}{p} \cdot (1 - \frac{1}{p}) \cdot \frac{1}{e} \\
&\geq \frac{r^2}{2 \cdot p \cdot e}
\end{aligned}$$

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 $\mathrm{E}[Y] \geq (p - r) + r^2/2ep \geq p/2e$. Therefore, $\mathrm{E}[p - Y] \leq p - p/2e = p(1 - 1/2e)$. The quantity $(p - Y)$ must be non-negative;

therefore, using the Markov's inequality [112, Theorem 3.2], we get

$$\begin{aligned} \Pr[(p-Y) > p(1-1/4e)] &< \frac{\mathrm{E}[(p-Y)]}{p\left(1-\frac{1}{4e}\right)} \\ &\le \frac{\left(1-\frac{1}{2e}\right)}{\left(1-\frac{1}{4e}\right)}, \end{aligned}$$

which implies that

$$\begin{aligned} \Pr[Y < p/4e] &< 9/10, \\ \text{that is,} \quad \Pr[Y \ge p/4e] &> 1/10 \end{aligned}$$

We will call each timestep of type A ***successful*** if $\ge p/4e$ 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 6.3. 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 [112, Theorem 4.2], and setting $a = 40eW/p + 40\ln(1/\epsilon)$, we get[4]

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

Therefore,

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

[4]As with the proof of Lemma 6.3, 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.

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. Again, we split type B timesteps into phases such that each phase has between p and $2p - 1$ steals. 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. I 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 $\Theta(p)$ steals, the expected number of steals during type B timesteps is $O(pD)$. Further, for any $\epsilon > 0$, we can show that the total number of steals 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 that fails, or executes a node from the dag. Therefore, if N_{steal} is the total the number of steals 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 $P(A \cup B) \leq P(A) + P(B)$), 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 large allocations, recall that we add binary forks of dummy threads. If S_a is the total space allocated in the program (not counting the deallocations), the additional number of nodes in the transformed dag is $O(S_a/K)$. The transformation increases the depth of the dag by at most a constant factor. Therefore, using Lemma 6.9, the modified time bound is stated as follows.

Theorem 6.10 *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 $O(W/p + D)$, which is asymptotically optimal [28].

If we parallelize the scheduler, we can account for scheduling overheads. Then the computation can be executed in $O(W/p + D\log p)$ timesteps including scheduling costs. Such a parallel implementation of the scheduler is described and analyzed in Appendix D.

6.4 Summary

In this chapter, I have presented the *DFDeques* scheduling algorithm, which enhances algorithm *AsyncDF* with ideas from previous work stealing approaches. In particular, algorithm *DFDeques* was designed to handle finer grained threads more efficiently than algorithm *AsyncDF* by increasing scheduling granularity beyond the granularities of individual threads. The scheduling granularity is increased by scheduling fine-grained threads close in the dag on the same processor, without repeated accesses to the global scheduling queue. I have shown that the expected space requirement of the algorithm for executing a computation with D depth, W work and S_1 serial space requirement, is $S_1 + O(p \cdot D)$, while the expected running time is $O(W/p + D)$. In the next chapter, I describe experiments with implementing the *DFDeques* algorithm.

Chapter 7

Experimental Evaluation of Algorithm *DFDeques*

In Chapter 6, I presented the *DFDeques* scheduling algorithm, which automatically increases scheduling granularity. In this chapter, I describe experiments to evaluate the performance of algorithm *DFDeques* in practice. To evaluate the algorithm, I have implemented it in the context of the Solaris Pthreads package. Results of the Pthreads-based experiments presented in this chapter indicate that in practice, the new *DFDeques* scheduling algorithm results in better locality and higher speedups compared to both algorithm *AsyncDF* and the FIFO scheduler. (This difference is more pronounced for finer thread granularities.) *DFDeques* provides this improved performance by scheduling threads close in the dag on the same processor. This typically results in better locality and lower scheduling overheads. Throughout this chapter, algorithm *AsyncDF* actually refers to the variant of the original algorithm, that is, it does not use the buffer queues Q_{in} and Q_{out}; this variant was described in Chapter 5.

Ideally, we would like to compare the Pthreads-based implementation of *DFDeques* with a space-efficient work-stealing scheduler. However, implementing a provably space-efficient work-stealing scheduler (*e.g.*, [25]) that can support the general Pthreads functionality would require significant modification to both the scheduling algorithm and the Pthreads implementation[1]. Therefore, to compare algorithm *DFDeques* to an existing, space-efficient work-stealing scheduler, I instead built a simple simulator that implements synthetic, purely nested-parallel benchmarks. The simulation results indicate that by adjusting the memory threshold, *DFDeques* can cover a wide range of space requirements and scheduling granularities. At one extreme it performs similar to algorithm *AsyncDF*, with a low space requirement and small scheduling granularity. At the other extreme, it behaves exactly like a work-stealing scheduler, with a higher space requirement and larger scheduling granularity.

This chapter is organized as follows. Section 7.1 describes the implementation of the algorithm in the context of Pthreads, along with the results of executing the set of Pthreads-based benchmarks that were introduced in Chapter 5. Section 7.2 presents the simulation results that compare *DFDeques* with *AsyncDF* and a work-stealing scheduler, and Section 7.3 summarizes

[1]The Pthreads implementation itself makes extensive use of blocking synchronization primitives such as Pthreads mutexes and condition variables. Therefore, even fully strict Pthreads benchmarks cannot be executed using such a work stealing scheduler in the existing Solaris Pthreads implementation.

the results presented in this chapter.

7.1 Experiments with Pthreads

I have implemented the *DFDeques* scheduling algorithm as part of the native Solaris Pthreads library described in Chapter 5 (Section 5.2). The machine used for all the experiments in this Section is the 8-processor Enterprise 5000 SMP described in Section 5.2. Recall that because Pthreads are not very fine grained, the user is required to create Pthreads that are coarse enough to amortize the cost of thread operations. This section shows that by using algorithm *DFDeques*, high parallel performance can be achieved without any additional coarsening of threads. 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.

As with the implementation of algorithm *AsyncDF* for scheduling Pthreads, I 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. Recall that algorithm *DFDeques* (Figure 6.4) was designed for purely nested parallel computations. However, the scheduler implementation described in this section is an extension of algorithm *DFDeques* that supports the full Pthreads functionality (including mutexes and condition variables). To support this functionality, it maintains additional entries in $\mathcal{R}'$ for threads suspended on synchronizations. When a thread suspends, a new deque is created for it to the immediate right of its current deque. Although the benchmarks presented in this chapter make limited use of mutexes and condition variables, the Pthreads library itself makes extensive use of them both.

Since the 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. I optimized the common cases of pushing and popping threads onto a processor's current deque by minimizing the synchronization time. A steal requires the scheduler lock to be held for a longer period of time, until the processor selects and steals from a target stack. This Pthreads-based implementation uses a slightly different stealing strategy than the one described in the original *DFDeques* algorithm (Figure 6.4). In this implementation, if an idle processor tries to steal from a deque that is currently not owned by any processor, the processor takes over ownership of that deque, and starts executing the top thread in that deque. Recall that in the original algorithm, the processor would instead steal the bottom thread from the deque and create a new deque for itself. The goal of this modification was to reduce the number of deques present in $\mathcal{R}'$; in practice, it also results in a small additional increase in the scheduling granularity.

In the Solaris Pthreads implementation, it is not always possible to place a reawakened thread on the same deque as the thread that wakes it up. Further, unlike in nested parallel computations, a thread that wakes up another thread may continue execution. Therefore, instead of requiring the processor that wakes up a thread to move the newly awakened thread to its deque, the thread remains in its current deque (that was created for it when it suspended). Therefore, this implementation of *DFDeques* is an approximation of the pseudocode in Figure 6.4. Further, since access to $\mathcal{R}'$ is serialized, and mutexes and condition variables are supported and used by the Pthreads implementation, setting $K = \infty$ does not produce the same schedule as the space-efficient work-stealing scheduler intended for fully strict computations [24]. Therefore, we can use this setting only as a rough approximation of a pure work-stealing scheduler.

I first list the benchmarks used in the experiments. Next, I compare the space and time performance of the library's original FIFO scheduler (labelled "FIFO") with the implementation of algorithm *AsyncDF* (labelled "ADF") as described in Chapter 5, and algorithm *DFDeques* (labelled "DFD") for a fixed value of the memory quota K. *DFDeques* (∞) is used as an approximation for a space-efficient work-stealing scheduler (labelled "DFD-inf"). To study how the performance of the schedulers is affected by thread granularity, I present results of the experiments at two different thread granularities. Finally, I measure the trade-off between running time, scheduling granularity, and space requirement using the new scheduler by varying the value of K for one of the benchmarks.

7.1.1 Parallel benchmarks

The Pthreads-based parallel benchmarks and the inputs used are described in detail in Chapter 5. 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 of the recursion tree. In the comparison results in Section 7.1.2, ***fine*** granularity refers to the thread granularity that provides good parallel performance using algorithm *AsyncDF* (this is the thread granularity used in Chapter 5). As shown in that chapter, even at this granularity, the number of threads significantly exceeds the number of processors, allowing for simple code and automatic load balancing, and yet resulting in performance equivalent to hand-partitioned, coarse-grained code. ***Finer*** granularity refers to the finest thread granularity that allows the cost of thread operations in a single-processor execution add up to at most 5% of the serial execution time[2]. Ideally, at this finer thread granularity, the parallel executions of the programs should also require at most 5% more time than their fine (or coarse) grained versions.

The two thread granularities for each of the benchmarks are specified below:

1. **Volume Rendering.** Each thread processes up to 64 4×4 pixel tiles of the rendered image at fine granularity, and up to 5 tiles at finer granularity.

2. **Dense Matrix Multiply.** The fine grained version uses 64×64 blocks at the leafs of the recursion tree in a divide-and conquer algorithm; the finer grained version uses 32×32 blocks.

3. **Sparse Matrix Vector Multiply.** At fine granularity, 64 threads are used to perform the multiplication, while 256 threads are used at the finer granularity.

4. **Fast Fourier Transform.** 256 threads are used to compute the 1D FFT at fine granularity, and 512 threads at finer granularity.

5. **Fast Multipole Method.** At fine granularity, each thread calculates 25 interactions of a cell with its neighboring cells, while at finer granularity, each thread computes 5 such interactions.

[2] The exception was the dense matrix multiply, which is written for $n \times n$ blocks, where n is a power of two. Therefore, finer 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.

Benchmark	Input size	Fine grained				Finer 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 7.1 : Input sizes for each benchmark, the total number of threads expressed in the program at fine and finer granularities, and the maximum number of simultaneously active threads created by each scheduler at both granularities. K was set to 50,000 bytes. The results for "DFD-inf" are not shown here; it creates up to twice as many threads as "DFD" for dense matrix multiply, and at most 15% more threads than "DFD" for the remaining benchmarks.

6. **Barnes-Hut.** At fine granularity, each thread computes the forces on particles in up to 8 leaf cells (on average), while at the finer granularity, each thread handles one leaf cell.

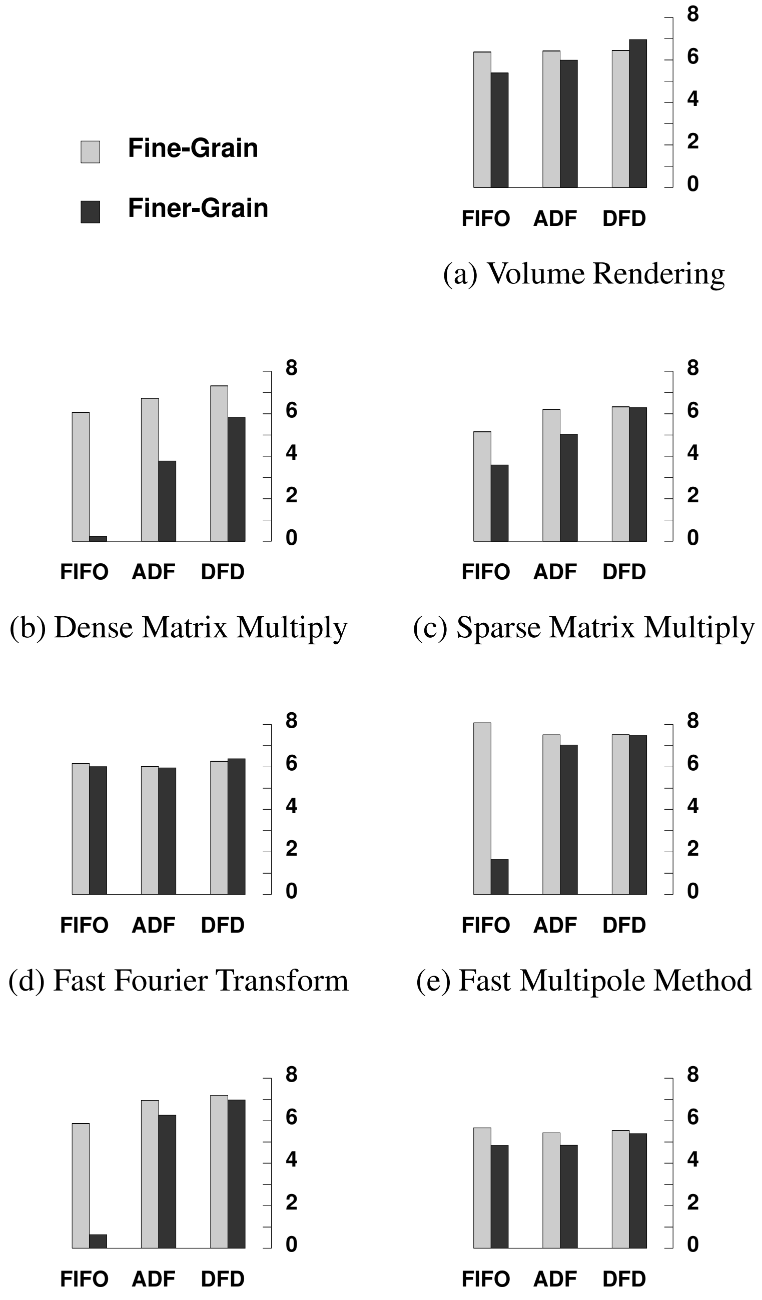
7. **Decision Tree Builder.** The thread granularity is adjusted by setting a threshold for the number of instances, below which the recursion is executed serially. The thresholds were set to 2000 and 200 for the fine and finer granularities, respectively.

