Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 7

Fork-Join Parallelism with a Data-Structures Focus

Dan Grossman

University of Washington

djg@cs.washington.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: `http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 7

FORK-JOIN PARALLELISM WITH A DATA-STRUCTURES FOCUS

Dan Grossman

University of Washington

djg@cs.washington.edu

ABSTRACT

This chapter is an introduction to parallel programming designed for use in a course on data structures and algorithms, although some of the less advanced material can be used in a second programming course. It assumes no background beyond sequential programming and a familiarity with common data structures (e.g., binary trees), algorithms (e.g., efficient sorting algorithms), and basic asymptotic analysis. The focus is on shared memory and fork-join parallelism, using Java and its ForkJoin framework for programming examples. After introducing the idea of multithreaded execution and a distinction between parallelism and concurrency control, the chapter explains how threads can communicate via shared memory. It then discusses in depth how to use divide-and-conquer recursive parallelism to perform a basic reduction, namely summing the elements of an array. It then generalizes this discussion to other reductions as well as map operations and parallel operations over other data structures. It then introduces how to analyze parallel algorithms in terms of work and span, and it introduces Amdahl's Law for analyzing the effect of parallelizing part of a computation. The last section of the chapter introduces three more sophisticated parallel algorithms — parallel-prefix sum, parallel quicksort (including a parallel partition), and parallel mergesort (including a parallel merge) — as examples of non-obvious parallel algorithms and how algorithms can be built on top of other parallel computations.

Relevant core courses: Data Structures and Algorithms, Second Programming Course in the Introductory Sequence

Relevant PDC topics: Shared memory, Language extensions, Libraries, Task/thread spawning, Load balancing, Performance metrics, Speedup, Amdahl's Law, Asymptotics, Time, Speedup, BSP/CILK, Dependencies, Task graphs, Work, (Make)span, Divide & conquer (parallel aspects), Recursion (parallel aspects), Scan (parallel-prefix), Reduction (map-reduce), Sorting, Why and what is parallel/distributed computing, Concurrency

Learning outcomes: Students mastering the material in this chapter should be able to:

- Write small parallel programs in terms of explicit threads that communicate via shared memory and synchronize via fork and join.
- Distinguish parallelism from concurrency control.
- Identify computations that are reductions or maps that can be performed in parallel.
- Analyze a fork-join computation in terms of work and span in general, and in particular be able to explain the linear work and logarithmic span of reductions and maps in the fork-join model.
- Define Amdahl's Law and use it to reason about the performance benefits of parallelism.
- Reason about non-trivial parallel algorithms in an analogous way to how students in a conventional data structures and algorithms course can reason about sequential algorithms.

Context for use: This chapter and the next one complement each other. Both are designed to be used in an intermediate-advanced data-structures courses — the course that covers asymptotic complexity, balanced trees, hash tables, graph algorithms, sorting, etc., though some of the material has also been used in second programming courses. This

chapter introduces a distinction between *parallelism* and *concurrency control (e.g., synchronization)* and then focuses on the former predominantly via *fork-join parallelism*. The next chapter focuses on concurrency control, predominantly via *mutual-exclusion locks*.

7.1 Meta-Introduction: An Instructor's View of This Material

7.1.1 Where This Material Fits in a Changing Curriculum

This chapter and the next one together introduce parallelism and concurrency control in the context of a data-structures course.

The case for teaching this material in data structures:

There may be multiple natural places to introduce this material, but "sophomore-level" data structures (after CS2 and discrete math, but before "senior-level" algorithms) works very well. Here are some reasons:

- There is room: This chapter's author made room by removing three weeks on skew heaps, leftist heaps, binomial queues, splay trees, disjoint-set, and network flow. Some of the trimming was painful and can be compensated for in senior-level algorithms, but all this material seems relatively less important. There was still plenty of room for essential data structures and related concepts such as asymptotic analysis, (simple) amortization, graphs, and sorting.

- Fork-join parallel algorithms are amenable to asymptotic analysis in terms of "work" and "span" over dags — all concepts that fit very naturally in the course. Amdahl's Law is fundamentally an asymptotic argument too.

- Ideas from sequential algorithms already in the course reappear with parallelism. For example, just as constant factors compel efficient quicksort implementations to switch to an $O(n^2)$ sort for small n, constant factors compel efficient parallel algorithms to switch to sequential algorithms for small problem sizes. Parallel sorting algorithms are also good examples of non-trivial parallelization.

- Many canonical examples for concurrent programming involve basic data structures: bounded buffers for condition variables, dictionaries for reader/writer locks, parallel

unstructured graph traversals, etc. Making a data structure "thread-safe" is an ideal way to think about what it means to be "thread-safe."

- We already used Java in the course. Java 7's ForkJoin framework is excellent for teaching fork-join parallelism. Java's built-in support for threads, locks, and condition variables is sufficient for teaching synchronization.

On the other hand, instructors wishing to introduce message passing or distributed computation will have to consider whether it makes sense in this course. This chapter and the next one focus on shared memory, only mentioning other models. This is not to claim that shared memory is "better," only that it is an important model and a good one to start with pedagogically. There is also little emphasis on asynchrony and masking I/O latency. Such topics are probably better covered in a course on systems programming, though one could add them to the material in the next chapter.

While most of the material in this chapter and the next one is not specific to a particular programming language, all examples and discussions use Java when a specific language is warranted. A C++ version of an earlier version of these materials is also available thanks to Steve Wolfman from the University of British Columbia. Porting to additional languages should be quite doable, and the author would be delighted to collaborate with people interested in doing so.

For more information on the motivation and context, see a SIGCSE2012 paper coauthored with Ruth E. Anderson [1].

7.1.2 Six Theses On A Successful Approach to this Material

In summary, this approach rests on several theses for how to teach this material:

1. Integrate it into a data-structures course.

2. Distinguish parallelism (using extra computational units to do more work per unit time) from concurrency control (managing access to shared resources). Teach parallelism first because it is easier and helps establish a non-sequential mindset.

3. Teach in a high-level language, using a library for fork-join parallelism. Teach how to use parallelism, threads, locks, etc. Do not teach how to implement them.

4. Conversely, do not teach in terms of higher-order parallel patterns like maps and reduces. Mention these, but have students actually do the divide-and-conquer underlying these patterns.

5. Assume shared memory since one programming model is difficult enough.

6. Given the limited time and student background, do not focus on memory-hierarchy issues (e.g., caching), much like these issues are mentioned (e.g., with B-trees) but rarely central in data-structures courses. (Adding a discussion should prove straightforward.)

7.1.3 How to Use These Materials — And Improve Them

These materials were originally written for CSE332 at the University of Washington (`http://courses.cs.washington.edu/courses/cse332/`). They account for 3 weeks of a required 10-week course (the University uses a quarter system). There are also PowerPoint slides, homework assignments, and a programming project. In fact, this written text was the last aspect to be written — the first edition of the course succeeded without written material and students reported parallelism to be their favorite aspect of the course.

The current home for all the materials is `http://homes.cs.washington.edu/~djg/teachingMaterials`. Feedback, typos, and suggestions for improvement are most welcome.

7.1.4 Acknowledgments (For This Chapter and the Next)

I deserve no credit for the material in this chapter and the next. If anything, my role was simply to distill decades of wisdom from others down to three weeks of core concepts and integrate the result into a data-structures course. When in doubt, I stuck with the basic and simplest topics and examples.

I was particularly influenced by Guy Blelloch and Charles Leiserson in terms of teaching parallelism before synchronization and emphasizing divide-and-conquer algorithms that do

not consider the number of processors. Doug Lea and other developers of Java's ForkJoin framework provided a wonderful library that, with some hand-holding, is usable by sophomores. Larry Snyder was also an excellent resource for parallel algorithms.

The treatment of shared-memory synchronization is heavily influenced by decades of operating-systems courses, but with the distinction of ignoring all issues of scheduling and synchronization implementation. Moreover, the emphasis on the need to avoid data races in high-level languages is frustratingly under-appreciated despite the noble work of memory-model experts such as Sarita Adve, Hans Boehm, and Bill Pugh.

Feedback from Ruth Anderson, Kim Bruce, Kristian Lieberg, Tyler Robison, Cody Schroeder, and Martin Tompa helped improve explanations and remove typos. Tyler and Martin deserve particular mention for using these notes when they were very new. James Fogarty made many useful improvements to the presentation slides that accompany this material. Steve Wolfman created a C++ version of the material.

Nicholas Shahan created almost all the images and diagrams in these notes, which make the accompanying explanations much better.

I have had enlightening and enjoyable discussions on "how to teach this stuff" with too many researchers and educators over the last few years to list them all, but I am grateful.

This work was funded in part via grants from the U.S. National Science Foundation and generous support, financial and otherwise, from Intel Labs University Collaborations.

7.2 Introduction

7.2.1 More Than One Thing At Once

In *sequential programming*, one thing happens at a time. Sequential programming is what most people learn first and how most programs are written. Probably every program you have written in Java (or a similar language) is sequential: Execution starts at the beginning of `main` and proceeds one assignment / call / return / arithmetic operation at a time.

Removing the one-thing-at-a-time assumption complicates writing software. The multiple *threads of execution* (things performing computations) will somehow need to coordinate so that they can work together to complete a task — or at least not get in each other's way while they are doing separate things. This chapter and the next cover basic concepts related to *multithreaded programming*, i.e., programs where there are multiple threads of execution. We will cover:

- How to create multiple threads
- How to write and analyze divide-and-conquer algorithms that use threads to produce results more quickly (this chapter)
- How to coordinate access to shared objects so that multiple threads using the same data do not produce the wrong answer (next chapter)

A useful analogy is with cooking. A sequential program is like having one cook who does each step of a recipe in order, finishing one step before starting the next. Often there are multiple steps that could be done at the same time — if you had more cooks. But having more cooks requires extra coordination. One cook may have to wait for another cook to finish something. And there are limited resources: If you have only one oven, two cooks cannot bake casseroles at different temperatures at the same time. In short, multiple cooks present efficiency opportunities, but also significantly complicate the process of producing a meal.

Because multithreaded programming is so much more difficult, it is best to avoid it if you can. For most of computing's history, most programmers wrote only sequential programs. Notable exceptions were:

- Programmers writing programs to solve such computationally large problems that it would take years or centuries for one computer to finish. So they would use multiple computers together.

- Programmers writing systems like an operating system where a key point of the system is to handle multiple things happening at once. For example, you can have more than one program running at a time. If you have only one processor, only one program can *actually* run at a time, but the operating system still uses threads to keep track of all the running programs and let them take turns. If the taking turns happens fast enough (e.g., 10 milliseconds), humans fall for the illusion of simultaneous execution. This is called *time-slicing*.

Sequential programmers were lucky: since every 2 years or so computers got roughly twice as fast, most programs would get exponentially faster over time without any extra effort.

Around 2005, computers stopped getting twice as fast every 2 years. To understand why requires a course in computer architecture. In brief, increasing the clock rate (very roughly and technically inaccurately speaking, how quickly instructions execute) became infeasible without generating too much heat. Also, the relative cost of memory accesses can become too high for faster processors to help.