7.1.2 Comparison results

In all the comparison results, the memory threshold was set to K =50,000 bytes for the *AsyncDF* and *DFDeques* algorithms. An 8KB (1 page) stack was allocated for each active thread. In each of the three versions (FIFO, *AsyncDF*, and *DFDeques*), the default Pthread stack size was set to 8KB, so that the Pthread implementation could cache previously-used stacks. In general, the space-efficient schedulers (*AsyncDF* and *DFDeques*) effectively conserve stack memory by creating fewer simultaneously active threads compared to the original FIFO scheduler (see Figure 7.1). The FIFO scheduler was found to spend significant portions of time executing system calls related to memory allocation for both stack and heap space.

The 8-processor speedups for all the benchmarks at the fine and finer thread granularities are shown in Figure 7.2. To concentrate on the impact of the scheduler, and to ignore the effect of increased thread overheads (up to 5% for all except dense matrix multiply), speedups for each thread granularity shown here are computed with respect to the single-processor execution of the *multithreaded* program at that granularity. The speedups indicate that both space-efficient algorithms *AsyncDF* and *DFDeques* outperform the library's original scheduler. However, at the finer thread granularity, algorithm *DFDeques* provides better performance than algorithm *AsyncDF*. This difference can be explained by the better locality and lower scheduling contention incurred by algorithm *DFDeques*.

For one of the benchmarks (dense matrix multiply), even algorithm *DFDeques* suffers a slow-down at finer granularity. Due to the use of a single scheduling queue protected by a common lock,



(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 7.2: Speedups on 8 processors with respect to single-processor executions for the three schedulers (the original "FIFO" scheduler, algorithm *AsyncDF* or "ADF", and algorithm *DFDeques* or "DFD") at both fine and finer thread granularities. K was set to 50,000 bytes. Performance of "DFD-inf", 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`.

Benchmark	FIFO	ADF	DFD
Vol. Rend.	4.2	3.0	1.8
Dense MM	24.0	13	8.7
Sparse MVM	13.8	13.7	13.7
FFTW	14.6	16.4	14.4
FMM	14.0	2.1	1.0
Barnes Hut	19.0	3.9	2.9
Decision Tr.	5.8	4.9	4.6

Figure 7.3 : L2 Cache miss rates (%) for Pthreads benchmarks at the finer thread granularities. "FIFO" is the original Pthreads scheduler that uses a FIFO queue, "ADF" algorithm *AsyncDF*, and "DFD" is algorithm *DFDeques*.

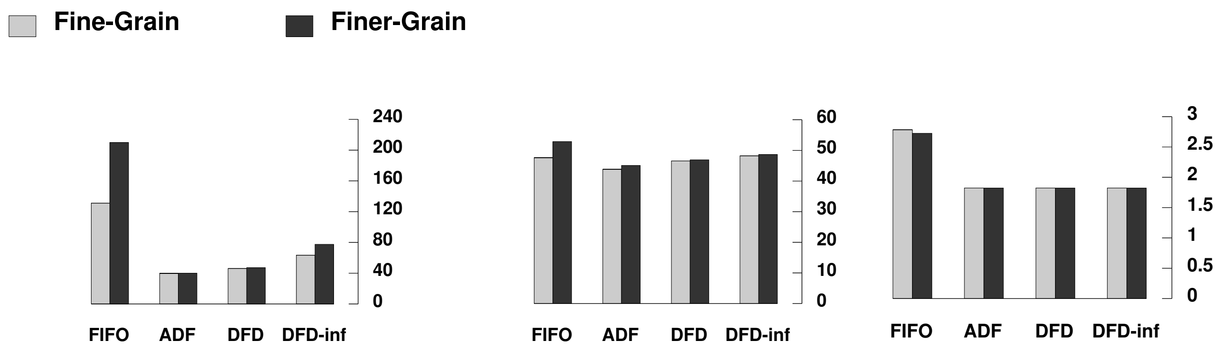
contention for the lock increases at the finer thread granularity. This slowdown indicates that the current implementation of Pthreads, with a single scheduling lock that also protects other shared data structures, will not efficiently support very fine-grained threads.

To verify that *DFDeques* results in better locality compared to the FIFO and *AsyncDF* schedulers, I measured the external (L2) cache miss rates for each benchmark using on-chip UltraSPARC performance counters. Figure 7.3, which lists the results at the finer granularity, shows that *DFDeques* achieves relatively low cache miss rates.

Three out of the seven benchmarks make significant use of heap memory. For these benchmarks, I measured the high water mark for heap memory allocation using the three schedulers. Figure 7.4 shows that algorithm *DFDeques* results in slightly higher heap memory requirement compared to algorithm *AsyncDF*, but still outperforms the original FIFO scheduler.

The Cilk [66] runtime system uses a provably space-efficient work stealing algorithm to schedule threads. Because it requires `gcc` to compile the benchmarks (which results in slower code compared to the native `cc` compiler on UltraSPARCs), a direct comparison of running times of Cilk benchmarks with my Pthreads-based system is not possible. However, in Figure 7.5 I compare the space performance of Cilk with algorithms *AsyncDF* and *DFDeques* for the dense matrix multiply benchmark (with 32×32 blocks). The figure indicates that *DFDeques* requires more memory than *AsyncDF*, but less memory than Cilk. In particular, like *AsyncDF*, the memory requirement of *DFDeques* increases slowly with the number of processors.

To test the performance of the schedulers on a benchmark that makes extensive use of locks throughout the execution, I measured the performance of just the octree-building phase of Barnes Hut. Threads in this phase are forked in a divide-and-conquer fashion; at the leafs of the recursion tree, each thread handles a constant number of particles, which it inserts into the octree. For inserting particles into the octree, I used the naive algorithm from the original SPLASH-2 benchmark [163]. Each particle is inserted starting at the root, and filters down the octree depending on its coordinates. When it reaches a leaf cell, it is inserted into the leaf; the leaf may consequently be subdivided if it now contains too many particles. Any modifications to the cells in the octree are protected by a lock (mutex) on the cell. Thus, the benchmark involves frequent locking of different cells of the octree. The benchmark does not scale very well because threads



(a) Dense Matr. Mult. (b) Fast Multipole Method (c) Decision Tree Builder

Figure 7.4: 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 my approximation of work stealing using *DFDeques* (∞).

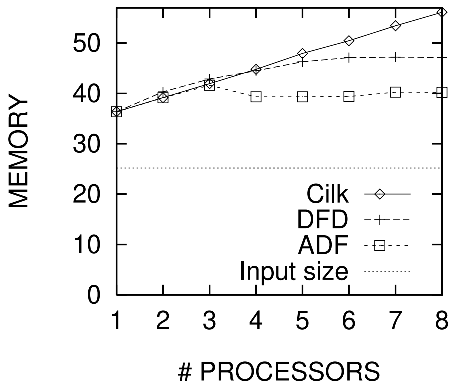


Figure 7.5: Variation of the memory requirement with the number of processors for dense matrix multiply using three schedulers: *AsyncDF* ("ADF"), *DFDeques* ("DFD"), and Cilk ("Cilk").

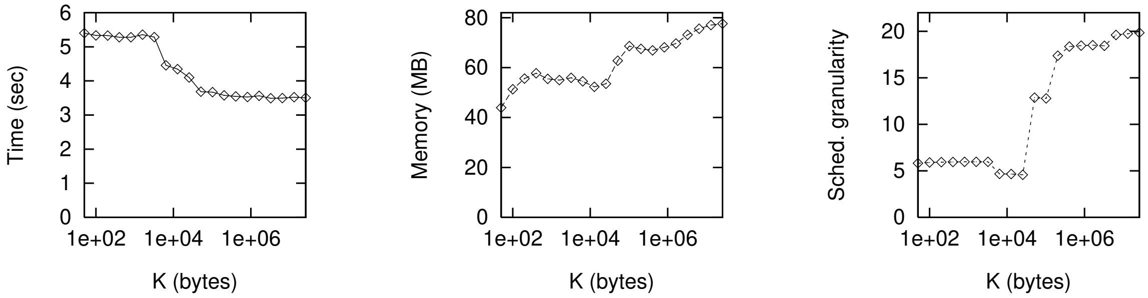
Figure 7.6: Speedups for the octree-building phase of Barnes Hut (for 1M particles). The phase involves extensive use of locks on cells of the tree to ensure mutual exclusion. The Pthreads-based schedulers (all except Cilk) support blocking locks.

often contend for locks on cells near the root of the tree. Figure 7.6 compares the performance of *DFDeques* with the original FIFO scheduler, *AsyncDF*, and with the Cilk runtime system. Due to limited use of floating point arithmetic, the single-processor performance of the benchmark on Cilk was comparable to that of my Pthreads-based implementation, making this comparison of speedups meaningful. Because threads suspend often on mutexes, *DFDeques* does not result in a large scheduling granularity; therefore, its performance is comparable to that of *AsyncDF*. The FIFO scheduler performs poorly due to the creation of an excessive number of Pthreads. The difference between my Pthreads-based schedulers and the Cilk scheduler is probably explained by the fact that Cilk uses a spin-waiting implementation for locks, while my Pthreads schedulers support blocking implementations of Pthread mutexes. Algorithm *DFDeques* could be easily extended to support blocking synchronization because it inherently allows more deques than processors; thus, when a thread suspends, it is placed on a new deque, where it resides when it is subsequently reawakened. Future work, as described in Chapter 8, involves more extensive experiments with such non-nested-parallel benchmarks.

7.1.3 Tradeoff between space, time, and scheduling granularity

Recall that between consecutive steals, each processor executes threads from a single deque. If the memory quota is exhausted by these threads and some thread reaches a memory allocation that requires more memory, the processor preempts the thread, gives up the deque, and performs the next steal. Therefore, when the value of the memory threshold is raised, the scheduling granularity typically increases. A larger scheduling granularity leads to better time performance; however, a larger memory threshold may also lead to a higher space requirement.

This section shows how the value of the memory threshold K affects the running time, the memory requirement, and the scheduling granularity in practice. 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 used as an 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 7.7 shows the resulting trade-off for this benchmark at the finer thread granularity. As expected, both memory and scheduling granularity increase with K, while running time reduces as K is increased.



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

Figure 7.7 : Trade-off between running time, memory allocation and scheduling granularity using algorithm *DFDeques*, as the memory quota K is varied, for the dense matrix multiply benchmark at the finer thread granularity.

7.2 A Simulator to Compare *DFDeques* with Previous Schedulers

Fully strict parallelism [24] does not support the use of synchronization primitives such as mutexes or condition variables, that are part of the standard Pthreads interface. Therefore, implementing a pure, space-efficient work stealing algorithm [24] intended for fully strict computations in the context of a Pthreads implementation would involve significant changes to the algorithm and to the implementation. To compare algorithm *DFDeques* with a work stealing algorithm, I instead built a simple system that simulates the parallel execution of fully strict synthetic benchmarks. The benchmarks represent recursive, divide-and-conquer computations expressed as a binary tree of forks, in which the memory or time requirements at each level of the tree can be varied. The implementation simulates a space-efficient work scheduler [24] (labeled "WS"), algorithm *AsyncDF* (labeled "ADF"), and algorithm *DFDeques* (labeled "DFD").

This section describes the synthetic benchmarks and their properties that can be varied in the simulator. I then present an overview of the implementation, followed by the simulation results.

7.2.1 Parallel computations modeled by the simulator

I simulate the execution of threads forked as a balanced binary tree with adjustable memory requirements and granularity. The simulated computation begins with a root thread, which may allocate some memory, and execute some number of instructions. It then forks 2 child threads; the child threads may allocate memory and execute instructions, after which they forks 2 child threads each, and so on. The forking stops when the specified nesting level is reached. When both children of a thread terminate, the thread deallocates the memory it had allocated and terminates immediately. All the real work is executed by threads before they fork; after the fork they simply synchronize with their children and exit. Thus the nodes representing the real work form a balanced binary tree (see Figure 7.8), which is referred to as the "tree" in the rest of this section. Each node in this tree may represent multiple actions that are executed serially (instead of a single

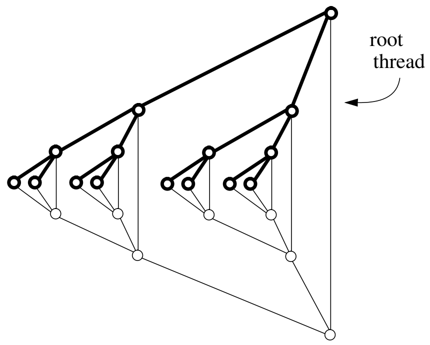


Figure 7.8: A sample dag for the class of computations simulated by the simulator. Each thread forks 2 children, which fork 2 children each, and so on. Each node here represents a variable number of actions performed serially, rather than just one action. The "real work" in each thread is executed before the fork; after the fork, the thread simply synchronizes with its 2 child threads and exits. The nodes performing the real work and the edges joining them are shown in bold; the simulator generates computations where these nodes and edges form a balanced binary tree as shown here.

action). In the remainder of this section, the ***depth*** of a thread refers to its depth in the recursion tree, that is, the number of threads forked on the path from the root thread to the given thread.

The characteristics of the threads in the binary tree that can be varied in the simulator are described below.

- **Granularity.** The ***granularity*** of each thread refers to the number of actions it executes before it forks any children. When a granularity of n units if specified for a thread, the simulator can either set the thread's granularity to an exact n units, or to a value chosen uniformly at random in the range $[1, 2n]$. The random values are intended to model irregular parallelism[3]. Further, the average granularity of a thread can be varied as follows.

  G1. Equal for all threads. This setting models simple recursive calls where an approximately equal amount of work is done in each recursive call.

  G2. Decreasing geometrically with the depth of the thread. Geometrically decreasing granularity is intended to model divide-and-conquer algorithms for which the work decreases with the depth in the recursion tree.

  G3. A small, constant granularity for all interior threads, and a larger, constant value for the leaf threads in the tree. This models a parallel loop executed as a binary tree of forks, in which the real work is executed at the leaves.

- **Memory requirement.** The ***memory requirement*** of a thread is the amount of memory it allocates when it starts executing. The memory requirement for each thread can be varied in the scheduler as follows.

[3] An alternative way to model irregularity in the execution tree would be to execute an irregular binary tree instead of a balanced tree. I chose to vary the thread granularities for simplicity.

M1. Constant for all threads. This models recursive divide-and-conquer computations that do not dynamically allocate memory, but require a small, constant amount of stack space for each recursive call, which is executed by a separate thread.

M2. Decreasing geometrically with the depth. This models divide-and-conquer algorithms that dynamically allocate memory at each stage of the recursion, and the allocations reduce with depth.

M3. Constant for all threads except all threads at a specified depth i; threads at depth i allocate a larger, constant amount of memory. This option is used with option G3 (*i.e.*, large granularity at the leaves), and models two nested parallel loops where the outer loop has 2^i iterations and allocates a large amount of memory at the start of each iteration, and then executes an inner (nested) loop with $2^{(d-i)}$ iterations, where d is the depth of the tree (that is, the number of threads from the root to a leaf). Thus threads at depth i represent the iterations of the outer loop forked as a binary tree. All other threads simply require a small, constant amount of space for their stacks.

7.2.2 Implementation of the Simulator

The simulator maintains a linked list of deques with threads stored in their depth-first execution order (as shown in Figure 6.3). Each processor may own at most one deque. In each timestep, the state of each processor is checked and modified. If the processor needs to execute a steal, a steal request for it is queued. At the end of the timestep, the steal requests are processed in a random order to avoid giving preference to any processors. All the steal requests that pick a particular processor's deque as their target are queued at the processor, and in each timestep, the processor picks one of the requests at random and services it. If its deque is empty, it returns a failure to the stealer, and otherwise it returns a thread from its deque. Each steal requires 1 timestep, and at most one steal request is serviced by each target processor in each timestep. A thread with granularity g implies it has g actions, where each action takes one timestep to execute.

In the simulation of algorithm *DFDeques*, each of the p processors makes steal requests to one of the leftmost p non-empty deques chosen at random, and threads are stolen from the bottom deques. The first processor to make a steal request to a deque not owned by any processor becomes the new owner of the deque; this change from the original *DFDeques* algorithm does not significantly modify the simulation results presented in this section. For algorithm *AsyncDF*, the top thread from the leftmost deque is always stolen by an idle processor. In both algorithms, a binary tree of m/K dummy threads is inserted before every large allocation of m ($m > K$) units. A processor must give up its deque and steal every time it executes a dummy thread. I assume the overhead of creating, executing and deleting a dummy thread is a constant number of timesteps. For both algorithms *AsyncDF* and *DFDeques*, the simulator assumes that deques are created, inserted into the ordered queue, and deleted from it at no additional cost. In the simulation of algorithm *AsyncDF*, a processor is allowed to use the memory quota of K units for multiple threads from a deque (instead of just one thread). This leads to a higher scheduling granularity compared to the original *AsyncDF* algorithm.

To implement the space-efficient work-stealing scheduler, the memory quota K is simply set to infinity in algorithm *DFDeques*. This ensures that a processor never gives up ownership of its deque due to a memory allocation, and there are always p deques in the system, one owned by each of the p processors.

7.2.3 Simulation results

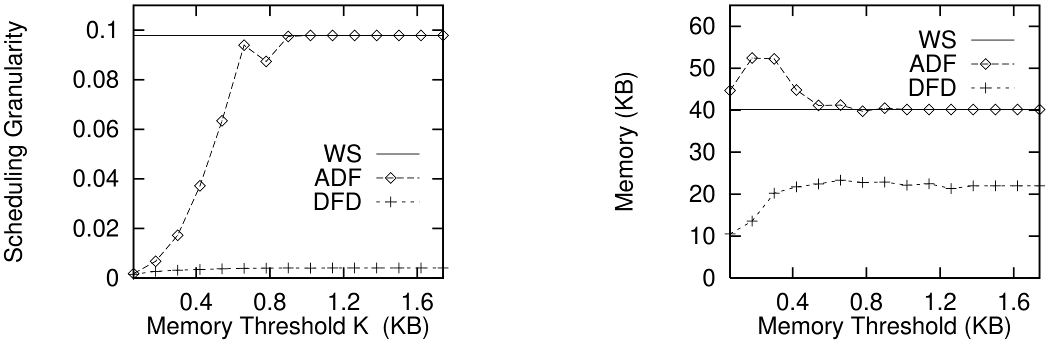
For each of the following computations, I measured the scheduling granularity (the average number of actions executed by a processor between two steals) and the total memory requirement (the high water mark of the total memory allocated across all processors) as the memory threshold K is varied. An ideal scheduler would result in a high scheduling granularity and a low memory requirement. All the experiments simulated binary trees of depth 15 on 64 processors. The units of memory allocation are assumed to be bytes; the memory requirements for threads were set such that the total memory allocated by all the $2^{16} - 1$ threads was approximately 4MB.

(a) G3 + M1: Simple binary tree or parallel loop with constant memory requirements

This experiment simulates a simple binary tree, where the memory requirement of each thread was selected uniformly at random, with a constant expected value. The granularity of interior threads is set to a small constant (2 units), while the granularity at the leaves is set to 50 units. Computations represented by such graphs include a simple divide-and-conquer algorithm or a parallel loop implemented as a tree, where the main work is executed at the leaves. The results in Figure 7.9 show that algorithm *DFDeques* is effective in increasing the granularity compared to the *AsyncDF*, but is not effective in controlling memory requirements, even compared to the WS scheduler. For such a computation with a depth D, the serial space S_1 is $O(D)$, and therefore the bounds of $p \cdot S_1$ and $S_1 + O(p \cdot D)$ are asymptotically equivalent. For small values of K, the *DFDeques* scheduler results in more preempted threads compared to WS, which explains the greater memory requirement. *DFDeques* behaves similar to WS when K is large compared to the memory requirement of individual threads. In practice, since most threads do very little work and typically require very small stacks in such a program, the value of K set by the user should indeed be large enough to provide good performance using *DFDeques*. The results were similar with constant granularity for all threads (instead of distinguishing the leaves from the interior threads).

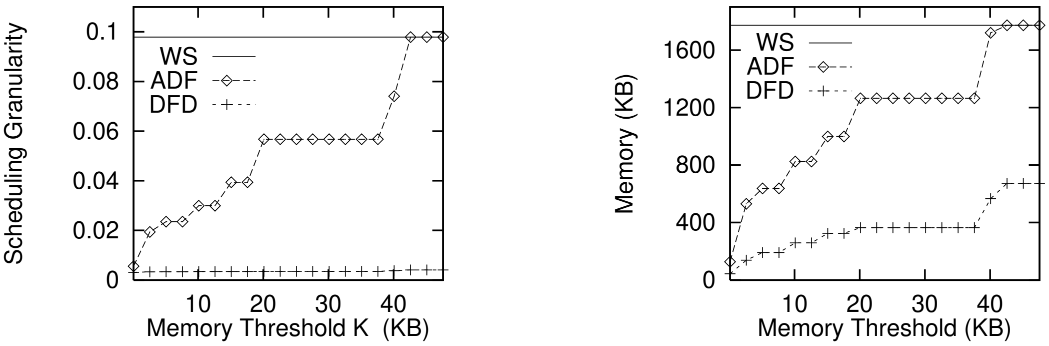
(b) G3 + M3: Nested loop with memory allocation in outer loop

This experiment simulates a binary tree of threads in which the memory requirement of each thread at depth 6 is large (40KB), while all other threads have (on average) a small constant memory requirement. Such a tree represents a nested parallel loop, with $2^6 = 64$ iterations in the outer loop, and $2^{15-6} = 512$ iterations in each inner loop. Each iteration of the outer loop allocates a large amount of memory to execute an inner loop. The thread granularities were set to a large constant value (50 units) at the leaves of the tree, and a small constant value (2 units) for all interior threads. We now begin to see the benefits of algorithm *DFDeques* (see Figure 7.10), since S_1 is no longer just $O(D)$; the value of K now provides a trade-off between the granularity and the memory requirement. Figure 7.10 also shows that work stealing results in high scheduling granularity and high space requirement, while *AsyncDF* results in low scheduling granularity and low space requirement. In contrast, *DFDeques* allows scheduling granularity to be traded with space requirement by varying K.



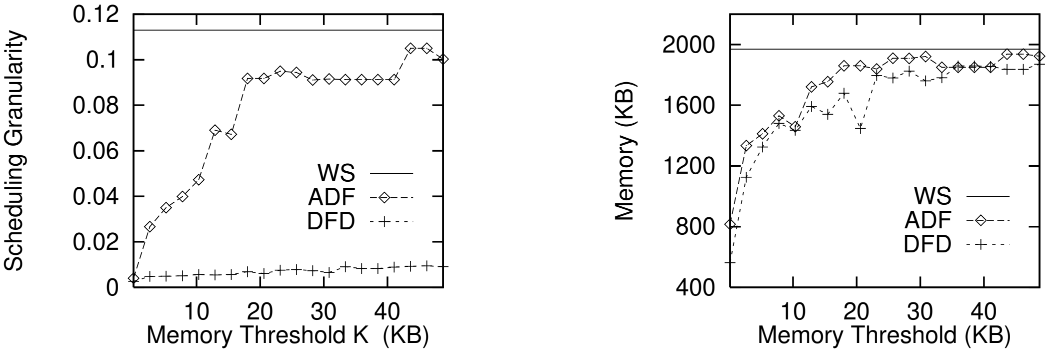
(a) Granularity (b) Memory

Figure 7.9: Simple binary tree or parallel loop with constant (average) granularities and memory requirements for each thread.



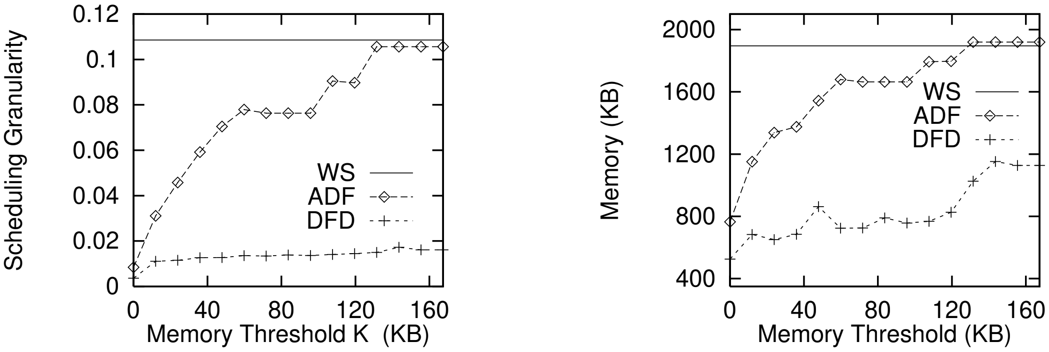
(a) Granularity (b) Memory

Figure 7.10: Nested loop with memory allocation in the outer loop.



(a) Granularity (b) Memory

Figure 7.11: Geometrically decreasing memory and random, non-decreasing granularity.



(a) Granularity (b) Memory

Figure 7.12: Geometrically decreasing memory and granularity.

(c) G1 + M2: Decreasing memory and random, non-decreasing granularity

In this experiment, the memory requirements of the threads were set to decrease geometrically (by a factor of 2) with their depth. The thread granularities are selected uniformly at random with 50 units as the expected value. Figure 7.11 shoes the results for this case. As before, the value of the memory threshold K is effective in generating a trade-off between space and granularity for algorithm *DFDeques*, which covers a range of behavior between that of algorithms WS and *AsyncDF*.

(d) G2 + M2: Geometrically decreasing memory and granularity

Figure 7.12 shows the results when both the memory requirements and the average granularity of the threads decreases geometrically (by a factor of 2) with the depth. Programs with such characteristics include divide-and-conquer algorithms such as a recursive matrix multiply or sort (not in-place). Once again, we see that *DFDeques* allows granularity to be traded with space by varying K.

7.3 Summary

In this chapter, I have described experiments with implementing the *DFDeques* scheduling algorithm that was presented in Chapter 6. The results of executing Pthreads-based benchmarks using the *DFDeques* scheduling algorithm indicate that it outperforms both the original FIFO and *AsyncDF* schedulers for finer-grained Pthreads. It's memory requirement is higher than *AsyncDF*, but lower than previous work-stealing schedulers and the FIFO scheduler. Further, simulation results for executing simple nested-parallel benchmarks using algorithm *DFDeques* indicate that the memory threshold provides a trade-off between the space requirements and the scheduling granularity.