Nonetheless, chip manufacturers still plan to make exponentially more powerful chips. Instead of one processor running faster, they will have more processors. The next computer you buy will likely have at least 4 processors (also called *cores*) on the same chip and the number of available cores will likely double every few years.

What would 256 cores be good for? Well, you can run multiple programs at once — for real, not just with time-slicing. But for an individual program to run any faster than with one core, it will need to do more than one thing at once. This is the reason that multithreaded programming is becoming more important. To be clear, *multithreaded programming is not new. It has existed for decades and all the key concepts are just as old.* Before there were multiple cores on one chip, you could use multiple chips and/or use time-slicing on one chip — and both remain important techniques today. The move to multiple cores on one chip is "just" having the effect of making multithreading something that more and more software wants to do.

7.2.2 Parallelism vs. Concurrency Control

This chapter and the next are organized around a fundamental distinction between *parallelism* and *concurrency control*. Unfortunately, the way we define these terms is not entirely standard (more common in some communities than in others), so you should not assume that everyone uses these terms precisely as we will. Nonetheless, most computer scientists agree that this distinction is important — we just disagree over the use of these English words.

Parallelism is about using additional computational resources to produce an answer faster.

As a canonical example, consider the trivial problem of summing up all the numbers in an array. We know no sequential algorithm can do better than $\Theta(n)$ time.[1]

Suppose instead we had 4 processors. Then hopefully we could produce the result roughly 4 times faster by having each processor add 1/4 of the elements and then we could just add these 4 partial results together with 3 more additions. $\Theta(n/4)$ is still $\Theta(n)$, but constant factors can matter. Moreover, when designing and analyzing a *parallel algorithm*, we should leave the number of processors as a variable, call it P. Perhaps we can sum the elements of an array in time $O(n/P)$ given P processors. As we will see, in fact the best bound under the assumptions we will make is $O(\log n + n/P)$.

In terms of our cooking analogy, parallelism is about using extra cooks (or utensils or pans or whatever) to get a large meal finished in less time. If you have a huge number of potatoes to slice, having more knives and people is really helpful, but at some point adding more people stops helping because of all the communicating and coordinating you have to do: it is faster for me to slice one potato by myself than to slice it into fourths, give it to four other people, and collect the results.

[1]For those not familiar with "big-Θ notation," you can always read Θ as O, the more common "big-O notation." For example, if an algorithm is $\Theta(n)$, then it is also $O(n)$. The difference is that O provides only an upper-bound, so, for example, a $O(n^2)$ algorithm is *also* a $O(n^3)$ algorithm and *might or might not* be a $O(n)$ algorithm. By definition, Θ is an upper and lower bound: a $\Theta(n^2)$ algorithm is neither $\Theta(n^3)$ nor $\Theta(n)$. To provide more precise claims, this chapter uses Θ where appropriate.

Concurrency control is about correctly and efficiently controlling access by multiple threads to shared resources.

Suppose we have a dictionary implemented as a hashtable with operations `insert`, `lookup`, and `delete`. Further suppose that inserting an item already in the table should update the key to map to the newly inserted value. Implementing this data structure for sequential programs is something we assume you could already do correctly. Now consider what might happen if different threads use the *same* hashtable, potentially at the same time. Two threads might even try to `insert` the same key at the same time. What could happen? You would have to look at your sequential code carefully, but it is entirely possible that the same key might end up in the table twice. That is a problem since a subsequent `delete` with that key might remove only one of them, leaving the key in the dictionary.

To prevent problems like this, concurrent programs use *synchronization primitives* to prevent multiple threads from *interleaving their operations* in a way that leads to incorrect results. Perhaps a simple solution in our hashtable example is to make sure only one thread uses the table at a time, finishing an operation before another thread starts. But if the table is large, this is unnecessarily inefficient most of the time if the threads are probably accessing different parts of the table.

In terms of cooking, the shared resources could be something like an oven. It is important not to put a casserole in the oven unless the oven is empty. If the oven is not empty, we could keep checking until it is empty. In Java, you might naively write:

```
while(true) {
   if(ovenIsEmpty()) {
      putCasseroleInOven();
      break;
   }
}
```

Unfortunately, code like this is broken if two threads run it at the same time, which is the primary complication in concurrent programming. They might both see an empty oven and

then both put a casserole in. We will need to learn ways to check the oven and put a casserole in without any other thread doing something with the oven in the meantime.

Confusion Over the Terms Parallelism and Concurrency

An exciting aspect of multithreading is that it sits at the intersection of many traditional areas of computer science: algorithms, computer architecture, distributed computing, programming languages, and systems. However, different traditions have used various terms in incompatible ways, including the words *parallelism* and *concurrency* used to describe the main contrast between this chapter and the next.

As we have used them, parallelism is anything about gaining efficiency by doing multiple things at once while concurrency is about all the issues that arise when different threads need to coordinate, share, and synchronize. This use of the terms is more common in the computer architecture and systems communities, particularly in recent years.

A more traditional use of the words, especially in the algorithms tradition, uses *parallelism* to describe when things *actually* happen at once and *concurrency* to describe when things *may* happen at once (e.g., if enough resources are available). This use of the terms is making a useful distinction within the realm of parallelism-for-gaining-efficiency but leaves us without a one-word term for the collection of issues and techniques in the next chapter.

In an attempt to make everyone happy (and with the risk of making everyone unhappy), this chapter and the next will try to strike a balance. We will use *parallelism* for anything related to gaining efficiency by doing multiple things at once. For example, a parallel algorithm will describe what may happen at the same time, without necessarily considering how many threads can actually run at once. But we will avoid using "concurrency" to describe coordination, sharing, and synchronization. Instead, we will use the term *concurrency control* when we need a single term for controlling/limiting how multiple threads may interact, and we will use more specific terms related to coordination and synchronization when possible.

Comparing Parallelism and Concurrency Control

We have emphasized here how parallelism and concurrency control are different. Is the

problem one of using extra resources effectively or is the problem one of preventing a bad interleaving of operations from different threads? It is all-too-common for a conversation to become muddled because one person is thinking about parallelism while the other is thinking about concurrency control.

In practice, the distinction between the two is not absolute. Many programs have aspects of each: how can we exploit parallelism while controlling access to shared resources? Suppose you had a huge array of values you wanted to insert into a hash table. From the perspective of dividing up the insertions among multiple threads, this is about parallelism. From the perspective of coordinating access to the hash table, this is about concurrency control. Also, parallelism does typically need some coordination: even when adding up integers in an array we need to know when the different threads are done with their chunk of the work.

We believe parallelism is an easier concept to start with than concurrency control. You probably found it easier to understand how to use parallelism to add up array elements than to understand why the while-loop for checking the oven was wrong. (And if you still don't understand the latter, don't worry, the next chapter will explain similar examples line-by-line.) So we will start with parallelism in this chapter, getting comfortable with multiple things happening at once. Then the next chapter will switch our focus to shared resources (using memory instead of ovens), learn many of the subtle problems that arise, and present programming guidelines to avoid them.

7.2.3 Basic Threads and Shared Memory

Before writing any multithreaded programs, we need some way to *make multiple things happen at once* and some way for those different things to *communicate*. Put another way, your computer may have multiple cores, but all the Java constructs you know are for sequential programs, which do only one thing at once. Before showing any Java specifics, we need to explain the *programming model*.

The model we will assume is *explicit threads* with *shared memory*. A *thread* is itself like a running sequential program, but one thread can create other threads that are part of the

same program and those threads can create more threads, etc. Two or more threads can communicate by writing and reading fields of the same objects. In other words, they share memory. This is only one model of parallel programming, but it is the only one we will use in this chapter. The next section briefly mentions other models that a full course on parallel programming would likely cover.

Conceptually, all the threads that have been started but not yet terminated are "running at once" in a program. In practice, they may not all be running at any particular moment:

- There may be more threads than processors. It is up to the Java implementation, with help from the underlying operating system, to find a way to let the threads "take turns" using the available processors. This is called *scheduling* and is a major topic in operating systems. All we need to know is that it is not under the Java programmer's control: you create the threads and the system schedules them.

- A thread may be waiting for something to happen before it continues. For example, the next section discusses the `join` primitive where one thread does not continue until another thread has terminated.

Let's be more concrete about what a thread is and how threads communicate. It is helpful to start by enumerating the key pieces that a *sequential* program has *while it is running* (see also Figure 7.1):

1. One *call stack*, where each *stack frame* holds the local variables for a method call that has started but not yet finished. Calling a method pushes a new frame and returning from a method pops a frame. Call stacks are why recursion is not "magic."

2. One *program counter*. This is just a low-level name for keeping track of what statement is currently executing. In a sequential program, there is exactly one such statement.

3. Static fields of classes.

4. Objects. An object is created by calling `new`, which returns a reference to the new object. We call the memory that holds all the objects the *heap*. This use of the word

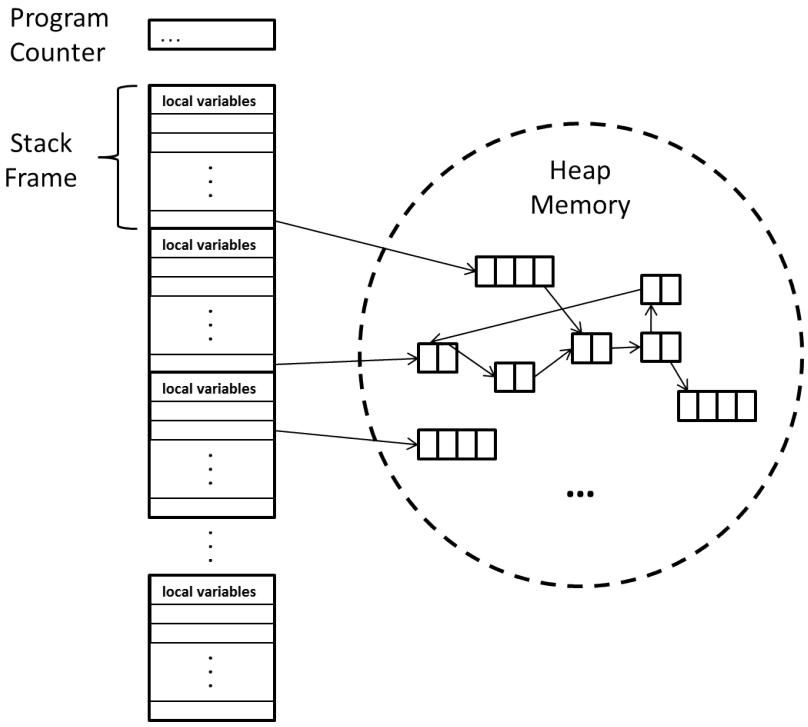


Figure (7.1) The key pieces of an executing sequential program: A program counter, a call stack, and a heap of objects. (There are also static fields of classes, which we can think of as being in the heap.)

"heap" has nothing to do with heap data structure used to implement priority queues.

It is separate memory from the memory used for the call stack and static fields.

With this overview of the sequential *program state*, it is much easier to understand threads:

Each thread has its own call stack and program counter, but all the threads share one collection of static fields and objects. (See also Figure 7.2.)