Chapter 8

Conclusions

This chapter first summarizes the contributions of this dissertation, and then describes possible directions for future research.

8.1 Thesis Contributions

(a) Scheduling algorithms. I presented two asynchronous scheduling algorithms, *AsyncDF* and *DFDeques*, that provide provable upper bounds on the space and time required to execute a parallel program. The bounds are expressed in terms of its total work, depth (length of the critical path) and serial space requirement. In particular, a program with a serial space requirement of S_1 and depth D can be executed on p processors using $S_1 + O(K \cdot p \cdot D)$ space[1]. Here, K (the memory threshold) is a user-adjustable runtime parameter, which provides a trade-off between running time and space requirement. I presented and analyzed serialized schedulers for both algorithms. I also described how to parallelize the schedulers, and analyzed the the total space and time requirements including scheduling overheads.

(b) Runtime systems. This dissertation has described the implementation of a specialized runtime system on the SGI Power Challenge SMP, that uses algorithm *AsyncDF* to schedule lightweight threads. I have also implemented algorithms *AsyncDF* and *DFDeques* in the context of a commercial user-level Pthreads library for Solaris-based SMPs. Although the space and time bounds of both algorithms were analyzed for purely nested parallel programs, their Pthreads-based implementations may, in practice, be used to execute more general styles of parallelism. The modified library supports the complete Pthreads API, including signal-handling and blocking synchronization. Therefore, any Pthreads programs can benefit from the new scheduling techniques.

(c) Benchmark implementation and evaluation. I used a variety of benchmarks with irregular or dynamic parallelism on the multithreaded runtime systems to evaluate the effectiveness of the two scheduling algorithms. These benchmarks include dense and sparse matrix multiplies, two N-body codes, a volume rendering benchmark, a high performance FFT package, and a data classifier. In contrast to the original FIFO scheduler, my new schedulers allow simpler code for the fine-grained

[1]For the *DFDeques* algorithm, this is the space bound in the expected case.

benchmarks to obtain the same high performance as their hand-partitioned, coarse-grained counterparts. My experimental results also indicate that the new scheduling algorithms are effective in reducing the memory requirements of the benchmarks compared to previous schedulers.

(d) Analysis of space-time trade-offs. In both scheduling algorithms, the memory threshold acts as a user-adjustable parameter providing a tradeoff between space and time requirements. The theoretical analysis in this dissertation reflects this tradeoff. In addition, I experimentally demonstrate the tradeoff in the execution of the parallel benchmarks. For algorithm *DFDeques*, I also demonstrate the trade-off between space requirement and scheduling granularity.

8.2 Future Work

The research presented in this dissertation can be extended in several ways.

(a) Beyond nested parallelism. The most obvious direction for future research is to extend both the analysis and the experiments to benchmarks that have a structure more general that nested parallelism. In particular, the approach for analyzing the space bound used in this dissertation can be applied to more general styles of parallelism if a well-defined, serial schedule can be identified. If the relative thread priorities determined by this serial schedule can be maintained (on-line) in a fairly efficient manner, then the space and time requirements of the scheduler can be bounded by a reasonable value. We have used such an approach to obtain a space- and time-efficient scheduler for programs with synchronization variables [19]. This scheduler could be used, for example, to provide a provably space-efficient implementation of languages with futures [77, 95, 33, 64, 80]. However, for programs with arbitrary synchronizations (such a locks), identifying the "natural" serial schedule becomes difficult. Further, programs with such synchronizations are typically not dag-deterministic. Therefore, the dag for each execution of the program may be significantly different. One approach is to analyze the space requirement in terms of the worst-case space requirement over serial schedules for all possible dags. However, the dags may vary significantly, and therefore identifying the possible serial schedules is a difficult problem [143]. An alternate approach is to simply specify the space requirement in terms of the serial space requirement of the dag for the particular parallel execution under consideration. Again, determining this space requirement either offline or by running the program serially may not be possible.

The benchmarks presented in this dissertation, including the Pthreads-based codes, are predominantly nested parallel. They use a limited amount of locking (such as in the tree construction phase in Barnes-Hut); in addition, the Pthreads library uses a moderate number of locks and condition variables in its own implementation. However, since the scheduling algorithms support the full Pthreads interface, in the future they should be evaluated for fine-grained programs with more general styles of parallelism. For example, it is possible that the scheduling algorithms will have to be modified to efficiently execute programs that make an extensive use of locks.

(b) Finding the right memory threshold. Recall that the memory thresholds K is a user-adjustable parameter that can be used to adjust the space-time trade-off in both the scheduling algorithms. One drawback of using this parameter is that the user must set it to a "reasonable" value. This value may vary for different applications, and may depend on the underlying thread implementation. In all my experiments, on a given multithreaded system, the same value of K works well for all the tested benchmarks. However, in practice, other benchmarks may exhibit very different trade-off curves. Future work involves having the system automatically set the value of K for each program. For example, it could use a trace-based scheme that runs the program for small problem sizes to learn the nature of the trade-off curve, and use the results to predict a suitable, static value of K for larger, more realistic problem sizes. Alternatively, the system could dynamically adjust the value of K as the benchmark runs. The value of K (and hence also the number of dummy threads added) would be selected based on some heuristics, or if possible, in some provably efficient manner. For both the static and dynamic approaches, the user could specify some space and time constraints (such as, how much space overhead in addition to the serial space requirement can be tolerated, or how much slowdown over the best possible case is acceptable).

(c) Supporting very fine-grained threads. Recall that each processor in the *DFDeques* algorithm treats its deque as a regular stack. Further, in a nested-parallel program with very fine-grained threads, the fine-grained threads typically do not allocate large amounts of memory. Consequently, a processor in *DFDeques* would often execute a large number of threads from a single deque between steals. The algorithm should, therefore, benefit from stack-based optimizations such as lazy thread creation [70, 111]; these methods avoid allocating resources for a thread unless it is stolen, thereby making most thread creations nearly as cheap as function calls. Future work involves using *DFDeques* to execute very fine-grained benchmarks in the context of a runtime system that supports such optimizations.

(d) Scaling beyond SMPs. All the experiments have been conducted on single SMP with up to 16 processors. The use of globally ordered data structures implies that the *AsyncDF* and *DFDeques* scheduling algorithms are better suited for such tightly-coupled parallel machines. Further, processors in SMPs have hardware-coherent caches of limited size (a few megabytes) and they do not support explicit data placement in these caches. Therefore, as long as a thread (or scheduling unit) executes on a processor long enough to make reasonable use of the processor's cache, which particular processor it is scheduled on has little impact on the overall performance. In contrast, on distributed memory machines (or software-coherent clusters), executing a thread where the data resides becomes important. Scaling the multithreading implementations to clusters of SMPs would therefore require some multi-level strategy. In particular, a space-efficient scheduler from this dissertation could be deployed within a single SMP, while some scheme based on data affinity could be used across SMPs. It will be interesting to explore both analytical and experimental solutions in this direction. At the other end of the spectrum of hardware platforms, the research presented in this dissertation could be applied to the scheduling of fine-grained threads on emerging multithreaded architectures, with the goal of minimizing the size of the cache footprint.

8.3 Summary

Conserving the space usage of parallel programs is often as important as reducing their running time. I have shown that for nested-parallel programs, space-efficient, asynchronous scheduling techniques result in good space and time performance in both theory and practice. For a majority of the benchmarks, these schedulers result in lower space requirements than previous schedulers.

In practice, these scheduling techniques can also be extended to programs with non-nested parallelism, thereby making them applicable to a large class of fine-grained, parallel applications.

The main goal in developing the provably space-efficient schedulers was to allow users to write high-level parallel programs without manually mapping the work onto the processors, while still being guaranteed of good space and time performance. It is relatively easy to convince people that writing parallel programs in a high-level parallel language or model is simpler than using more popular, low-level paradigms. It is much harder to convince them that the high-level models can also perform as well; I hope this dissertation will be a step in that direction.

Appendix A

A Tighter Bound on Space Requirement

In Chapter 3 (Section 3.3) algorithm *AsyncDF* was shown to execute a parallel computation with depth D and serial space requirement S_1 on p processors using $S_1 + O(p \cdot D)$ space. In particular, when actions that allocate space are represented by heavy nodes, and each heavy node allocates at most K space (the value of the memory threshold), I proved that any prefix of the parallel computation has $O(p \cdot D)$ heavy premature nodes. Here D is the maximum number of actions along any path in the program dag. Therefore, the value of D and the number of premature nodes (and hence the space bound) depends on the definition of an action; recall that an action is a "unit" of work that may allocate or deallocate space, and requires a timestep to be executed. An action may be as small as a fraction of a machine instruction, or as large as several machine instructions, depending on the definition of a timestep and the underlying architecture. In this section, we give a more precise space bound by specifying the bound in terms of a ratio of the depth D, and the number of actions between consecutive heavy nodes. Being a ratio of two values specified in terms of actions, it is no longer dependent on the granularity of an action.

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

Lemma A.1 *Let G be a dag with W nodes, depth D and granularity g, in which every node allocates at most K space. Let s_1 be the* 1DF*-schedule for G, and s_p the parallel schedule for G executed by the* AsyncDF *algorithm on p processors. Then the number of heavy premature nodes in any prefix of s_p with respect to the corresponding prefix of s_1 is $O(p \cdot D/g)$.*

Proof: The proof is similar to the proof for Lemma 3.2. Consider an arbitrary prefix σ_p of s_p, and let σ_1 be the corresponding prefix of s_1. As with Lemma 3.2, we pick a path P from the root to the last non-premature node v to be executed in σ_p, such that for every edge (u, u') along the path, u is the last parent of u' to be executed. Let u_i be the i^{th} *heavy* node along P; let δ be the number of heavy nodes on P. Let t_i be the timestep in which u_i gets executed; let $t_{\delta+1}$ be the last timestep in σ_p. For $i = 1, \ldots, \delta$, let I_i be the interval $\{t_i + 1, \ldots, t_{i+1}\}$.

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

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

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

■

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

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

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

Appendix B

Analysis of the Parallelized Scheduler for *AsyncDF*

In this section, we prove the space and time bounds for a parallel computation executed using the parallelized scheduler for algorithm *AsyncDF*, as stated in Section 3.4 (Theorem 3.8). These bounds include the space and time overheads of the scheduler. We first define a class of dags that are more general than the dags used to represent parallel computations so far. This class of dags will be used to represent the computation that is executed by the parallelized scheduler. We then use this class of dags to prove the time and space bounds of the parallel computation including scheduler overheads.

B.1 Latency-weighted dags

We extend the definition of a program dag (as defined in Section 2.1.2) by allowing nonnegative weights on the edges; we call this new dag a ***latency-weighted dag***. Let $G = (V, E)$ be a latency-weighted dag representing a parallel computation. Each edge $(u, v) \in E$ has a nonnegative weight $l(u, v)$ which represents the ***latency*** between the actions of the nodes u and v. The ***latency-weighted length*** of a path in G is the sum of the total number of nodes in the path $plus$ the sum of the latencies on the edges along the path. We define ***latency-weighted depth*** D_l of G to be the maximum over the latency-weighted lengths of all paths in G. Since all latencies are nonnegative, $D_l \geq D$. The dag described in Section 2.1.2 is a special case of a latency-weighted dag in which the latencies on all the edges are zero. We will use non-zero latencies to model the delays caused by the parallelized scheduler.

Let $t_e(v)$ be the timestep in which a node $v \in V$ gets executed. Then v becomes ***ready*** at a timestep $i \leq t_e(v)$ such that $i = \max_{(u,v)\in E}(t_e(u) + l(u, v) + 1)$. Thus, a p-schedule $V_1, V_2, \ldots, V_T$ for a latency-weighted dag must obey the latencies on the edges, that is, $\forall(u, v) \in E$, $u \in V_j$ and $v \in V_i \Rightarrow i > j + l(u, v)$. We can now bound the time required to execute a greedy schedule for latency-weighted dags; this proof uses an approach similar to that used by [26] for dags without latencies.

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

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

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

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

Representing the parallel execution as a latency-weighted dag

This section describes how the computation executed by the parallelized scheduler can be represented as a latency-weighted dag, and then proves properties of such a dag.

As in Section 3.3.2, we call the first node in each batch of a thread a heavy node, and the remaining nodes light nodes. With the parallelized scheduler, we consider a thread (or its leading heavy node v) to become ***ready*** when all the parents of v have been executed, *and* the scheduler has made v available for scheduling. Since this may require a scheduling iteration after the parents of v have been executed and inserted into Q_{in}, the cost of this iteration imposes latencies on edges into v, resulting in a latency-weighted dag. We can now characterize the latency-weighted dag generated by the parallelized scheduler. The constant-time accesses to the queues Q_{out} and Q_{in} are represented as additional actions in this dag, while the cost of the scheduling iterations are represented by the latency-weighted edges. As with the analysis of the serial scheduler in Section 3.3, we assume for now that every action allocates at most K space (where K is the user-specified, constant memory threshold), and deal with larger allocations later.

Lemma B.2 *Consider a parallel computation with work W and depth D, in which every action allocates at most K space. Using the parallelized scheduler with αp processors acting as sched-*

ulers, the remaining $(1-\alpha)p$ worker processors execute a latency-weighted dag with $O(W)$ work, $O(D)$ depth, and a latency-weighted depth of $O(\frac{W}{\alpha p} + \frac{D \cdot \log p}{\alpha})$. Further, after the last parent of a node in the dag is executed, at most one iteration may complete before the node becomes ready.

Proof: Let G be the resulting dag executed by the worker processors. Each thread is executed non-preemptively as long as it does not terminate or suspend, and does not need to allocate more than a net of K units of memory. Each time a thread is scheduled and then preempted or suspended, a processor performs two constant-time accesses to the queues Q_{out} and Q_{in}. As shown in Figure B.1, we represent these accesses as a series of a constant number of actions (nodes) added to the thread; these nodes are added both before a heavy node (to model the delay while accessing Q_{out}) and after the series of light nodes that follow the heavy node (to model the delay while accessing Q_{in}). We will now consider the first of these added nodes to be the heavy node, instead of the real heavy node that allocates space; this gives us a conservative bound on the space requirement of the parallel computation, because we are assuming that the memory allocation has moved to an earlier time. A thread executes at least one action from the original computation every time it is scheduled. Since the original computation has W nodes, the total work performed by the worker processors is $O(W)$, that is, the resulting dag has $O(W)$ work; similarly, its depth is $O(D)$.

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

Finally, we show that G has a latency-weighted depth of $O(\frac{W}{\alpha p} + \frac{D \cdot \log p}{\alpha})$. Consider any path in G. Let l be its length. For any edge $e = (u, v)$ along the path, if u is the last parent of v, we just showed that v becomes ready by the end of at most two scheduling iterations after u is executed. Therefore the latency $l(u, v)$ is at most the duration of these 2 scheduling iterations. Let n and n' be the number of dead threads deleted by these two scheduling iterations, respectively. Then, using Lemma 3.7, $l(u, v) = O(\frac{n}{\alpha p} + \frac{n'}{\alpha p} + \frac{\log p}{\alpha})$. Because each thread is deleted by the scheduler at most once, a total of $O(W)$ deletions take place. Since any path in G has $O(D)$ edges, the latency weighted depth of the path is $O(D)$ plus the sum of the latencies on $O(D)$ edges, which is $O(\frac{W}{\alpha p} + \frac{D \cdot \log p}{\alpha})$. ∎

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

B.2 Time Bound

We can now bound the total running time of the resulting schedule, including scheduler overheads.

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

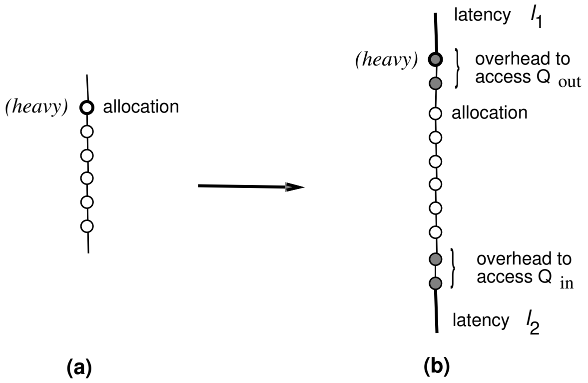


Figure B.1 : (a) A portion of the original computation dag, and (b) the corresponding portion of the latency-weighted dag as executed by the parallelized scheduler. This portion is the sequence of nodes in a thread executed non-preemptively on a processor, and therefore consists of a heavy node followed by a series of light nodes. The dag executed by the parallelized scheduler has latencies l_1 and l_2 imposed by scheduling iterations (shown as bold edges here), while the additional grey nodes represent the constant delay to access Q_{in} and Q_{out}. We consider the first of these grey nodes to be a heavy node, instead of the original heavy node that performs the real allocation.

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

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

2. $|Q_{out}| = p \cdot \log p$. Since $(1-\alpha)p$ worker processors will require at least $\frac{\log p}{(1-\alpha)}$ timesteps to execute $p \log p$ actions, none of the worker processors will be idle for the first $\frac{\log p}{(1-\alpha)}$ steps after t_i. However, if the $(i+1)^{th}$ scheduling iteration, which is currently executing, has to delete n_{i+1} dead threads, it may execute for $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ timesteps (using Lemma 3.7). Thus, in the worst case, the processors will be busy for $\frac{\log p}{(1-\alpha)}$ steps and then remain idle for another $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ steps, until the next scheduling iteration ends. We call such timesteps ***idling*** timesteps. Of the $O(\frac{n_{i+1}}{\alpha p} + \frac{\log p}{\alpha})$ idling steps, $\Theta(\frac{\log p}{\alpha})$ steps are within a factor of $\frac{c(1-\alpha)}{\alpha}$ of the preceding $\frac{\log p}{(1-\alpha)}$ steps when all worker processors were busy (for some constant c); therefore, they can add up to $O(\frac{W}{p} \cdot \frac{(1-\alpha)}{\alpha}) = O(\frac{W}{\alpha p})$. In addition, because each thread is deleted only once, at most W threads can be deleted. Therefore, if the $(i+1)^{th}$ scheduling iteration results in an additional $O(\frac{n_{i+1}}{\alpha p})$ idle steps, they add up to $O(\frac{W}{\alpha p})$ idle steps over all the scheduling iterations. Therefore, a total of $O(\frac{W}{\alpha p})$ idling steps can result due the

scheduler.

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

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

B.3 Space bound

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

From Lemma B.2, we know that for a parallel computation with depth D, the parallelized scheduler executes a dag of depth $\Theta(D)$. Therefore, using an approach similar to that of Lemma 3.2, we can prove the following bound for the parallelized scheduler.

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

Proof: The approach used in the proof is similar to that of Lemma 3.2. Let G be the latency-weighted dag representing the executed computation; according to Lemma B.2, G has a depth of $\Theta(D)$. Let σ_p be any prefix of the parallel schedule s_p generated by the parallelized scheduler, and let σ_1 be the corresponding serial prefix, that is, the longest prefix of the 1DF-schedule containing only nodes in σ_p.

Let v be any one of the last non-premature nodes executed in σ_p. Let P be the path from the root to v such that, for every edge (u, u') along the path, u is the last parent (or any one of the last parents) of u' to be executed. Let u_i be the node along P at depth i. Then u_1 is the root node, and $u_\delta = v$, where δ is the depth of node v. For $i = 1, \ldots, \delta$, let t_i be the timestep in which u_i gets executed. Let $t_0 = 0$ and let $t_{\delta+1}$ be the last timestep in which nodes from σ_p are executed. For $i = 0, \ldots, \delta$, let I_i be the interval $\{t_i + 1, \ldots, t_{i+1}\}$. Then the intervals $I_0, \ldots, I_\delta$ cover all the timesteps in which the nodes in σ_p are executed.

During interval I_0, only the root node u_1 is ready, and therefore, no premature nodes are executed during I_0. Consider any interval I_i, for $i = 1, \ldots, \delta - 1$. We will bound the number of heavy premature nodes executed in this interval. By the end of timestep t_i, the last parent of u_{i+1} has been executed. Therefore, if u_{i+1} is a light node, it will be executed in the next timestep, and the remaining worker processors may execute another $(1-\alpha)p - 1$ heavy premature nodes with it, that is, at most $(1-\alpha)p - 1$ heavy premature get executed in interval I_i.

Consider the case when u_{i+1} is a heavy node. Due to latencies on the edges, u_{i+1} may not be ready in timestep $t_i + 1$. At the start of this timestep $t_i + 1$, Q_{out} may contain at most $p \cdot \log p$ nodes, all of which may be heavy premature nodes and will be executed in the interval I_i. Let $t_r > t_i$ be the timestep in which u_{i+1} becomes ready. By Lemma B.2, at most one scheduling iteration may complete after t_i and before t_r. This iteration may add at most $p \cdot \log p$ heavy nodes to Q_{out}; all of these nodes may be premature in the worst case, and will be executed in interval I_i. Timestep t_r onwards, u_{i+1} must be added to Q_{out} before any premature nodes, since it has a lower 1DF-number. When u_{i+1} is taken off Q_{out} and executed (at timestep t_{i+1}) by a worker processor, the remaining $(1-\alpha)p - 1$ worker processors may pick premature nodes from Q_{out} to execute in the same timestep.

Therefore, a total of at most $O(p \cdot \log p)$ heavy premature nodes may get executed in any interval I_i, for $i = 1, \ldots, \delta - 1$. Similarly, because u_δ is the last non-premature node to be executed in σ_p, at most another $2p \cdot \log p$ heavy premature nodes may get executed during the interval I_δ following its execution. Therefore, since $\delta = O(D)$, summing over all the intervals $I_0, \ldots, I_\delta$, at most $(p \cdot \log p \cdot D)$ heavy premature nodes may be scheduled in any prefix of s_p. ∎

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

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

We now bound the space required for $\mathcal{R}$, along with the space required to store suspended threads. We will use the limit on the number of non-premature nodes in the any prefix of the parallel schedule (Lemma B.4) to derive this bound. Recall that $\mathcal{R}$ consists of ready threads, stubs for live threads, and dead threads. At any timestep, the number of suspended threads plus the number of ready threads and stubs in $\mathcal{R}$ equals the number of active threads in the system. Let us call a thread a ***premature thread*** at timestep j if at least one of its heavy nodes that was executed or put on Q_{out} is premature in the j-prefix σ_p of the parallel schedule. The total number of active threads at any timestep is at most the number of premature threads, plus the number of stubs (which is $O(p \cdot \log p)$), plus the number of active threads that are not premature (which is bounded by the maximum number of active threads in the 1DF-schedule). A 1DF-schedule may have at most D active threads at any timestep. Further, the number of premature threads at any timestep t is at most the number of premature nodes in the t-prefix of the schedule, which is at most $O(p \cdot \log p \cdot D)$ (using Lemma B.4). Therefore, the total number of active threads (including suspended threads) at any timestep is at most $O(p \cdot \log p \cdot D)$.

The scheduler performs lazy deletions of dead threads; therefore, the number of dead threads in $\mathcal{R}$ must also be counted. Let λ_i be the i^{th} scheduling iteration that moves at least one thread to Q_{out} (*i.e.*, $q_o \geq 1$ in Figure 3.5 for such an iteration). Consider any such iteration λ_i. Recall that this iteration must deletes at least all the dead threads up to the second ready or seed thread in $\mathcal{R}$ (step 3 in Figure 3.5). We will show that after scheduling iteration λ_i performs deletions, all remaining dead threads in $\mathcal{R}$ must be premature threads at that timestep. Let T_1 and T_2 be the first two ready (or seed) threads in $\mathcal{R}$. Since the scheduler deletes all dead threads up to T_2, there can be no more dead threads to the immediate right of T_1 that may had a higher priority than T_1, that is, T_1 now has a higher priority than all the dead threads in $\mathcal{R}$. Since all the remaining dead threads have been executed before T_1, they must be premature. Therefore, all dead threads in $\mathcal{R}$ at the end of scheduling iteration λ_i must be premature, and are therefore $O(p \cdot \log p \cdot D)$ in number (using Lemma B.4). The scheduling iterations between λ_i and λ_{i+1} do not move any threads to Q_{out}, and therefore do not create any new entries in R. They may, however, mark existing active threads as dead. Thus the number of dead threads in $\mathcal{R}$ may increase, but the total number of threads in $\mathcal{R}$ remains the same. Scheduling iteration λ_{i+1} must delete all dead threads up to the second ready thread in $\mathcal{R}$. Therefore, before it creates any new threads, the iteration reduces the number of dead threads back to $O(p \cdot \log p \cdot D)$. Thus at any time, the total space required for $\mathcal{R}$ and for the suspended threads, is $O(p \cdot \log p \cdot D)$. ∎

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

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

As in Section 3.3.2, allocations larger than K units are handled by delaying the allocation with parallel dummy threads. If S_a is the total space allocated, the number of dummy nodes added is at most S_a/K, and the depth is increased by a constant factor. Thus, using the parallelized scheduler, the final time bound of $O(W/p + S_a/pK + D \cdot \log p)$ and the space bound of $S_1 + O(K \cdot D \cdot p \cdot \log p)$ follow, as stated in Theorem 3.8. These bounds include the scheduler overheads.

Appendix C

Algorithms to Access Queues Q_{in} and Q_{out}

Figures C.1 and C.2 present the algorithms for the worker processors and the scheduler thread of algorithm *AsyncDF* (as shown in Figure 3.2) to access to the queues Q_{in} and Q_{out}. The algorithms shown here are designed for the serial scheduler; thus, only one processor at a time accesses Q_{in} and Q_{out} as the scheduler. In contrast, multiple processors may concurrently access Q_{in} and Q_{out} as workers. The algorithms can be extended for the parallelized scheduler described in Section 3.4.

According to Lemma 3.5, neither of the two queues can have more than $3p$ threads (p is the total number of processors). Therefore, both Q_{in} and Q_{out} can be implemented as arrays of length $L = 3p$ that wrap around. Threads are added to the tail and removed from the head. The head index H and the tail index T increase monotonically, and the current locations of the head and tail are accessed using modulo L arithmetic. The algorithms assume that values of indices H and T do not overflow.

Some operations may get arbitrarily delayed on any processor (*e.g.*, due to external interruptions). Therefore, to ensure correctness, I introduce auxiliary bit arrays *full*$_{in}$ and *full*$_{out}$, and an array *next_reader* of index values. These auxiliary arrays are each of length L. The bit in the i^{th} location of *full*$_{in}$ (or *full*$_{out}$) is set when the i^{th} location of the corresponding queue contains a thread. The i^{th} location of the *next_reader* array contains the value h_i of the head index H for which a worker may next read a thread from the i^{th} location of Q_{out}; according to the algorithm in Figure C.1, $h_i \equiv i \pmod{L}$. The algorithm assumes that reads and writes to each element of *next_reader* are atomic.