- When a new thread starts running, it will have its own new call stack. It will have one frame on it, which is *like* that thread's `main`, but it won't actually be `main`.
- When a thread returns from its first method, it terminates.

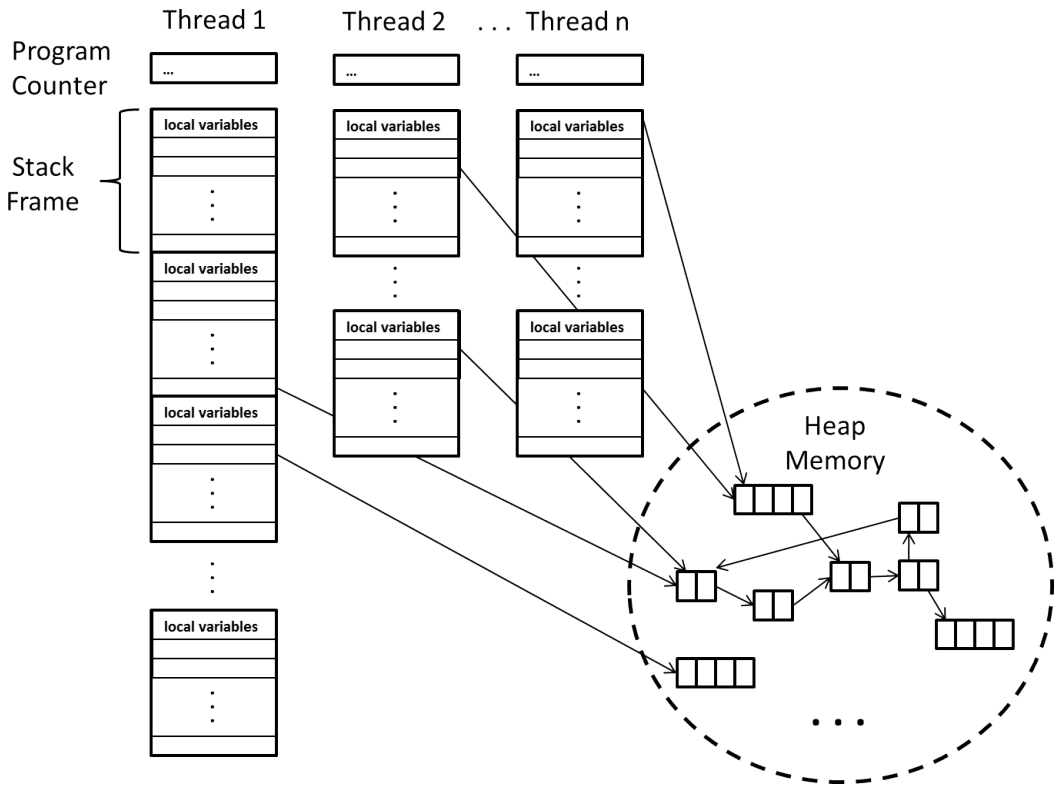


Figure (7.2) The key pieces of an executing multithreaded program: Each thread has its own program counter and call-stack, but objects in the heap may be shared by multiple threads. (Static fields of classes are also shared by all threads.)

- Each thread has its own program counter and local variables, so there is no "interference" from other threads for these things. The way loops, calls, assignments to variables, exceptions, etc. work for each thread is just like you learned in sequential programming and is separate for each thread.

- What is different is how static fields and objects work. In sequential programming we know `x.f = 42; y = x.f;` always assigns `42` to the variable `y`. But now the object that `x` refers to might also have its `f` field written to by other threads, so we cannot be so sure.

In practice, even though all objects *could* be shared among threads, most are not. In fact, just as having static fields is often poor style, having lots of objects shared among threads is often poor style. But we need *some* shared objects because that is how threads communicate. If we are going to create parallel algorithms where helper threads run in parallel to compute partial answers, they need some way to communicate those partial answers back to the "main" thread. The way we will do it is to have the helper threads write to some object fields that the main thread later reads.

We finish this section with some Java specifics for exactly how to create a new thread in Java. The details vary in different languages and in fact most of this chapter uses a different Java library with slightly different specifics. In addition to creating threads, we will need other language constructs for coordinating them. For example, for one thread to read the result another thread wrote as its answer, the reader often needs to know the writer is done. We will present such primitives as we need them.

To create a new thread in Java requires that you define a new class (step 1) and then perform two actions at run-time (steps 2–3):

1. Define a subclass of `java.lang.Thread` and override the `public` method `run`, which takes no arguments and has return type `void`. The `run` method will act like "main" for threads created using this class. It must take no arguments, but the example below shows how to work around this inconvenience.

2. Create an instance of the class you defined in step 1. That is, if you defined class `ExampleThread`, then use `new` to create an `ExampleThread` object. Note this does *not* yet create a running thread. It just creates an object of class `ExampleThread`, which is a subclass of `Thread`.

3. Call the `start` method of the object you created in step 2. This step does the "magic" creation of a new thread. That new thread will execute the `run` method of the object. Notice that you do *not* call `run`; that would just be an ordinary method call. You call `start`, which makes a new thread that runs `run`. The call to `start` "returns immediately" so the caller continues on, in parallel with the newly-created thread running `run`. The new thread terminates when its `run` method completes.

Here is a complete example of a useless Java program that starts with one thread and then creates 20 more threads:

```
class ExampleThread extends java.lang.Thread {
   int i;
   ExampleThread(int i) { this.i = i; }
   public void run() {
      System.out.println("Thread " + i + " says hi");
      System.out.println("Thread " + i + " says bye");
   }
}
class M {
   public static void main(String[] args) {
     for(int i=1; i <= 20; ++i) {
       ExampleThread t = new ExampleThread(i);
       t.start();
     }
   }
```

```
}
```

When this program runs, it will print 40 lines of output, one of which is:

```
Thread 13 says hi
```

Interestingly, we cannot predict the order for these 40 lines of output. In fact, if you run the program multiple times, you will probably see the output appear in different orders on different runs. After all, each of the 21 separate threads running "at the same time" (conceptually, since your machine may not have 21 processors available for the program) can run in an unpredictable order. The main thread is the first thread and then it creates 20 others. The main thread always creates the other threads in the same order, but it is up to the Java implementation to let all the threads "take turns" using the available processors. There is no guarantee that threads created earlier run earlier. Therefore, multithreaded programs are often *nondeterministic*, meaning their output can change even if the input does not. This is a main reason that multithreaded programs are more difficult to test and debug. Figure 7.3 shows two possible orders of execution, but there are many, many more.

So is any possible ordering of the 40 output lines possible? No. Each thread still runs sequentially. So we will always see `Thread 13 says hi` *before* the line `Thread 13 says bye` even though there may be other lines in-between. We might also wonder if two lines of output would ever be mixed, something like:

```
Thread 13 Thread says 14 says hi hi
```

This is really a question of how the `System.out.println` method handles concurrency and the answer happens to be that it will always keep a line of output together, so this would not occur. In general, concurrency introduces new questions about how code should and does behave.

We can also see how the example worked around the rule that `run` must override `java.lang.Thread`'s `run` method and therefore not take any arguments. The standard idiom is to pass any "arguments" for the new thread to the *constructor*, which then stores

Example Run 1:

```
Thread 1 says hi
Thread 3 says hi
Thread 2 says hi
Thread 1 says bye
Thread 5 says hi
Thread 3 says bye
Thread 7 says hi
Thread 7 says bye
Thread 5 says bye
Thread 6 says hi
Thread 6 says bye
Thread 9 says hi
Thread 9 says bye
Thread 4 says hi
Thread 2 says bye
Thread 8 says hi
Thread 18 says hi
Thread 17 says hi
Thread 15 says hi
Thread 14 says hi
Thread 14 says bye
Thread 11 says hi
Thread 11 says bye
Thread 12 says hi
Thread 12 says bye
Thread 19 says hi
Thread 4 says bye
Thread 10 says hi
Thread 10 says bye
Thread 19 says bye
Thread 15 says bye
Thread 13 says hi
Thread 13 says bye
Thread 17 says bye
Thread 16 says hi
Thread 18 says bye
Thread 8 says bye
Thread 16 says bye
Thread 20 says hi
Thread 20 says bye
```

Example Run 2:

```
Thread 1 says hi
Thread 3 says hi
Thread 3 says bye
Thread 2 says hi
Thread 6 says hi
Thread 4 says hi
Thread 6 says bye
Thread 1 says bye
Thread 2 says bye
Thread 5 says hi
Thread 4 says bye
Thread 9 says hi
Thread 5 says bye
Thread 8 says hi
Thread 7 says hi
Thread 8 says bye
Thread 9 says bye
Thread 11 says hi
Thread 11 says bye
Thread 13 says hi
Thread 7 says bye
Thread 13 says bye
Thread 12 says hi
Thread 10 says hi
Thread 12 says bye
Thread 14 says hi
Thread 15 says hi
Thread 10 says bye
Thread 15 says bye
Thread 16 says hi
Thread 14 says bye
Thread 19 says hi
Thread 18 says hi
Thread 20 says hi
Thread 16 says bye
Thread 17 says hi
Thread 20 says bye
Thread 18 says bye
Thread 19 says bye
Thread 17 says bye
```

Figure (7.3) Two possible outputs from running the program that creates 20 threads that each print two lines.

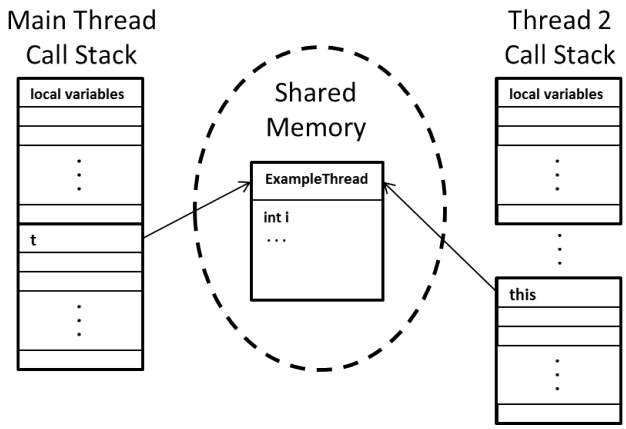


Figure (7.4) How we can use shared memory, specifically fields in a subclass of `Thread`, to pass data to a newly created thread.

them in fields so that `run` can later access them. In this simple example, this trick is the only use of shared memory since the helper threads (the ones doing the printing) do not need to pass any other data to/from the main thread or each other.

It may not look like this is using shared memory, but it is: When the main thread calls `new ExampleThread(i)`, this is a normal call to a constructor. A new object gets created and the main thread runs the constructor code, writing to the `i` field of the new object. Later, after the "magic" call to `start`, the new running thread is running the `run` method with `this` bound to the same object that the main thread wrote to. The helper thread then reads `i`, which just means `this.i`, and gets the value previously written by the main thread: shared-memory communication. See Figure 7.4.