The auxiliary bit arrays are required to provide synchronization between the workers and the scheduler. For example, in Figure C.1, instead of using the *full*$_{out}$ array, suppose the workers were to simply use fetch-and-add( ) modulo L to obtain an index into Q_{out}, and then read a thread from the corresponding location of Q_{out}. In this case, a worker may read a thread from a location before the scheduler has written a thread to it, resulting in incorrect behavior. Similarly, before writing a thread to Q_{out}, the scheduler must check the value of the corresponding bit in *full*$_{out}$ to ensure that the location in Q_{out} is now empty. The *next_reader* array is used to order the accesses by workers while reading Q_{out}. Without this array, two (or more) workers may end up reading from the same location in Q_{out}. This can happen if one worker obtains an index and is delayed before it reads the corresponding thread from Q_{out}, while another worker subsequently obtains the same index (modulo L) into Q_{out}.

Algorithm *AsyncDF* inherently ensures that, irrespective of how long a worker or a scheduler may get delayed, there can never be more than $3p$ threads in (or being inserted into) Q_{in}. Thus, the functions defined in Figure C.2 ensure that the scheduler must read the thread written by a worker from a given location in Q_{in} before another worker attempts to write to the same location (when the tail wraps around). Therefore, the algorithm for the worker in Figure C.2 does not require checking the contents of an auxiliary array.

At the start of the execution, all the locations in the *full*$_{in}$ and *full*$_{out}$ arrays are set to FALSE ; H is initialized to 0 and T is initialized to -1. The i^{th} element of the *next_reader* array is initialized to i.

Worker	**Scheduler**
<u>*To delete a thread from Q_{out}:*</u>	<u>*To insert m threads into Q_{out}:*</u>
begin Q_{out}-removal	**begin** Q_{out}-insertion
h := fetch-and-add $(H, 1)$;	$t := T$;
while (*next_reader* $[h \bmod L] \neq h$) ;	$T := T + m$;
while (*full*$_{out}$ $[h \bmod L]$ = FALSE) ;	**for** $i := 1, \ldots, m$:
read thread at Q_{out} $[h \bmod L]$;	**while** (*full*$_{out}$ $[(t+i) \bmod L]$ = TRUE) ;
full$_{out}$ $[h \bmod L]$:= FALSE ;	write thread at Q_{out} $[(t+i) \bmod L]$;
next_reader $[h \bmod L] := h + L$;	*full*$_{out}$ $[(t+i) \bmod L]$:= TRUE ;
end Q_{out}-removal	**end** Q_{out}-insertion

Figure C.1 : Functions to access queues Q_{out}. The *next_reader* array is used to order the reads by multiple workers from any given location in Q_{out}.

Time analysis. The above algorithms to access Q_{out} and Q_{in} work correctly without assuming any bound on the time required for the various operations. However, to bound the time for each queue access (which is required to analyze the total running time and space requirement), we now assume the cost model described in Section 3.3.1. We also assume that reads and writes take constant time. Because each processor can now execute a fetch-and-add operation in constant time, the worker processors can add a thread to Q_{in} in constant time. Further, when there are a sufficient number of threads in Q_{out}, a worker can remove a thread from it in constant time. Thus, at any timestep, if Q_{out} has n threads, and p_i worker processors are idle, then $\min(n, p_i)$ of the p_i idle processors are guaranteed to succeed in picking a thread from Q_{out} within a constant number of timesteps.

Worker	**Scheduler**
To insert a thread into Q_{in}: **begin** Q_{in}-insertion t := fetch-and-add (T, 1); $t := t + 1$; write thread at $Q_{in}\,[t \bmod L]$; *full*$_{in}\,[t \bmod L]$:= TRUE ; **end** Q_{in}-insertion	*To remove all threads from Q_{in}:* **begin** Q_{in}-removal $t := T$; $n := t - H + 1$; **for** $i := 0, \ldots, n-1$: **while** (*full*$_{in}\,[H \bmod L]$ = FALSE); read the thread at $Q_{in}\,[H \bmod L]$; *full*$_{in}\,[H \bmod L]$:= FALSE ; H := H + 1; **end** Q_{in}-removal

Figure C.2: Functions to access queues Q_{in}. The workers need not check the full bit before inserting a thread, since there can never be more than L threads that are either in Q_{in} or are about to be put into Q_{in}.

Appendix D

A Parallelized Scheduler for Algorithm *DFDeques*

I first describe and analyze a synchronous, parallelized scheduler for algorithm *DFDeques*, and then briefly outline the implementation and analysis of an asynchronous version of the parallel scheduler. The parallelization described here is similar to the parallel implementation of the scheduler for algorithm *AsyncDF* presented in Section 3.4. The scheduler needs to support fast lookups (*i.e.*, finding the m^{th} deque from the left end on $\mathcal{R}'$) and must also provide delete and insert operations for deques in $\mathcal{R}'$. Therefore, both versions implement $\mathcal{R}'$ as an array, and perform insertions and deletions of deques lazily. The synchronous scheduler requires the processors to suspend execution of the parallel computation after every few timesteps, and synchronously perform update to $\mathcal{R}'$ (such as inserts and deletes). In contrast, the asynchronous version of the parallelized scheduler uses a constant fraction of the processors to continuously execute the updates to $\mathcal{R}'$, while the remaining processors asynchronously execute the parallel computation.

D.1 A Synchronous, Parallelized Scheduler

The globally ordered set of ready threads $\mathcal{R}'$ is implemented as an array; the deques in the the array are arranged from left to right in decreasing order of their thread priorities. The processors execute the code shown in Figure 6.4; they select a steal target by randomly picking one of the first (leftmost) p deques in $\mathcal{R}'$. Deques are deleted lazily; when a processor needs to delete a deque, it simply marks it as deleted. Subsequent steals to this deque fail. Deques that have been marked for deletion but have not yet been deleted from $\mathcal{R}'$ are called ***dead*** deques; all other deques are called ***live*** deques. When a processor creates a new deque, it adds the deque to a local set of deques rather than directly to $\mathcal{R}'$. These new deques do not serve as targets for steals by other processors until they are subsequently inserted into $\mathcal{R}'$. Such deques that have been created but not yet inserted into $\mathcal{R}'$ are referred to as ***nascent*** deques. Besides a pointer to a deque Δ, each entry in $\mathcal{R}'$ also contains a linked list ***nlist*** of pointers to all the nascent deques that have been created when threads were stolen from the deque Δ. The pointers in this nlist are stored in the correct priority order, that is, the order in which the steals from Δ were performed. The entry for each deque Δ in $\mathcal{R}'$ also contains an ***ncount*** value that reflects the number of nascent deques created due to steals from deque Δ. Thus, when a processor steals a thread from a deque Δ, it increments the ncount value

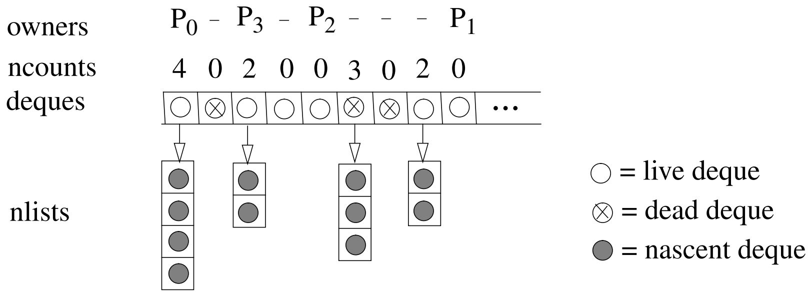


Figure D.1 : An example snapshot of $\mathcal{R}'$ during the execution of the parallelized scheduler for algorithm *DFDeques*. Dead deques are shown crossed out, while nascent deques are shown as shaded gray. The ncount values record the number of nascent deques created as a result of steals from each deque.

of the deque, and adds a pointer to the newly created deque to the nlist for Δ. Figure D.1 shows an example snapshot of $\mathcal{R}'$ at some intermediate timestep.

To amortize the cost of scheduling operations such as insertions and deletions of deques from $\mathcal{R}'$, they are performed every $\log p$ timesteps. Thus, after every $\log p$ timesteps, all the processors temporarily interrupt their current thread and execute a ***scheduling iteration*** during which they update $\mathcal{R}'$. Further, to keep processors busy between scheduling iterations, they are allowed to steal from the leftmost $p \cdot \log p$ deques in $\mathcal{R}'$ (instead of the leftmost p deques). A scheduling iteration involves deleting all the dead deques that are to the left of the first (leftmost) $p \cdot \log p$ live deques[1] in $\mathcal{R}'$. The processors finally collect all the nascent deques and insert then into the correct positions in $\mathcal{R}'$. The scheduling iteration then ends, and the processors resume execution of the code in Figure 3.2.

All elements in the nlists are allocated from a contiguous chunk of memory. Subsequently performing insertions of nascent deques into $\mathcal{R}'$ requires a list ranking operation on these nlists. List ranking on $O(p \cdot \log p)$ contiguous elements can be executed on p processors in $O(\log p)$ time. Flattening the lists for insertion into $\mathcal{R}'$ also requires a scan operation on the leftmost $p \cdot \log p$ ncount values, which can be performed in $O(\log p)$ time on p processors. Adding to or deleting from one end (the left end) of an array is straightforward[2].

As noted in Section 6.3.1, we assume that in every timestep, a processor either attempts a steal or executes some work. If the steal is successful, it begins executing the first action of the stolen thread in the same timestep; this assumption is made only to simplify the analysis. Our asymptotic bounds are not affected if a successful (or failed) steal attempt causes a constant delay.

We now prove the following lemma bounding the time required to execute a scheduling iteration.

Lemma D.1 *In the synchronous scheduler for algorithm DFDeques (K), a scheduling iteration that deletes n deques requires $O(n/p + \log p)$ timesteps to execute.*

[1] As with the parallelized *AsyncDF* scheduler in Chapter 3, other dead deques are deleted during some subsequent scheduling iteration, when they appear to the left of the leftmost $(p \cdot \log p + 1)$ live deques in $\mathcal{R}'$.

[2] Since the array $\mathcal{R}'$ may need to be occasionally shrunk (grown) by copying over all the elements to a new space, all our scheduling costs have amortized bounds.

Proof: This proof is similar to the proof of Lemma 3.7, except that operations are performed on deques rather than individual threads. Since in each scheduling iteration only the deques to the left of the the leftmost $p \cdot \log p$ live deques are deleted, the n deques to be deleted appear among the leftmost $n + p \cdot \log p$ deques in $\mathcal{R}'$. Therefore, finding and deleting these n threads from the $n + p \cdot \log p$ threads at the left end of array $\mathcal{R}'$ requires $O(n/p + \log p)$ timesteps on p processors[3]. Further, at most $p \cdot \log p$ new deques may be created between consecutive scheduling iterations, and they will need to be inserted between the leftmost $p \cdot \log p$ live deques in $\mathcal{R}'$. By performing a parallel prefix sum operation on the ncounts of the first $p \cdot \log p$ entries, and a list-ranking operation on their nlists, these insertions can be executed on p processors in $O(\log p)$ time. ∎

We can now bound the time required for the parallel computation.

Theorem D.2 *Consider a parallel computation with depth D and total work W. The expected time for algorithm* DFDeques *to executes this computation using the parallelized scheduler on p processors, is $O(W/p + D \cdot \log p)$. Further, for any $\epsilon > 0$, the execution requires $O(W/p + (D + \ln(1/\epsilon)) \cdot \log p)$ time with probability at least $1 - \epsilon$.*

Proof: I present only the expected case analysis here; the analysis of the high probability bound follows in a similar manner as that for Lemma 6.9.

We call the timesteps required to execute scheduling iterations as the ***scheduling timesteps***, and the remaining timesteps as the ***worker timesteps***. There are $\log p$ worker timesteps between consecutive scheduling iterations. Let n_i be the number of deques deleted in the i^{th} scheduling iteration. Let R_i be the number of live deques in $\mathcal{R}'$ at the end of the i^{th} scheduling iteration. We look at two types of scheduling iterations, A and B, depending on whether R_i is greater or less than $p \cdot \log p$. All timesteps at the end of a type A (B) scheduling iteration are called type A (B) timesteps.

Type A: $R_i \geq p \cdot \log p$, that is, at the end of the scheduling iteration $\mathcal{R}'$ contains at least $p \cdot \log p$ live deques. We will call each sequence of $\log p$ worker timesteps following a type A scheduling iteration a ***phase***. We will call a phase ***successful*** if $\geq (p \cdot \log p/10e)$ nodes get executed during the phase. Then, similar to the analysis for Lemma 6.9, we can compute a lower bound for the probability of success of a phase as follows.

Consider any given phase. For $1 \leq i \leq \log p$, let X_{ij} be a random variable with value 1 if the j^{th} bin gets at least one steal request in the i^{th} timestep of the phase, and 0 otherwise. The variable X_{ij} must be 1 every time the j^{th} bin gets exactly 1 request (it can be 1 at other times too). Let r_i be the number of steal requests during the i^{th} timestep of this phase. Then, the probability that the

[3]Since the array size may have to be grown and shrunk from time to time, these are amortized bounds for insertions or deletions into the array. However, since the amortized nature of the bound does not affect our analyses, for simplicity, we use them as worst-case bounds in the rest of the paper.

j^{th} bin gets exactly 1 request equals $(r_i/p \cdot \log p) \cdot (1 - 1/p \cdot \log p)^{r_i - 1}$. Therefore,

$$\begin{aligned} \mathrm{E}[X_{ij}] &\geq \frac{r_i}{p \cdot \log p} \cdot \left(1 - \frac{1}{p \cdot \log p}\right)^{r_i - 1} \\ &\geq \frac{r_i}{p \cdot \log p} \cdot \left(1 - \frac{1}{p \cdot \log p}\right)^{p \cdot \log p} \\ &\geq \frac{r_i}{p \cdot \log p} \cdot \frac{1}{e} \cdot \left(1 - \frac{1}{p \cdot \log p}\right) \\ &\geq \frac{r_i}{p \cdot \log p} \cdot \frac{1}{2e} \qquad \text{for } p > 1 \end{aligned}$$

Recall that since $p - r_i$ processors are busy, each of them must execute a node, and may own a non-empty deque. Further, in each timestep of the phase, at most p additional deques can become empty (and be marked as dead). Therefore, the i^{th} timestep of the phase must have at least $(p \log p - (i-1) \cdot p - p + r_i) = (p \log p - i \cdot p + r_i)$ non-empty deques among the leftmost $p \cdot \log p$ deques. Because every non-empty deque that gets a steal request must result in a node being executed, the expected number of nodes X_i executed in the i^{th} timestep of the given phase is given by

$$\mathrm{E}[X_i] \geq p - r_i + \frac{r_i}{(2e \cdot p \cdot \log p)} \cdot (p \log p - i \cdot p + r_i)$$

This value of X_i is minimized at $r_i = p$. Therefore, we have

$$\begin{aligned} \mathrm{E}[X_i] &\geq p - p + \frac{p}{(2e \cdot p \cdot \log p)} \cdot (p \log p - i \cdot p + p) \\ &= \frac{p \log p - i \cdot p + p}{2e \cdot \log p} \\ &= \frac{p}{2e} - \frac{(i-1) \cdot p}{2e \cdot \log p} \end{aligned}$$

Therefore,

$$\begin{aligned} \sum_{i=1}^{\log p} E[X_i] &\geq \frac{p \log p}{2 \cdot e} - \frac{p}{2e \cdot \log p} \cdot \sum_{i=1}^{\log p} (i-1) \\ &= \frac{p \log p}{2 \cdot e} - \frac{p}{2e \cdot \log p} \cdot \frac{\log p \cdot (\log p - 1)}{2} \\ &\geq \frac{p \log p}{4e} \end{aligned}$$

Thus, the expected number of nodes executed in each phase is at least $p \log p/4e$. Let Y be the random variable denoting the number of nodes executed in each phase. Then, using Markov's inequality (similar to the proof for Lemma 6.9), we get

$$\begin{aligned}
\Pr[(p \cdot \log p - Y) > p \cdot \log p \cdot (1 - 1/10e)] &< \frac{\mathrm{E}[(p \cdot \log p - Y)]}{p \cdot \log p \cdot (1 - 1/10e)} \\
&\geq \frac{p \cdot \log p \cdot (1 - 1/4e)}{p \cdot \log p \cdot (1 - 1/10e)}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Pr[Y < p \cdot \log p/10e] &< \frac{(1 - 1/4e)}{(1 - 1/10e)} \\
&< \frac{19}{20}
\end{aligned}$$

Therefore, at least $p \log p/10e$ nodes are executed in a phase, that is, the phase is successful, with probability greater than $1/20$. Because an execution can have at most $10e \cdot W/p \log p$ successful phases of type A, and each type A phase succeeds with at least constant probability, the expected number of type A phases in the entire execution is $O(W/p \log p)$. Since each phase has $\log p$ timesteps, the expected number of type A timesteps is $O(W/p)$. Further, the type A scheduling iteration itself takes $O(n_i/p + \log p)$ timesteps (using Lemma D.1). The $O(\log p)$ component of this value must be at most proportional to the type A worker timesteps. Therefore, the expected number of scheduling timesteps for all type A scheduling iterations add up to $O(W/p) + \sum_{R_i \geq p \cdot \log p} O(n_i/p)$.

Type B: $R_i < p \cdot \log p$, that is, at the end of the scheduling iteration $\mathcal{R}'$ contains $R_i < p \cdot \log p$ live deques. Since nascent deques are not added to $\mathcal{R}'$ until the end of the next scheduling iteration, the number of live deques in $\mathcal{R}'$ remains under $p \cdot \log p$ during the worker timesteps following the i^{th} scheduling iteration. Using the potential function argument of Lemma 6.9, it can be shown that the expected number of steals executed over all type B worker timesteps is $O(D \cdot p \cdot \log p)$. This bound holds even when nascent deques exist, although they are not steal targets. This is because for every nascent deque is created by a successful steal, which results in a drop in the potential function[4]. Therefore, the expected number of total type B worker timesteps is at most $O(W/p + D \cdot \log p)$. Further, for every $\log p$ of these worker timesteps, one type B scheduling iteration with $O(n_i/p + \log p)$ timesteps is executed (using Lemma D.1). Therefore, such type B scheduling iterations add up to $O(W/p + D \cdot \log p + \sum_{R_i < p \cdot \log p} O(n_i/p))$ timesteps.

Finally, at most W deques may be deleted over the entire execution, *i.e.*, $\sum_{R_i < p \cdot \log p} n_i + \sum_{R_i \geq p \cdot \log p} n_i \leq W$. Therefore, the expected number of scheduling and worker timesteps required to execute the computation is $O(W/p + D \cdot \log p)$. ∎

Since dummy threads are introduced before large allocations, a computation with work W and depth D that allocates a total S_a space requires $O(W/p + S_a/pK + D \cdot \log p)$ timesteps to execute on p processors. Finally, we prove the space bound (including scheduler space) for a computation executed with the parallelized scheduler.

[4]If ϕ is the total potential of all the deques at the end of of the i^{th} scheduling iteration, then the argument is based on showing that the total potential of all deques (including nascent deques) is at most a constant fraction of ϕ after every $\Theta(p \cdot \log p)$ steals.

Theorem D.3 *Consider a computation with D depth and a serial space requirement of S_1. The expected amount of space (including scheduler space) required to execute the computation on p processors using algorithm DFDeques with the parallelized scheduler is $S_1 + O(D \cdot p \cdot \log p)$. Further, for any $\epsilon > 0$, the space requirement is $S_1 + O(p \cdot \log p \cdot (D + \ln(1/\epsilon)))$ with probability at least $1 - \epsilon$.*

Proof: Recall that we are now allowing steals to target the leftmost $p \cdot \log p$ deques (instead of the leftmost p deques). We first provide an informal extension to the proof for Lemma 6.3. Consider any parallel prefix σ_p of the parallel schedule after the first j timesteps. As described in Section 6.3.3, we split the first j timesteps into $(\delta + 1)$ intervals, where $\delta \leq D$ is the depth of the last non-premature node to be executed in the first j timesteps. For any $1 \leq i \leq \delta$, consider any interval I_i. When node u_i (as defined in that section) is first put on a deque, in the worst case, its deque may be a nascent deque. Then the deque will belong to the nlist of some entry in $\mathcal{R}'$, rather than in $\mathcal{R}'$ itself. Therefore, it would not be a potential steal target even if it is among the highest priority $p \cdot \log p$ deques. However, the next scheduling iteration, which will put the deque in $\mathcal{R}'$, must start within the next $\log p$ timesteps; until then, p processors may perform $p \cdot \log p$ steals and therefore execute at most $p \cdot \log p$ heavy premature nodes. After the next scheduling step, however, the analysis is similar to that of Lemma 6.3. Timesteps in this case are split into phases of $\Theta(p \cdot \log p)$ (instead of $\Theta(p)$) steals. We can then show that the expected number of phases during all the $(\delta + 1)$ intervals is $O(D)$, and therefore, the expected number of steals executed (or the number of heavy premature nodes in any prefix σ_p) is $O(D \cdot p \cdot \log p)$. Thus, using an argument similar to that of Lemma 6.4, we can show that the expected space allocation of the parallel computation is $S_1 + O(D \cdot p \cdot \log p)$. The high probability bound can be proved in a manner similar to Lemma 6.3.

Next, we bound the total scheduler space, that is, the space required to store ready threads and deques in $\mathcal{R}'$, nascent deques in nlists, and the suspended threads. Each thread can be stored using a constant amount of space (this does not include stack space), and each deque requires a constant amount of space, plus space proportional to the number of threads it holds. Using Lemma 7.4, there are at most $O(D \cdot p \cdot \log p)$ active threads, which is the total number of suspended threads and ready threads in $\mathcal{R}'$. The space for the nlists is proportional to the number of nascent deques. The total number of deques (including nascent deques) is at most the number of active threads, plus the total number of dead deques.

We now bound the total number of dead deques to be $O(D \cdot p \cdot \log p)$ using induction on scheduling iterations. Again, this analysis is similar to that of Lemma B.5. At the start of the first scheduling iteration, there can be at most $p \cdot \log p$ dead deques. Let there be at most $O(D \cdot p \cdot \log p)$ dead threads at the start of the i^{th} scheduling iteration. If $\mathcal{R}'$ contains less than $p \cdot \log p$ live deques, all dead threads in $\mathcal{R}'$ are deleted by the end of the iteration. Else, let Δ be the leftmost live deque in $\mathcal{R}'$. After dead deques are deleted in this iteration, any remaining dead deques must have held nodes that are now premature, since they had lower priorities than ready nodes in Δ (or, if Δ is empty, then the node currently being executed by the processor that owns Δ). Since there are at most $O(D \cdot p \cdot \log p)$ premature nodes, the expected number of dead thread remaining after the i^{th} iteration is at most $O(D \cdot p \cdot \log p)$. Further, at most another $p \cdot \log p$ dead deques can be created until the start of the next scheduling iteration. Therefore, the expected number of dead threads at the start of the next scheduling iteration is $O(D \cdot p \cdot \log p)$.

Thus, the total scheduler space required is $O(D \cdot p \cdot \log p)$. Therefore, the expected space bound for a computation with D depth and S_1 serial space requirement using the parallelized scheduler on p processors, is $S_1 + O(D \cdot p \cdot \log p)$ on p processors, including scheduler space. ∎

D.2 An Asynchronous Version of the Scheduler

The parallelized scheduler described in Section D.1 requires all p processors to interrupt their execution and synchronize every $\log p$ timesteps. This can be fairly expensive in real parallel machines, where the processors execute asynchronously. Therefore, similar to the scheme in Chapter 3, we can instead dedicate a constant fraction αp of the processors (for $0 < \alpha < 1$) to the task of maintaining $\mathcal{R}'$; we call these processors the ***scheduling*** processors. The remaining $(1 - \alpha)p$ ***worker*** processors always asynchronously execute the parallel computation as described in Figure 6.4. As with the parallelized scheduler presented in Section 3.4, the scheduling processors synchronously execute a sequence of tasks repeatedly in a while loop; a ***scheduling iteration*** is now used to denote each iteration of this loop.

I now briefly describe the implementation and analysis of such a scheduler.

Data structures. In addition to the array $\mathcal{R}'$ of deques, the asynchronous scheduler maintains an array $\mathcal{L}$ of length $p \cdot \log p$. At the end of each scheduling iteration, the i^{th} entry in $\mathcal{L}$ points to the i^{th} live deque in $\mathcal{R}'$ (from left to right); $\mathcal{L}$ may contain null pointers if there are fewer than $p \cdot \log p$ live deques in $\mathcal{R}'$. Thus, when a worker processors needs to perform a steal from the m^{th} deque, looks up the m^{th} entry in $\mathcal{L}$. The array $\mathcal{L}$ is not strictly required, but simplifies the scheduling operations, because it allows workers to continue accessing the deques while $\mathcal{R}'$ is being updated by the scheduling processors.

Instead of allowing multiple entries in an nlist for every element (deque) in $\mathcal{R}'$, we now allow at most one entry in each nlist. Thus, the ncount value for each deque may be 0 or 1. If a deque is chosen as a steal target, and the stealer finds the ncount value to be 1 (due to some previous steal from the same deque), the steal fails. Thus, between consecutive scheduling iterations, at most one thread can be stolen from each deque. The nlist entries are emptied out (*i.e.*, inserted into $\mathcal{R}'$) and the ncount values are reset to 0 during some subsequent scheduling iteration. Limiting the number of entries in the nlist to 1 (or some constant) allows the nascent deques to be collected and inserted into $\mathcal{R}'$' without a list-ranking operation.

Scheduling iterations. In each scheduling iteration, the αp scheduling processors first delete all dead deques to the left of the leftmost $p \cdot \log p$ live deques. Then they scan the ncount entries of the leftmost $p \cdot \log p$ live deques, empty out their nlist entries and insert them in the appropriate positions in $\mathcal{R}'$. Finally, they update the entries in $\mathcal{L}$ to point to the leftmost $p \cdot \log p$ live deques in $\mathcal{R}'$. Each individual entry is updated atomically, so that the workers never see an inconsistent pointer in $\mathcal{L}$. Note that while the updates are being performed, some deque may have two entries in $\mathcal{L}$ (the old entry and the new entry) pointing to it. This does not affect out space or time bounds.

Analysis

This section briefly describes the analysis of the space and time bounds for the asynchronous version of the scheduler, without stating the proofs formally.

Space bound. Because we allow only one thread to be stolen from each deque between consecutive scheduling iterations, at most $p \cdot \log p$ threads may be stolen in that interval. Therefore, the analysis of the space bound remains the same as that for the synchronous scheduler (see Theorem D.3). The space required by the nlists is proportional to the number of deques in $\mathcal{R}'$, and the space required for $\mathcal{L}$ is $\Theta(p \cdot \log p)$. Thus, the expected space requirement for a parallel computation with depth D and a serial space requirement of S_1 on p processors is $S_1 + O(p \cdot \log p \cdot D)$.

Time bound. The time bound can be proved using arguments similar to those of Lemmas B.3 and 6.9. A scheduling iteration now does not require a list-ranking operation, but needs to update $\mathcal{L}$; this update requires $O(\log p)$ time on the αp scheduling processors. Hence, the bound on the time required to execute a scheduling iteration remains the same as that for the synchronous scheduler (see Lemma D.1). The restriction that at most one thread may be stolen from each deque between consecutive scheduling iterations does not affect the argument for the time bound in Lemma D.2. However, due to this restriction, worker processors may be idle (*i.e.*, their steal attempts will fail) for a large number of timesteps while the scheduling processors are updating $\mathcal{R}'$ and $\mathcal{L}$. Therefore, these additional idling steps must be accounted for (similar to the proof of Lemma B.3). For a dag with W nodes, the scheduler may delete a total of at most W deques. Therefore, we can show that the total number of such idling steps is $O(W/p + D \cdot \log p)$. The expected time required to execute a parallel computation with with W work and D depth (assuming all space that is allocated is touched) on p processors is therefore $O(W/p + D \cdot \log p)$.