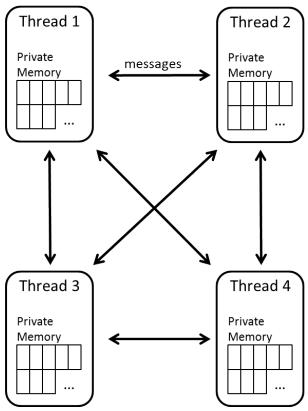
As this example shows, shared-memory communication relies on aliasing. The `t` on some iteration of the for-loop refers to the same object as `this` in one of the new threads' `run` method. Aliasing is often difficult to reason about, but it is the way we communicate in shared-memory multithreaded programs.

7.2.4 Other Models

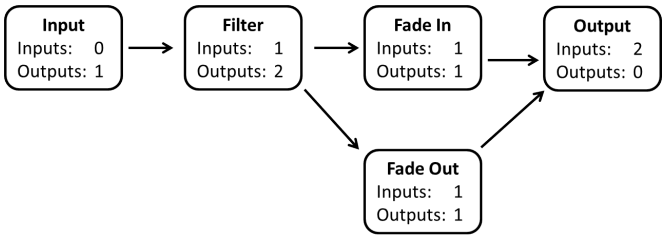
While this chapter and the next focus on using threads and shared memory to create parallelism, it would be misleading to suggest that this is the only model for parallel programming. Shared memory is often considered *convenient* because communication uses "regular" reads and writes of fields to objects. However, it is also considered *error-prone* because communication is implicit; it requires deep understanding of the code/documentation to know which memory accesses are doing inter-thread communication and which are not. The definition of shared-memory programs is also much more subtle than many programmers think because of issues regarding *data races*, as discussed in the next chapter (Section **??**).

Here are three well-known, popular alternatives to shared memory. As is common in computer science, no option is "clearly better." Different models are best-suited to different problems, and any model can be abused to produce incorrect or unnecessarily complicated software. One can also build abstractions using one model on top of another model, or use multiple models in the same program. These are really different perspectives on how to describe parallel/concurrent programs.

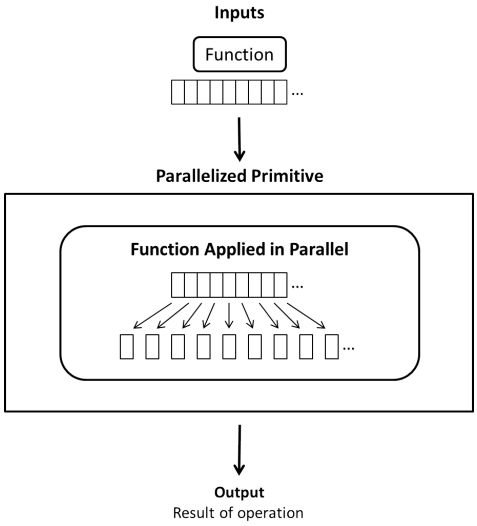
Message-passing is the natural alternative to shared memory. In this model, we have explicit threads, but they do not share objects. To communicate, there is a separate notion of a *message*, which sends a *copy* of some data to its recipient. Since each thread has its own objects, we do not have to worry about other threads wrongly updating fields. But we do have to keep track of different copies of things being produced by messages. When processors are far apart, message passing is likely a more natural fit, just like when you send email and a copy of the message is sent to the recipient. Here is a visual interpretation of message-passing:



Dataflow provides more structure than having "a bunch of threads that communicate with each other however they want." Instead, the programmer uses primitives to create a directed acyclic graph (DAG). A node in the graph performs some computation using inputs that arrive on its incoming edges. This data is provided by other nodes along their outgoing edges. A node starts computing when all of its inputs are available, something the implementation keeps track of automatically. Here is a visual interpretation of dataflow where different nodes perform different operations for some computation, such as "filter," "fade in," and "fade out:"



Data parallelism does not have explicit threads or nodes running different parts of the program at different times. Instead, it has primitives for parallelism that involve applying the *same* operation to different pieces of data at the same time. For example, you would have a primitive for applying some function to every element of an array. The implementation of this primitive would use parallelism rather than a sequential for-loop. Hence all the parallelism is done for you provided you can express your program using the available primitives. Examples include vector instructions on some processors and map-reduce style distributed systems. Here is a visual interpretation of data parallelism:



7.3 Basic Fork-Join Parallelism

This section shows how to use threads and shared memory to implement simple parallel algorithms. The only synchronization primitive we will need is `join`, which causes one thread to wait until another thread has terminated. We begin with simple pseudocode and then show how using threads in Java to achieve the same idea requires a bit more work (Section 7.3.1). We then argue that it is best for parallel code to *not* be written in terms of the number of processors available (Section 7.3.2) and show how to use recursive divide-and-conquer instead (Section 7.3.3). Because Java's threads are not engineered for this style of programming, we switch to the Java ForkJoin framework which is designed for our needs (Section 7.3.4). With all of this discussion in terms of the single problem of summing an array of integers, we then turn to other similar problems, introducing the terminology of *maps* and *reduces* (Section 7.3.5) as well as data structures other than arrays (Section 7.3.6).

7.3.1 A Simple Example: Okay Idea, Inferior Style

Most of this section will consider the problem of computing the sum of an array of integers. An $O(n)$ sequential solution to this problem is trivial:

```
int sum(int[] arr) {
```

```
   int ans = 0;
   for(int i=0; i < arr.length; i++)
      ans += arr[i];
   return ans;
}
```

If the array is large and we have extra processors available, we can get a more efficient parallel algorithm. Suppose we have 4 processors. Then we could do the following:

- Use the first processor to sum the first 1/4 of the array and store the result somewhere.
- Use the second processor to sum the second 1/4 of the array and store the result somewhere.
- Use the third processor to sum the third 1/4 of the array and store the result somewhere.
- Use the fourth processor to sum the fourth 1/4 of the array and store the result somewhere.
- Add the 4 stored results and return that as the answer.

This algorithm is clearly correct provided that the last step is started only after the previous four steps have completed. The first four steps can occur in parallel. More generally, if we have P processors, we can divide the array into P equal segments and have an algorithm that runs in time $O(n/P + P)$ where n/P is for the parallel part and P is for combining the stored results. Later we will see we can do better if P is very large, though that may be less of a practical concern.

In pseudocode, a convenient way to write this kind of algorithm is with a `FORALL` loop. A `FORALL loop` is like a `for` loop except it does all the iterations in parallel. Like a regular `for` loop, the code after a `FORALL` loop does not execute until the loop (i.e., all its iterations) are done. Unlike the `for` loop, the programmer is "promising" that all the iterations can

be done at the same time without them interfering with each other. Therefore, if one loop iteration writes to a location, then another iteration must not read or write to that location. However, it is fine for two iterations to read the same location: that does not cause any interference.

Here, then, is a pseudocode solution to using 4 processors to sum an array. Note it is essential that we store the 4 partial results in separate locations to avoid any interference between loop iterations.[2]

```
int sum(int[] arr) {
  results = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; ++i) {
     results[i] = sumRange(arr,(i*len)/4,((i+1)*len)/4);
  }
  return results[0] + results[1] + results[2] + results[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; ++j)
      result += arr[j];
   return result;
}
```

Unfortunately, Java and most other general-purpose languages do not have a `FORALL` loop. (Java has various kinds of for-loops, but all run all iterations on one thread.) We can encode this programming pattern explicitly using threads as follows:

[2]We must take care to avoid bugs due to integer-division truncation with the arguments to `sumRange`. We need to process each array element exactly once even if `len` is not divisible by 4. This code is correct; notice in particular that `((i+1)*len)/4` will always be `len` when `i==3` because `4*len` is divisible by 4. Moreover, we could write `(i+1)*len/4` since `*` and `/` have the same precedence and associate left-to-right. But `(i+1)*(len/4)` would *not* be correct. For the same reason, defining a variable `int rangeSize = len/4` and using `(i+1)*rangeSize` would *not* be correct.

1. In a regular `for` loop, create one thread to do each iteration of our `FORALL` loop, passing the data needed in the constructor. Have the threads store their answers in fields of themselves.

2. Wait for all the threads created in step 1 to terminate.

3. Combine the results by reading the answers out of the fields of the threads created in step 1.

To understand this pattern, we will first show a *wrong* version to get the idea. That is a common technique in this chapter — learning from wrong versions is extremely useful — but wrong versions are always clearly indicated.

Here is our WRONG attempt:

```
class SumThread extends java.lang.Thread {
   int lo; // fields for communicating inputs
   int hi;
   int[] arr;
   int ans = 0; // for communicating result
   SumThread(int[] a, int l, int h) {
     lo=l; hi=h; arr=a;
   }
   public void run() { // overriding, must have this type
     for(int i=lo; i<hi; i++)
       ans += arr[i];
   }
}
class C {
  static int sum(int[] arr) {
    int len = arr.length;
    int ans = 0;
```

```
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) {
      ts[i] = new SumThread(arr,(i*len)/4,((i+1)*len)/4);
      ts[i].start();
    }
    for(int i=0; i < 4; i++) {
      ans += ts[i].ans;
    }
    return ans;
  }
}
```

The code above gets most of the pieces right. The `sum` method creates 4 instances of `SumThread`, which is a subclass of `java.lang.Thread`, and calls `start` on each of them. The `SumThread` constructor takes as arguments the data that the thread needs to do its job, in this case, the array and the range for which this thread is responsible. (We use a convenient convention that ranges *include* the low bound and *exclude* the high bound.) The `SumThread` constructor stores this data in fields of the object so that the new thread has access to them in the `run` method. (Typically `run` would pass the relevant data to helper methods, but here the problem is simple enough not to bother.)

Notice each `SumThread` object also has an `ans` field. This is shared memory for communicating the answer back from the helper thread to the main thread. In particular, for the thread created on the `i`th iteration of the for-loop in `sum`, the field `this.ans` is an alias for `ts[i].ans`. So the main thread can sum the 4 `ans` fields from the threads it created to produce the final answer.

The bug in this code has to do with synchronization: The main thread does not wait for the helper threads to finish before it sums the `ans` fields. Remember that `start` returns immediately — otherwise we would not get any parallelism. So the `sum` method's second for-loop probably starts running before the helper threads are finished with their work. Having

one thread (the main thread) read a field while another thread (the helper thread) is writing the same field is a bug, and here it would produce a wrong (too-small) answer. We need to delay the second for-loop until the helper threads are done.

There is a method in `java.lang.Thread`, and therefore in all its subclasses such as `SumThread`, that is just what we need. If one thread, in our case the main thread, calls the `join` method of a `java.lang.Thread` object, in our case one of the helper threads, then this call *blocks* (i.e., does not return) unless/until the thread corresponding to the object has terminated. So we can add another for-loop to `sum` in-between the two loops already there to make sure all the helper threads finish before we add together the results:

```
for(int i=0; i < 4; i++)
   ts[i].join();
```

Notice it is the main thread that is calling `join`, which takes no arguments. On the first loop iteration, the main thread will block until the first helper thread is done. On the second loop iteration, the main thread will block until the second helper thread is done. It is certainly possible that the second helper thread actually finished before the first thread. This is not a problem: a call to `join` when the helper thread has already terminated just returns right away (no blocking).

Essentially, we are using two for-loops, where the first one creates helper threads and the second one waits for them all to terminate, to encode the idea of a FORALL-loop. This style of parallel programming is called "fork-join parallelism." It is like we create a "(4-way in this case) fork in the road of execution" and send each helper thread down one path of the fork. Then we join all the paths of the fork back together and have the single main thread continue. Fork-join parallelism can also be *nested*, meaning one of the helper threads forks its own helper threads. In fact, we will soon argue that this is better style. The term "join" is common in different programming languages and libraries, though honestly it is not the most descriptive English word for the concept.

It is common to combine the joining for-loop and the result-combining for-loop. Understanding why this is still correct helps understand the `join` primitive. So far we have

suggested writing code like this in our `sum` method:

```
for(int i=0; i < 4; i++)
   ts[i].join();
for(int i=0; i < 4; i++)
   ans += ts[i].ans;
return ans;
```

There is nothing wrong with the code above, but the following is also correct:

```
for(int i=0; i < 4; i++) {
   ts[i].join();
   ans += ts[i].ans;
}
return ans;
```

Here we do not wait for all the helper threads to finish before we start producing the final answer. But we still ensure that the main thread does not access a helper thread's `ans` field until at least that helper thread has terminated.

There is one last Java-specific detail we need when using the `join` method defined in `java.lang.Thread`. It turns out this method can throw a `java.lang.InterruptedException` so a method calling `join` will not compile unless it catches this exception or declares that it might be thrown. In many kinds of concurrent programming, it is bad style or even incorrect to ignore this exception, but for basic parallel programming like we are doing, this exception is a nuisance and will not occur. So we will say no more about it. Also the ForkJoin framework we will use starting in Section 7.3.4 has a different `join` method that does not throw exceptions.

Here, then, is a complete and correct program.[3] There is no change to the `SumThread` class. This example shows many of the key concepts of fork-join parallelism, but Section 7.3.2

[3]Technically, for very large arrays, `i*len` might be too large and we should declare one of these variables to have type `long`. We will ignore this detail throughout this chapter to avoid distractions, but `long` is a wiser choice when dealing with arrays that may be very large.

will explain why it is poor style and can lead to suboptimal performance. Sections 7.3.3 and 7.3.4 will then present a similar but better approach.

```
class SumThread extends java.lang.Thread {
   int lo; // fields for communicating inputs
   int hi;
   int[] arr;
   int ans = 0; // for communicating result
   SumThread(int[] a, int l, int h) {
     lo=l; hi=h; arr=a;
   }
   public void run() { // overriding, must have this type
     for(int i=lo; i<hi; i++)
       ans += arr[i];
   }
}
class C {
  static int sum(int[] arr) throws java.lang.InterruptedException {
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) {
      ts[i] = new SumThread(arr,(i*len)/4,((i+1)*len)/4);
      ts[i].start();
    }
    for(int i=0; i < 4; i++) {
      ts[i].join();
      ans += ts[i].ans;
    }
```

```
    return ans;
  }
```

7.3.2 Why Not To Use One Thread Per Processor

Having now presented a basic parallel algorithm, we will argue that the approach the algorithm takes is poor style and likely to lead to unnecessary inefficiency. Do not despair: the concepts we have learned like creating threads and using `join` will remain useful — and it was best to explain them using a too-simple approach. Moreover, many parallel programs are written in pretty much exactly this style, often because libraries like those in Section 7.3.4 are unavailable. Fortunately, such libraries are now available on many platforms.

The problem with the previous approach was dividing the work into exactly 4 pieces. This approach assumes there are 4 processors available to do the work (no other code needs them) and that each processor is given approximately the same amount of work. Sometimes these assumptions may hold, but it would be better to use algorithms that do not rely on such brittle assumptions. The rest of this section explains in more detail why these assumptions are unlikely to hold and some partial solutions. Section 7.3.3 then describes the better solution that we advocate.

Different computers have different numbers of processors

We want parallel programs that effectively use the processors available to them. Using exactly 4 threads is a horrible approach. If 8 processors are available, half of them will sit idle and our program will be no faster than with 4 processors. If 3 processors are available, our 4-thread program will take approximately twice as long as with 4 processors. If 3 processors are available and we rewrite our program to use 3 threads, then we will use resources effectively and the result will only be about 33% slower than when we had 4 processors and 4 threads. (We will take 1/3 as much time as the sequential version compared to 1/4 as much time. And 1/3 is 33% slower than 1/4.) But we do not want to have to edit our code every time we run it on a computer with a different number of processors.

A natural solution is a core software-engineering principle you should already know: Do

not use constants where a variable is appropriate. Our `sum` method can take as a parameter the number of threads to use, leaving it to some other part of the program to decide the number. (There are Java library methods to ask for the number of processors on the computer, for example, but we argue next that using that number is often unwise.) It would look like this:

```
static int sum(int[] arr, int numThreads) throws java.lang.InterruptedException {
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[numThreads];
  for(int i=0; i < numThreads; i++) {
    ts[i] = new SumThread(arr,(i*len)/numThreads,((i+1)*len)/numThreads);
    ts[i].start();
  }
  for(int i=0; i < numThreads; i++) {
    ts[i].join();
    ans += ts[i].ans;
  }
  return ans;
}
```

Note that you need to be careful with integer division not to introduce rounding errors when dividing the work.

The processors available to part of the code can change

The second dubious assumption made so far is that every processor is available to the code we are writing. But some processors may be needed by other programs or even other parts of the same program. We have parallelism after all — maybe the caller to `sum` is already part of some outer parallel algorithm. The operating system can reassign processors at any time, even when we are in the middle of summing array elements. It is fine to assume that

the underlying Java implementation will try to use the available processors effectively, but we should not assume 4 or even `numThreads` processors will be available from the beginning to the end of running our parallel algorithm.

We cannot always predictably divide the work into approximately equal pieces

In our `sum` example, it is quite likely that the threads processing equal-size chunks of the array take approximately the same amount of time. They may not, due to memory-hierarchy issues or other architectural effects, however. Moreover, more sophisticated algorithms could produce a large *load imbalance*, meaning different helper threads are given different amounts of work. As a simple example (perhaps too simple for it to actually matter), suppose we have a large `int[]` and we want to know how many elements of the array are prime numbers. If one portion of the array has more large prime numbers than another, then one helper thread may take longer.

In short, giving each helper thread an equal number of data elements is not necessarily the same as giving each helper thread an equal amount of work. And any load imbalance hurts our efficiency since we need to wait until all threads are completed.

A solution: Divide the work into smaller pieces

We outlined three problems above. It turns out we can solve all three with a perhaps counterintuitive strategy: *Use substantially more threads than there are processors.* For example, suppose to sum the elements of an array we created one thread for each 1000 elements. Assuming a large enough array (size greater than 1000 times the number of processors), the threads will not all run at once since a processor can run at most one thread at a time. But this is fine: the system will keep track of what threads are waiting and keep all the processors busy. There is some overhead to creating more threads, so we should use a system where this overhead is small.

This approach clearly fixes the first problem: any number of processors will stay busy until the very end when there are fewer 1000-element chunks remaining than there are processors. It also fixes the second problem since we just have a "big pile" of threads waiting

to run. If the number of processors available changes, that affects only how fast the pile is processed, but we are always doing useful work with the resources available. Lastly, this approach helps with the load imbalance problem: Smaller chunks of work make load imbalance far less likely since the threads do not run as long. Also, if one processor has a slow chunk, other processors can continue processing faster chunks.

We can go back to our cutting-potatoes analogy to understand this approach: Rather than give each of 4 cooks (processors), 1/4 of the potatoes, we have them each take a moderate number of potatoes, slice them, and then return to take another moderate number. Since some potatoes may take longer than others (they might be dirtier or have more eyes), this approach is better balanced and is probably worth the cost of the few extra trips to the pile of potatoes — especially if one of the cooks might take a break (processor used for a different program) before finishing his/her pile.

Unfortunately, this approach still has two problems addressed in Sections 7.3.3 and 7.3.4.

1. We now have more results to combine. Dividing the array into 4 total pieces leaves $\Theta(1)$ results to combine. Dividing the array into 1000-element chunks leaves `arr.length`/1000, which is $\Theta(n)$, results to combine. Combining the results with a sequential for-loop produces an $\Theta(n)$ algorithm, albeit with a smaller constant factor. To see the problem even more clearly, suppose we go to the extreme and use 1-element chunks — now the results combining reimplements the original sequential algorithm. In short, we need a better way to combine results.

2. Java's threads were not designed for small tasks like adding 1000 numbers. They will work and produce the correct answer, but the constant-factor overheads of creating a Java thread are far too large. A Java program that creates 100,000 threads on a small desktop computer is unlikely to run well at all — each thread just takes too much memory and the scheduler is overburdened and provides no asymptotic run-time guarantee. In short, we need a different implementation of threads that is designed for this kind of fork/join programming.

7.3.3 Divide-And-Conquer Parallelism

This section presents the idea of divide-and-conquer parallelism using Java's threads. Then Section 7.3.4 switches to using a library where this programming style is actually efficient. This progression shows that we can understand all the ideas using the basic notion of threads even though in practice we need a library that is designed for this kind of programming.

The key idea is to *change our algorithm* for summing the elements of an array to use recursive divide-and-conquer. To sum all the array elements in some range from `lo` to `hi`, do the following:

1. If the range contains only one element, return that element as the sum. Else in parallel:
   (a) Recursively sum the elements from `lo` to the middle of the range.
   (b) Recursively sum the elements from the middle of the range to `hi`.
2. Add the two results from the previous step.

The essence of the recursion is that steps 1a and 1b will themselves use parallelism to divide the work of their halves in half again. It is the same divide-and-conquer recursive idea as you have seen in algorithms like mergesort. For sequential algorithms for simple problems like summing an array, such fanciness is overkill. But for parallel algorithms, it is ideal.