Bibliography

[1] S.J. Aarseth, M. Henon, and R. Wielen. Numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37(2):183–187, 1974.

[2] D. Adams. *A Computation Model with Data Flow Sequencing*. PhD thesis, Stanford University, December 1968.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Annual symposium on parallel algorithms and architectures (SPAA'98)*, 1998.

[4] Arvind and K. P. Gostelow. The U-interpreter. *Computer*, 15(2), 1981.

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

[6] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[7] J. L. Baer, D. P. Bovet, and G. Estrin. Legality and other properties of graph models of computations. *Journal of the ACM*, 17(3):543–554, July 1970.

[8] Henri E. Ball and Andrew S. Tanenbaum. Distributed Programming with Shared Data. In *IEEE Conference on Computer Languages*, pages 82–91. IEEE, 1988.

[9] Hesheng Bao, Jacobo Bielak, Omar Ghattas, Loukas F. Kallivokas, David R. O'Hallaron, Jonathan 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.

[10] Hesheng Bao, Jacobo Bielak, Omar Ghattas, David R. O'Hallaron, Loukas F. Kallivokas, Jonathan Richard Shewchuk, and Jifeng Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

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

[12] Gérard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1978.

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

[14] Andrew J. Bennett. *Parallel Graph Reduction for Shared-Memory Architectures*. PhD thesis, Department of Computing, Imperial College, London, U. K., 93.

[15] B. N. Bershad, E. Lazowska, and H. Levy. PRESTO : A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[16] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall Inc., New Jersey, 1989.

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

[18] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[19] Guy Blelloch, Phil Gibbons, Yossi Matias, and Girija Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, Newport, RI, June 1997.

[20] Guy Blelloch and Girija Narlikar. A practical comparison of N-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[21] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 17–19, 1995.

[22] R. D. Blumofe, M. Frigo, C. F.oerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *IPPS: 10th International Parallel Processing Symposium*. IEEE Computer Society Press, 1996.

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

[24] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In Shafi Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Los Alamitos, CA, USA, November 1994. IEEE Computer Society Press.

[25] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[26] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 362–371, San Diego, California, 16–18 May 1993.

[27] A. Borodin and I. Munro. *Computational Complexity of Algebraic and Numeric Processes*. American Elsevier, 1975.

[28] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[29] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244–258, July 1988.

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

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

[32] F. Warren Burton. Speculative Computation, Parallelism and Functional Programming. *IEEE Transactions on Computers*, 34(12):1190–1193, December 1985.

[33] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 95–113. The MIT Press, Cambridge, Massachusetts, 1990.

[34] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 1–20, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[35] G. Cattaneo, G. di Giore, and M. Ruotolo. Another C threads library. *SIGPLAN Notices*, 27(12):81–90, December 1992.

[36] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 239–259, May 1993.

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

[38] K. Mani Chandy and Jayadev Misra. *Parallel Program Design : a Foundation*. Addison-Wesley, Reading, Mass., 1988.

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

[40] G. Cheng, M. Feng, C.E.Leiserson, K. H.Randall, , and A. F.Stark. Detecting data race in cilk programs that use locks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, Newport, Rhode Island, 1998.

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

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

[43] 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.

[44] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, School of Computer Science, February 1988.

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

[46] Digital Equipment Corporation. Guide to DECthreads, December 1997. available at `http://www.unix.digital.com/faqs/publications/pub_page/V40D_DOCS.HTM`.

[47] M. Cosnard, E. Jeannot, and L. Rougeot. Low memory cost dynamic scheduling of large coarse grain task graphs. In *IEEE International Parallel Processing Symposium (IPPS)*, Orlando, FL, 1998.

[48] D. E. Culler and G. Arvind. Resource requirements of dataflow programs. In H. J. Siegel, editor, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–151, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[49] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273. IEEE Computer Society Press, 1993.

[50] J.-L. Dekeyser and P. Marquet. Supporting irregular and dynamic computations in data parallel languages. *Lecture Notes in Computer Science*, 1132:197–219, 1996.

[51] J. B. Dennis. *First Version of a Data Flow Procedure Language*, volume 19, pages 362–376. Springer-Verlag, 1974.

[52] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In Andrei Ershov and Valery A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*. Springer, Berlin, July 1972.

[53] Digital Equipment Corporation. *Digital UNIX (Version 4.0): New and Changed Features*. Order number: AA-QTLMA-TE.

[54] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[55] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, June 1990.

[56] B. Epstein. Support for speculative computation in MultiScheme. Bachelor's thesis, Brandeis University, May 1989.

[57] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 34–46. Springer-Verlag, Berlin, DE, 1987.

[58] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, Newport, Rhode Island, June 22–25, 1997.

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

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

[61] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994.

[62] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, 1994.

[63] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, June 1997.

[64] D. P. Friedman and D. S. Wise. The impact of applicative multiprogramming on multiprocessing. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 263–272, 1976.

[65] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[66] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, 17–19 June 1998.

[67] Cong Fu and Tao Yang. Space and time efficient execution of parallel irregular computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 57–68, June18–21 1997.

[68] J. L. Gaudiot and M. D. Ercegovac. Performance analysis of dataflow computers with lvariable resolution actors. In *Proc. 4th Int. Conf. Distrib. Comput. Syst.*, pages 2–9, San Francicsco, CA, May 1984.

[69] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[70] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.

[71] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(3):416–429, March 1969.

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

[73] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[74] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[75] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 350–359, 1994.

[76] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. Parallel programming based on continuation-passing thread. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, Capri, Italy, October 1994.

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

[78] A.G. Hauptmann, R.E. Jones, K. Seymore, M.A. Siegler, S.T. Slattery, and M.J. Witbrock. Experiments in information retrieval from spoken documents. In *BNTUW-98 Proc. DARPA Workshop on Broadcast News Understanding Systems*, February 1998.

[79] Hewlett Packard. *HP-UX 11.00: An Extended Product Brief.*

[80] P. Hibbard. Parallel processing facilities. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 1–7. IRIA, 1975.

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

[82] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

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

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

[85] Jing Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[86] Y. I. Ianov. The logical schemes of algorithms. In *Problems of Cybernetics*, volume 1, pages 82–140. Pergamon Press, New York, 1960.

[87] IBM. *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*.

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

[89] D. Jayasimha and M. Loui. The communication complexity of parallel algorithms. Technical Report 629, Center for Supercomputing Research and Development , University of Illinois at Urbana-Champaign, 1987.

[90] Vijay Karamcheti, John Plevyak, and Andrew A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 37(1):21–40, August 1996.

[91] R. Karp and R. Miller. Properties of a model for parallel computation: Determinacy, termination, queuing. *SIAM Journal on Applied Mathematics*, 14:1390–1411, 1966.

[92] Richard Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proc. ACM Symposium on the Theory of Computing*, pages 290–300, Chicago, IL, May 1988.

[93] Richard M. Karp. A note on the applicaton of graph theory to digital computer programming. *Information and Control*, 3(2):179–190, June 1960.

[94] R. B. Kieburtz. A RISC architecture for symbolic computation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987.

[95] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *Proc. Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[96] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proc. Programming Language Design and Implementation*, Portland, Oregon, June 21–23, 1989.

[97] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, October 1985.

[98] R.B. Lee. *Performance Characteristics of Parallel Processor Organizations*. PhD thesis, Stanford University, May 1980.

[99] T. Leighton, M. Newman, A. G. Ranade, and E. Schwabe. Dynamic tree embeddings in butterflies and hypercubes. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 224–234, Santa Fe, NM, June 1989.

[100] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1), Jan-Feb 1978.

[101] David K. Lowenthal and Gregory R. Andrews. Shared filaments: Efficient fine-grain parallelism. Technical Report TR 93-13a, University of Arizona, January 1993.

[102] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared memory machines. Technical Report TR 96-1, University of Arizona, January 1996.

[103] Fred L. Luconi. Output functional computational structures. In *Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory*, pages 76–84, Schenectady, New York, 15–18 October 1968. IEEE.

[104] Rosalind B. Marimont. A new method of checking the consistency of precedence matrices. *Journal of the ACM*, 6(2):164–171, April 1959.

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

[106] David Martin and Gerald Estrin. Models of computations and systems—evaluation of vertex probabilities in graph models of computations. *Journal of the ACM*, 14(2):281–299, April 1967.

[107] Carolyn McCreary and Helen Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

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

[109] 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.

[110] Joseph Mohan. *Performance of Parallel Programs: Model and Analysis*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1984.

[111] Eric Mohr, David Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 1990.

[112] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[113] Frank Mueller. A library implementation of POSIX threads under Unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, January 1993.

[114] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proc. SC98: High Performance Networking and Computing*, Orlando, FL, November 1998. IEEE.

[115] Rishiyur S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 376–390, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[116] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, Sebastopol, CA, USA, 1997.

[117] 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, October 1997.

[118] R. B. Osborne. Speculative computation in Multilisp. In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems, Sendai, Japan*, pages 103–137. Springer-Verlag, Berlin, DE, 1988.

[119] D. W. Palmer, J. F. Prins, S. Chatterjee, and R. E. Faith. Piecewise execution of nested data-parallel programs. *Lecture Notes in Computer Science*, 1033, 1996.

[120] Papadimitriou, Afrati, and Papageorgiou. Scheduling DAGs to minimize time and communication. In *AWOC: Aegean Workshop on Computing: VLSI Algorithms and Architectures*. LNCS, Springer-Verlag, 1988.

[121] Christos Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pages 510–513, Chicago, IL, May 1988.

[122] Christos H. Papadimitriou and Jeffrey D. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

[123] J. D. Pehoushek and J. S. Weening. Low-cost process creation and dynamic partitioning in qlisp. In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems, Sendai, Japan*, pages 182–199. Springer-Verlag, Berlin, DE, 1988.

[124] S. L. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In K. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Languages and Architectures Europe, volume 1*, pages 193–206. Springer-Verlag, New York, NY, 1989.

[125] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP - A high-performance architecture for parallel graph reduction. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer-Verlag, Berlin, DE, 1987.

[126] James Philbin, Jan Edler, Otto J. Anshus, and Craig C. Douglas. Thread scheduling for cache locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[127] N. Pippenger. Pebbling. In *Proceedings of the Fifth IBM Symposium on Mathematical Foundations of Computer Science*, 1980.

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

[129] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, January 1991.

[130] R.T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proc. Eastern Joint Comput. Conf.*, volume 16, pages 133–138, 1959.

[131] Christopher Provenzano, et.al. An implementation of the POSIX 1003.1c thread standard, 1996. `http://www.mit.edu:8001/people/proven/pthreads.html`.

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

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

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

[135] J. R. Rodriguez. *A Graph Model for Parallel Computation*. PhD thesis, Dep. Elec. Eng., MIT, Cambridge, MA, 1967.

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

[137] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In ACM-SIGACT; ACM-SIGARCH, editor, *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, SC, July 1991. ACM Press.

[138] 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.

[139] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel progams. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26. ACM, ACM, July 1986.

[140] Vivek Sarkar and John Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *1986 ACM Symposium on Lisp and Functional Programming*, pages 202–211, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

[141] Savage and Vitter. Parallelism in space-time trade-offs. *ADVCR: Advances in Computing Research*, 4, 1987.

[142] SiliconGraphics. *Topics in IRIX Programming (IRIX 6.4)*. Part No.: 007-2478-004.

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

[144] A. Singh. Pegasus: A controllable, interactive, workload generator for multiprocessors. Master's thesis, Computer Science Department, Carnegie-Mellon University, December 1981.

[145] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, July 1994.

[146] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[147] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[148] B. T. Smith. Parallel computing forum (PCF) Fortran. In *Aspects of Computation on Asynchronous Parallel Processors. Proc. IFIP WG 2.5 Working Conference.* North-Holland, August 1988.

[149] Vason P. Srini and Jorge F. Asenjo. Analysis of Cray-1S architecture. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 194–206, Stockholm, Sweden, June 13–17, 1983.

[150] Dan Stein and Devang Shah. Implementing lightweight threads. In *Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition*, pages 1–10, San Antonio, TX, 1992. USENIX.

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

[152] W. D. Strecker. *Analysis of the Instruction Execution Rate in Certain Computer Structures*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1970.

[153] Sun Microsystems. *Multithreaded Programming Guide (Solaris 2.5)*.

[154] C. D. Thompson. A complexity theory for VLSI. Technical Report CMU-CS-80-132, Carnegie-Mellon University, 1980.

[155] J. L. Traff. *Distributed and parallel graph algorithms : Models and experiments*. PhD thesis, Department of Computer Science, University of Copenhagen, 1995.

[156] Andy Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Operating Systems Review*, 23(5):159–66, 1989.

[157] David A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

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

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

[160] H. Venkateswaran and Martin Tompa. A new pebble game that characterizes parallel complexity classes. *SIAM Journal on Computing*, 18(3):533–549, June 1989.

[161] B. Weissman. Performance counters and state sharing annotations: A unified approach to thread locality. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 262–273, October 1998.

[162] B. Weissman and B. Gomes. Enabling finegrained parallelism in object-oriented languages. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 1998.

[163] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[164] Xingbin Zhang and Andrew A. Chien. Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 37–47, New York, June18–21 1997. ACM Press.