As a small example (too small to actually want to use parallelism), consider summing an array with 10 elements. The algorithm produces the following tree of recursion, where the range `[i,j)` includes `i` and excludes `j`:

```
Thread: sum range [0,10)
   Thread: sum range [0,5)
      Thread: sum range [0,2)
         Thread: sum range [0,1) (return arr[0])
         Thread: sum range [1,2) (return arr[1])
         add results from two helper threads
```

```
   Thread: sum range [2,5)
      Thread: sum range [2,3) (return arr[2])
      Thread: sum range [3,5)
         Thread: sum range [3,4) (return arr[3])
         Thread: sum range [4,5) (return arr[4])
         add results from two helper threads
      add results from two helper threads
   add results from two helper threads
Thread: sum range [5,10)
   Thread: sum range [5,7)
      Thread: sum range [5,6) (return arr[5])
      Thread: sum range [6,7) (return arr[6])
      add results from two helper threads
   Thread: sum range [7,10)
      Thread: sum range [7,8) (return arr[7])
      Thread: sum range [8,10)
         Thread: sum range [8,9) (return arr[8])
         Thread: sum range [9,10) (return arr[9])
         add results from two helper threads
      add results from two helper threads
   add results from two helper threads
add results from two helper threads
```

The total amount of work done by this algorithm is $O(n)$ because we create approximately $2n$ threads and each thread either returns an array element or adds together results from two helper threads it created. Much more interestingly, if we have $O(n)$ processors, then this algorithm can run in $O(\log n)$ time, which is exponentially faster than the sequential algorithm. The key reason for the improvement is that the algorithm is combining results in parallel. The recursion forms a binary tree for summing subranges and the height of

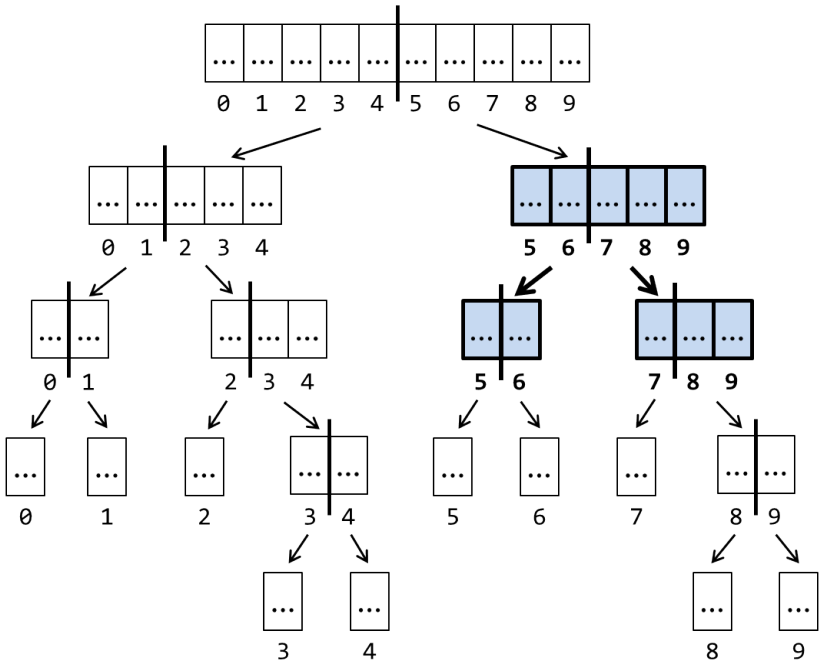


Figure (7.5) Recursion for summing an array where each node is a thread created by its parent. For example (see shaded region), the thread responsible for summing the array elements from index 5 (inclusive) to index 10 (exclusive), creates two threads, one to sum the elements from index 5 to 7 and the other to sum the elements from index 7 to 10.

this tree is $\log n$ for a range of size n. See Figure 7.5, which shows the recursion in a more conventional tree form where the number of nodes is growing exponentially faster than the tree height. With enough processors, the total running time corresponds to the tree *height*, not the tree *size*: this is the fundamental running-time benefit of parallelism. Later sections will discuss why the problem of summing an array has such an efficient parallel algorithm; not every problem enjoys exponential improvement from parallelism.

Having described the algorithm in English, seen an example, and informally analyzed its running time, let us now consider an actual implementation with Java threads and then modify it with two important improvements that affect only constant factors, but the constant factors are large. Then the next section will show the "final" version where we use the improvements and use a different library for the threads.

To start, here is the algorithm directly translated into Java, omitting some boilerplate

like putting the main `sum` method in a class and handling `java.lang.InterruptedException`.[4]

```
class SumThread extends java.lang.Thread {
  int lo; // fields for communicating inputs
  int hi;
  int[] arr;
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { arr=a; lo=l; hi=h; }
  public void run() {
    if(hi - lo == 1) {
      ans = arr[lo];
    } else {
      SumThread left  = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right = new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.start();
      left.join();
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
int sum(int[] arr) {
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
```

[4]For the exception, you cannot declare that `run` throws this exception because it overrides a method in `java.lang.Thread` that does not have this declaration. Since this exception is not going to be raised, it is reasonable to insert a `catch` statement and ignore this exception. The Java ForkJoin framework introduced in Section 7.3.4 does not have this problem; its `join` method does not throw checked exceptions.

```
}
```

Notice how each thread creates two helper threads `left` and `right` and then waits for them to finish. Crucially, the calls to `left.start` *and* `right.start` precede the calls to `left.join` *and* `right.join`. If for example, `left.join()` came before `right.start()`, then the algorithm would have no effective parallelism whatsoever. It would still produce the correct answer, but so would the original much simpler sequential program.

As a minor but important coding point, notice that the "main" `sum` method calls the `run` method directly. As such, this is an ordinary method call like you have used since you started programming; the caller and callee are part of the same thread. The fact that the object is a subclass of `java.lang.Thread` is only relevant if you call the "magic" `start` method, which calls `run` in a new thread.

In practice, code like this produces far too many threads to be efficient. To add up four numbers, does it really make sense to create six new threads? Therefore, implementations of fork/join algorithms invariably use a *cutoff* below which they switch over to a sequential algorithm. Because this cutoff is a constant, it has no effect on the asymptotic behavior of the algorithm. What it does is eliminate the vast majority of the threads created, while still preserving enough parallelism to balance the load among the processors.

Here is code using a cutoff of 1000. As you can see, using a cutoff does not really complicate the code.

```
class SumThread extends java.lang.Thread {
  static int SEQUENTIAL_CUTOFF = 1000;
  int lo; // fields for communicating inputs
  int hi;
  int[] arr;
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { arr=a; lo=l; hi=h; }
  public void run(){
    if(hi - lo < SEQUENTIAL_CUTOFF) {
```

```
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    } else {
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.start();
      left.join();
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
int sum(int[] arr) {
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

Using cut-offs is common in divide-and-conquer programming, even for sequential algorithms. For example, it is typical for quicksort to be slower than an $O(n^2)$ sort like insertion sort for small arrays ($n < 10$ or so). Therefore, it is common to have the recursive quicksort switch over to insertion sort for small subproblems. In parallel programming, switching over to a sequential algorithm below a cutoff is *the exact same idea*. In practice, the cutoffs are usually larger, with numbers between 500 and 5000 being typical.

It is often worth doing some quick calculations to understand the benefits of things like cutoffs. Suppose we are summing an array with 2^{30} elements. Without a cutoff, we would use $2^{31} - 1$, (i.e., two billion) threads. With a cutoff of 1000, we would use approximately 2^{21} (i.e., 2 million) threads since the last 10 levels of the recursion would be eliminated.

Computing $1 - 2^{21}/2^{31}$, we see we have eliminated 99.9% of the threads. Use cutoffs!

Our second improvement may seem anticlimactic compared to cutoffs because it only reduces the number of threads by an additional factor of two. Nonetheless, it is worth seeing for efficiency especially because the ForkJoin framework in the next section performs poorly if you do not do this optimization "by hand." The key is to notice that all threads that create two helper threads are not doing much work themselves: they divide the work in half, give it to two helpers, wait for them to finish, and add the results. Rather than having all these threads wait around, it is more efficient to create *one helper thread* to do half the work and have the thread do the other half *itself*. Modifying our code to do this is easy since we can just call the `run` method directly, which recall is just an ordinary method call unlike the "magic" `start` method.

```
class SumThread extends java.lang.Thread {
  static int SEQUENTIAL_CUTOFF = 1000;
  int lo; // fields for communicating inputs
  int hi;
  int[] arr;
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { arr=a; lo=l; hi=h; }
  public void run(){
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    } else {
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.run();
      left.join();
```

```
      ans = left.ans + right.ans;
    }
  }
}
int sum(int[] arr) {
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

Notice how the code above creates two `SumThread` objects, but only creates one helper thread with `left.start()`. It then does the right half of the work itself by calling `right.run()`. There is only one call to `join` because only one helper thread was created. The order here is still essential so that the two halves of the work are done in parallel. Creating a `SumThread` object for the right half and then calling `run` rather than creating a thread may seem odd, but it keeps the code from getting more complicated and still conveys the idea of dividing the work into two similar parts that are done in parallel.

Unfortunately, even with these optimizations, the code above will run poorly in practice, especially if given a large array. The implementation of Java's threads is not engineered for threads that do such a small amount of work as adding 1000 numbers: it takes much longer just to create, start running, and dispose of a thread. The space overhead may also be prohibitive. In particular, it is not uncommon for a Java implementation to pre-allocate the maximum amount of space it allows for the call-stack, which might be 2MB or more. So creating thousands of threads could use gigabytes of space. Hence we will switch to the library described in the next section for parallel programming. We will return to Java's threads when we learn concurrency control in the next chapter because the synchronization operations we will use work with Java's threads.

7.3.4 The Java ForkJoin Framework

Java 7 includes classes in the `java.util.concurrent` package designed exactly for the kind of fine-grained fork-join parallel computing this chapter uses. In addition to supporting lightweight threads (which the library calls ForkJoin tasks) that are small enough that even a million of them should not overwhelm the system, the implementation includes a scheduler and run-time system with provably optimal expected-time guarantees, as described in Section 7.4. Similar libraries for other languages include Intel's Thread Building Blocks, Microsoft's Task Parallel Library for C#, and others. The core ideas and implementation techniques go back much further to the Cilk language, an extension of C developed since 1994.

This section describes just a few practical details and library specifics. Compared to Java threads, the core ideas are all the same, but some of the method names and interfaces are different — in places more complicated and in others simpler. Naturally, we give a full example (actually two) for summing an array of numbers. The actual library contains many other useful features and classes, but we will use only the primitives related to forking and joining, implementing anything else we need ourselves.

For introductory notes on using the library and avoiding some difficult-to-diagnose pitfalls, see `http://homes.cs.washington.edu/~djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html`.

We first show a full program (minus a `main` method) that is as much as possible like the version we wrote using Java threads. We show a version using a sequential cut-off and only one helper thread at each recursive subdivision though removing these important improvements would be easy. After discussing this version, we show a second version that uses Java's generic types and a different library class. This second version is better style, but easier to understand after the first version.

FIRST VERSION (INFERIOR STYLE):

```
import java.util.concurrent.ForkJoinPool;
```

```
import java.util.concurrent.RecursiveAction;

class SumArray extends RecursiveAction {
   static int SEQUENTIAL_THRESHOLD = 1000;

   int lo;
   int hi;
   int[] arr;
   int ans = 0;
   SumArray(int[] a, int l, int h) { lo=l; hi=h; arr=a; }

   protected void compute() {
      if(hi - lo <= SEQUENTIAL_THRESHOLD) {
         for(int i=lo; i < hi; ++i)
            ans += arr[i];
      } else {
          SumArray left  = new SumArray(arr,lo,(hi+lo)/2);
          SumArray right = new SumArray(arr,(hi+lo)/2,hi);
          left.fork();
          right.compute();
          left.join();
          ans = left.ans + right.ans;
      }
    }
}
class Main {
  static final ForkJoinPool fjPool = new ForkJoinPool();
  static int sumArray(int[] array) {
```

```
    SumArray t = new SumArray(array,0,array.length);
    fjPool.invoke(t);
    return t.ans;
  }
}
```

While there are many differences compared to using Java's threads, the overall structure of the algorithm should look similar. Furthermore, most of the changes are just different names for classes and methods:

- Subclass `java.util.concurrent.RecursiveAction` instead of `java.lang.Thread`.
- The method that "magically" creates parallelism is called `fork` instead of `start`.
- The method that starts executing when a new thread begins is called `compute` instead of `run`. Recall these methods can also be called normally.
- (The method `join` is still called `join`.)

The small additions involve creating a `ForkJoinPool` and using the `invoke` method on it. These are just some details because the library is not built into the Java *language*, so we have to do a little extra to initialize the library and start using it. Here is all you really need to know:

- The entire program should have exactly one `ForkJoinPool`, so it makes sense to store it in a static field and use it for all the parallel algorithms in your program.
- Inside a subclass of `RecursiveAction`, you use `fork`, `compute`, and `join` just like we previously used `start`, `run`, and `join`. But outside such classes, you cannot use these methods or the library will not work correctly. To "get the parallelism started," you instead use the `invoke` method of the `ForkJoinPool`. You pass to `invoke` a subclass of `RecursiveAction` and that object's `compute` method will be called. Basically, use `invoke` once to start the algorithm and then `fork` or `compute` for the recursive calls.

Conversely, do *not* use `invoke` from "inside" the library, i.e., in `compute` or any helper methods it uses.

We will present one final version of our array-summing program to demonstrate one more class of the ForkJoin framework that you should use as a matter of style. The `RecursiveAction` class is best only when the subcomputations do not produce a result, whereas in our example they do: the sum of the range. It is quite common not to produce a result, for example a parallel program that increments every element of an array. So far, the way we have "returned" results is via a field, which we called `ans`.

Instead, we can subclass `RecursiveTask` instead of `RecursiveAction`. However, `RecursiveTask` is a generic class with one type parameter: the type of value that `compute` should return. Here is the full version of the code using this more convenient and less error-prone class, followed by an explanation:

FINAL, BETTER VERSION:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class SumArray extends RecursiveTask<Integer> {
   static int SEQUENTIAL_THRESHOLD = 1000;

   int lo;
   int hi;
   int[] arr;
   SumArray(int[] a, int l, int h) { lo=l; hi=h; arr=a; }

   public Integer compute() {
      if(hi - lo <= SEQUENTIAL_THRESHOLD) {
         int ans = 0;
```

```
        for(int i=lo; i < hi; ++i)
           ans += arr[i];
        return ans;
      } else {
          SumArray left  = new SumArray(arr,lo,(hi+lo)/2);
          SumArray right = new SumArray(arr,(hi+lo)/2,hi);
          left.fork();
          int rightAns = right.compute();
          int leftAns = left.join();
          return leftAns + rightAns;
      }
    }
}
class Main {
  static final ForkJoinPool fjPool = new ForkJoinPool();
  static int sumArray(int[] array) {
     return fjPool.invoke(new SumArray(array,0,array.length));
  }
}
```

Here are the differences from the version that subclasses `RecursiveAction`:

- `compute` returns an object of whatever (object) type we want. Because `int` is not an object type, we write `Integer` instead. Java implicitly converts an `int` to/from an `Integer` in all the places it is needed in this program.
- The `join` method also returns the value returned by `compute` in the thread that is being joined to. Therefore, we do not need an `ans` field: we write `int leftAns = left.join()` to get the value that the thread bound to `left` returned from its `compute` method.

- The `invoke` method works similarly when passed a subclass of a `RecursiveTask` class: It returns the value of the `compute` method that is called.

- The class declaration of `SumArray` indicates that it extends `RecursiveTask<Integer>`. More generally, always put the same type between the angle brackets as the return type of `compute`.

- Note that because Java expressions are always evaluated left-to-right, we could replace the last 3 lines of the else-branch in `compute` with just `return right.compute() + left.join();`. But this is rather subtle — `left.join() + right.compute()` would destroy the parallelism — so we do not advise writing this shorter but less clear code.

If you are familiar with Java's generic types, this use of them should not be particularly perplexing. The library is also using static overloading for the `invoke` method. But as *users* of the library, it suffices just to follow the pattern in the example above.

Why is there `RecursiveAction` and `RecursiveTask`? The former is better style when there really is nothing to return; otherwise `RecursiveTask` is better. Both are actually implemented in terms of the same superclass inside the library. You can use that class directly, but it is less convenient.

Given the convenience of not needing a field for returning values, why not also provide the convenience of not needing fields to pass arguments (`arr`, `lo`, and `hi`)? That *would* be nice, but there just isn't a particularly pleasant way to write such a library in Java. The ForkJoin framework is just a library; it cannot make any changes/extensions to the Java language. It uses some advanced Java features to be as convenient as it can be.

7.3.5 Reductions and Maps

It may seem that given all the work we did to implement something as conceptually simple as summing an array that fork/join programming is too complicated. To the contrary, it turns out that many, many problems can be solved very much like we solved this one. Just like regular for-loops took some getting used to when you started programming but now you

can recognize exactly what kind of loop you need for all sorts of problems, divide-and-conquer parallelism often follows a small number of patterns. Once you know the patterns, most of your programs are largely the same.

For example, here are several problems for which efficient parallel algorithms look almost identical to summing an array:

- Count how many array elements satisfy some property (e.g., how many elements are the number 42).
- Find the maximum or minimum element of an array.
- Given an array of strings, compute the sum (or max, or min) of all their lengths.
- Find the left-most array index that has an element satisfying some property.

Compared to summing an array, all that changes is the base case for the recursion and how we combine results. For example, to find the index of the leftmost 42 in an array of length n, we can do the following (where a final result of n means the array does not hold a 42):

- For the base case, return `lo` if `arr[lo]` holds 42 and `n` otherwise.
- To combine results, return the smaller number.

Implement one or two of these problems to convince yourself they are not any harder than what we have already done. Or come up with additional problems that can be solved the same way.

Problems that have this form are so common that there is a name for them: *reductions*, which you can remember by realizing that we take a collection of data items (in an array) and *reduce* the information down to a single result. As we have seen, the way reductions can work in parallel is to compute answers for the two halves recursively and in parallel and then merge these to produce a result.

However, we should be clear that *not every problem over an array of data can be solved with a simple parallel reduction.* To avoid getting into arcane problems, let's just describe a general situation. Suppose you have sequential code like this:

```
interface BinaryOperation<T> {
   T m(T x, T y);
}
class C<T> {
   T fold(T[] arr, BinaryOperation<T> binop, T initialValue) {
     T ans = initialValue;
     for(int i=0; i < arr.length; ++i)
       ans = binop.m(ans,arr[i]);
     return ans;
   }
 }
```

The name `fold` is conventional for this sort of algorithm. The idea is to start with `initialValue` and keep updating the "answer so far" by applying some binary function `m` to the current answer and the next element of the array.

Without any additional information about what `m` computes, this algorithm cannot be effectively parallelized since we cannot process `arr[i]` until we know the answer from the first `i-1` iterations of the for-loop. For a more humorous example of a procedure that cannot be sped up given additional resources: 9 women can't make a baby in 1 month.

So what do we have to know about the `BinaryOperation` above in order to use a parallel reduction? It turns out all we need is that the operation is *associative*, meaning for all a, b, and c, $m(a, m(b, c))$ is the same as $m(m(a, b), c)$. Our array-summing algorithm is correct because $a + (b + c) = (a + b) + c$. Our find-the-leftmost-index-holding 42 algorithm is correct because *min* is also an associative operator.

Because reductions using associative operators are so common, we could write one generic algorithm that took the operator, and what to do for a base case, as arguments. This is an example of higher-order programming, and the fork-join framework has several classes providing this sort of functionality. Higher-order programming has many, many advantages (see the end of this section for a popular one), but when first *learning* a programming pat-

tern, it is often useful to "code it up yourself" a few times. For that reason, we encourage writing your parallel reductions manually in order to see the parallel divide-and-conquer, even though they all really look the same.

Parallel reductions are not the only common pattern in parallel programming. An even simpler one, which we did not start with because it is just so easy, is a parallel *map*. A map performs an operation on each input element independently; given an array of inputs, it produces an array of outputs of the same length. A simple example would be multiplying every element of an array by 2. An example using two inputs and producing a separate output would be vector addition. Using pseudocode, we could write:

```
int[] add(int[] arr1, int[] arr2) {
  assert(arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  FORALL(int i=0; i < arr1.length; i++)
    ans[i] = arr1[i] + arr2[i];
  return ans;
}
```

Coding up this algorithm in the ForkJoin framework is straightforward: Have the main thread create the `ans` array and pass it before starting the parallel divide-and-conquer. Each thread object will have a reference to this array but will assign to different portions of it. Because there are no other results to combine, subclassing `RecursiveAction` is appropriate. Using a sequential cut-off and creating only one new thread for each recursive subdivision of the problem remain important — these ideas are more general than the particular programming pattern of a map or a reduce.

Recognizing problems that are fundamentally maps and/or reduces over large data collections is a valuable skill that allows efficient parallelization. In fact, it is one of the key ideas behind Google's MapReduce framework and the open-source variant Hadoop. In these systems, the programmer just writes the operations that describe how to map data (e.g., "multiply by 2") and reduce data (e.g., "take the minimum"). The system then does

all the parallelization, often using hundreds or thousands of computers to process gigabytes or terabytes of data. For this to work, the programmer must provide operations that have no side effects (since the order they occur is unspecified) and reduce operations that are associative (as we discussed). As parallel programmers, it is often enough to "write down the maps and reduces" — leaving it to systems like the ForkJoin framework or Hadoop to do the actual scheduling of the parallelism.

7.3.6 Data Structures Besides Arrays

So far we have considered only algorithms over one-dimensional arrays. Naturally, one can write parallel algorithms over any data structure, but divide-and-conquer parallelism requires that we can efficiently (ideally in $O(1)$ time) divide the problem into smaller pieces. For arrays, dividing the problem involves only $O(1)$ arithmetic on indices, so this works well.

While arrays are the most common data structure in parallel programming, balanced trees, such as AVL trees or B trees, also support parallel algorithms well. For example, with a binary tree, we can fork to process the left child and right child of each node in parallel. For good sequential cut-offs, it helps to have stored at each tree node the number of descendants of the node, something easy to maintain. However, for trees with guaranteed balance properties, other information — like the height of an AVL tree node — should suffice.

Certain tree problems will not run faster with parallelism. For example, searching for an element in a balanced binary search tree takes $O(\log n)$ time with or without parallelism. However, maps and reduces over balanced trees benefit from parallelism. For example, summing the elements of a binary tree takes $O(n)$ time sequentially where n is the number of elements, but with a sufficiently large number of processors, the time is $O(h)$, where h is the height of the tree. Hence, tree balance is even more important with parallel programming: for a balanced tree $h = \Theta(\log n)$ compared to the worst case $h = \Theta(n)$.

For the same reason, parallel algorithms over regular linked lists are typically poor. Any problem that requires reading all n elements of a linked list takes time $\Omega(n)$ regardless of how many processors are available. (Fancier list data structures like skip lists are better for

exactly this reason — you can get to all the data in $O(\log n)$ time.) Streams of input data, such as from files, typically have the same limitation: it takes linear time to read the input and this can be the bottleneck for the algorithm.

There can still be benefit to parallelism with such "inherently sequential" data structures and input streams. Suppose we had a map operation over a list but each operation was itself an expensive computation (e.g., decrypting a significant piece of data). If each map operation took time $O(x)$ and the list had length n, doing each operation in a separate thread (assuming, again, no limit on the number of processors) would produce an $O(x+n)$ algorithm compared to the sequential $O(xn)$ algorithm. But for simple operations like summing or finding a maximum element, there would be no benefit.

7.4 Analyzing Fork-Join Algorithms

As with any algorithm, a fork-join parallel algorithm should be correct and efficient. This section focuses on the latter even though the former should always be one's first concern. For efficiency, we will focus on asymptotic bounds and analyzing algorithms that are not written in terms of a fixed number of processors. That is, just as the size of the problem n will factor into the asymptotic running time, so will the number of processors P. The ForkJoin framework (and similar libraries in other languages) will give us an optimal expected-time bound for any P. This section explains what that bound is and what it means, but we will not discuss *how* the framework achieves it.

We then turn to discussing Amdahl's Law, which analyzes the running time of algorithms that have both sequential parts and parallel parts. The key and depressing upshot is that programs with even a small sequential part quickly stop getting much benefit from running with more processors.

Finally, we discuss Moore's "Law" in contrast to Amdahl's Law. While Moore's Law is also important for understanding the progress of computing power, it is not a mathematical theorem like Amdahl's Law.

7.4.1 Work and Span

Defining Work and Span We define T_P to be the time a program/algorithm takes to run if there are P processors available during its execution. For example, if a program was the only one running on a quad-core machine, we would be particularly interested in T_4, but we want to think about T_P more generally. It turns out we will reason about the general T_P in terms of T_1 and T_∞:

- T_1 is called the *work*. By definition, this is how long it takes to run on one processor. More intuitively, it is just the total of all the running time of all the pieces of the algorithm: we have to do all the work before we are done, and there is exactly one processor (no parallelism) to do it. In terms of fork-join, we can think of T_1 as doing one side of the fork and then the other, though the total T_1 does not depend on how the work is scheduled.

- T_∞ is called the *span*, though other common terms are the *critical path length* or *computational depth*. By definition, this is how long it takes to run on an unlimited number of processors. Notice this is *not* necessarily $O(1)$ time; the algorithm still needs to do the forking and combining of results. For example, under our model of computation — where creating a new thread and adding two numbers are both $O(1)$ operations — the algorithm we developed is asymptotically optimal with $T_\infty = \Theta(\log n)$ for an array of length n.

We need a more precise way of characterizing the execution of a parallel program so that we can describe and compute the work, T_1, and the span, T_∞. We will describe a program execution as a directed acyclic graph (dag)[5] where:

- Nodes are pieces of work the program performs. Each node will be a constant, i.e.,

[5]A directed acyclic graph (the word *graph* here is the name for a data structure and does not mean a chart for presenting data) is a set of nodes and edges between pairs of nodes where each edge has a direction (a *source* that it starts from and a *target* that it ends at). Furthermore, the graph is acyclic, which means you cannot start at any node, follow edges from source-to-target, and get back to the starting node.

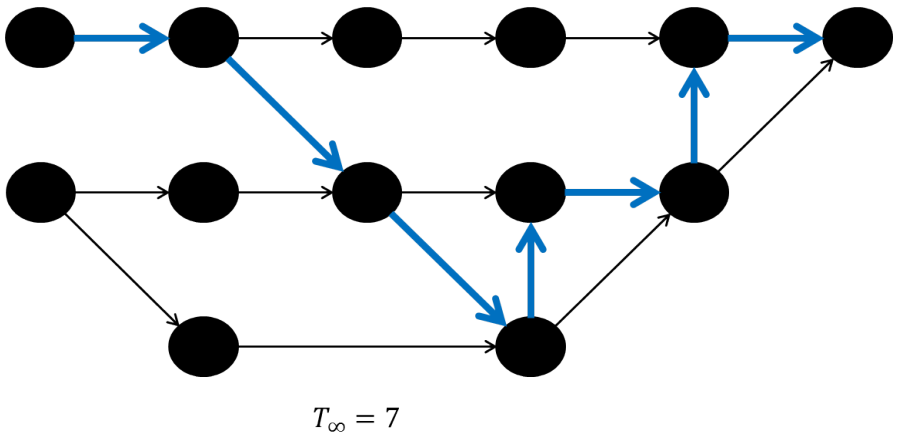


Figure (7.6) An example dag and the path (see thicker blue arrows) that determines its span.

$O(1)$, amount of work that is performed sequentially. So T_1 is asymptotically just the number of nodes in the dag.

- Edges represent that the source node must complete before the target node begins. That is, there is a *computational dependency* along the edge. This idea lets us visualize T_∞: With unlimited processors, we would immediately start every node as soon as its predecessors in the graph had finished. Therefore T_∞ is just the length of the longest path in the dag.

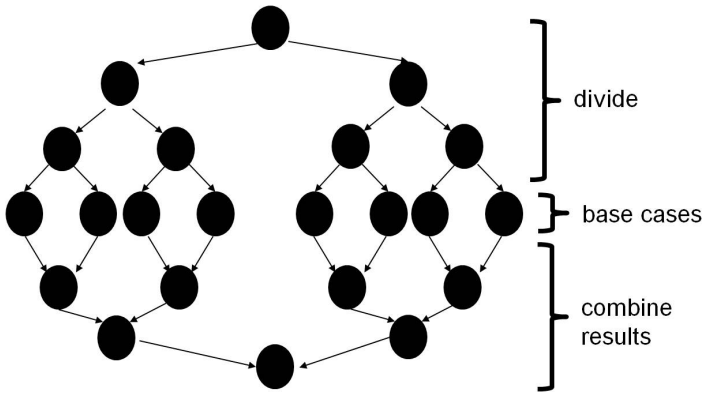
Figure 7.6 shows an example dag and the longest path, which determines T_∞.

If you have studied combinational hardware circuits, this model is strikingly similar to the dags that arise in that setting. For circuits, *work* is typically called the *size* of the circuit, (i.e., the amount of hardware) and *span* is typically called the *depth* of the circuit, (i.e., the time, in units of "gate delay," to produce an answer).

With basic fork-join divide-and-conquer parallelism, the execution dags are quite simple: The $O(1)$ work to set up two smaller subproblems is one node in the dag. This node has two outgoing edges to two new nodes that start doing the two subproblems. (The fact that one subproblem might be done by the same thread is not relevant here. Nodes are not threads. They are $O(1)$ pieces of work.) The two subproblems will lead to their own dags. When we

join on the results of the subproblems, that creates a node with incoming edges from the last nodes for the subproblems. This same node can do an $O(1)$ amount of work to combine the results. (If combining results is more expensive, then it needs to be represented by more nodes.)

Overall, then, the dag for a basic parallel reduction would look like this:



The root node represents the computation that divides the array into two equal halves. The bottom node represents the computation that adds together the two sums from the halves to produce the final answer. The base cases represent reading from a one-element range assuming no sequential cut-off. A sequential cut-off "just" trims out levels of the dag, which removes most of the nodes but affects the dag's longest path by "only" a constant amount. Note that this dag is a conceptual description of how a program executes; the dag is not a data structure that gets built by the program.

From the picture, it is clear that a parallel reduction is basically described by two balanced binary trees whose size is proportional to the input data size. Therefore T_1 is $O(n)$ (there are approximately $2n$ nodes) and T_∞ is $O(\log n)$ (the height of each tree is approximately $\log n$). For the particular reduction we have been studying — summing an array — Figure 7.7 visually depicts the work being done for an example with 8 elements. The work in the nodes in the top half is to create two subproblems. The work in the nodes in the bottom half is to combine two results.

The dag model of parallel computation is much more general than for simple fork-join algorithms. It describes all the work that is done and the earliest that any piece of that work

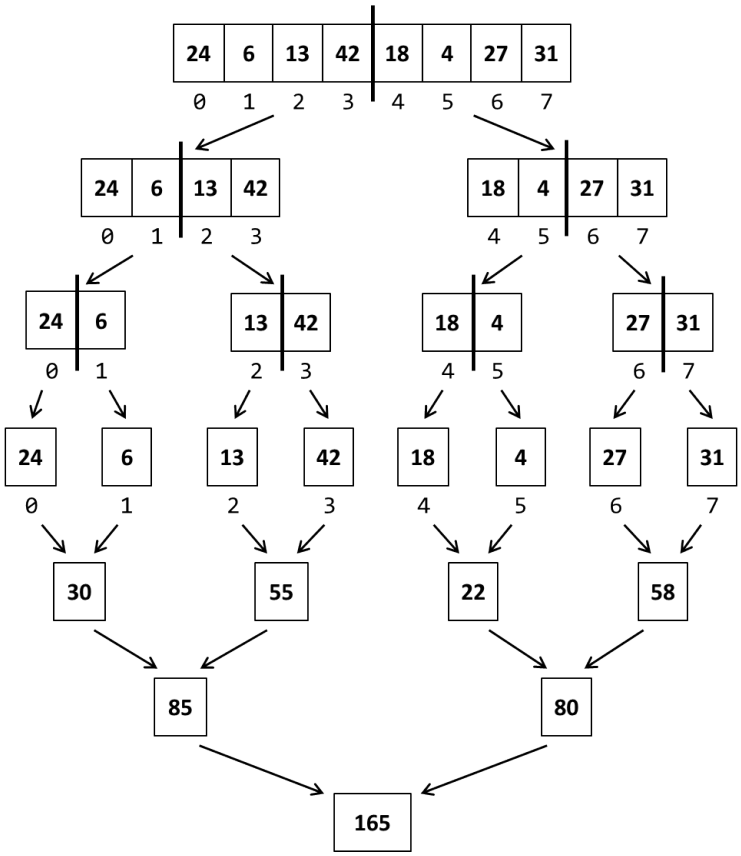


Figure (7.7) Example execution dag for summing an array.

could begin. To repeat, T_1 and T_∞ become simple graph properties: the number of nodes and the length of the longest path, respectively.

Defining Speedup and Parallelism Having defined work and span, we can use them to define some other terms more relevant to our real goal of reasoning about T_P. After all, if we had only one processor then we would not study parallelism and having infinity processors is impossible.

We define the *speedup* on P processors to be T_1/T_P. It is basically the ratio of how much faster the program runs given the extra processors. For example, if T_1 is 20 seconds and T_4 is 8 seconds, then the speedup for $P = 4$ is 2.5.

You might naively expect a speed-up of 4, or more generally P for T_P. In practice, such a *perfect speedup* is rare due to several issues including the overhead of creating threads and communicating answers among them, memory-hierarchy issues, and the inherent computational dependencies related to the span. In the rare case that doubling P cuts the running time in half (i.e., doubles the speedup), we call it *perfect linear speedup*. In practice, this is not the absolute limit; one can find situations where the speedup is even higher even though our simple computational model does not capture the features that could cause this.

It is important to note that reporting only T_1/T_P can be "dishonest" in the sense that it often overstates the advantages of using multiple processors. The reason is that T_1 is the time it takes to run the *parallel algorithm* on one processor, but this algorithm is likely to be much slower than an algorithm designed sequentially. For example, if someone wants to know the benefits of summing an array with parallel fork-join, they probably are most interested in comparing T_P to the time for the sequential for-loop. If we call the latter S, then the ratio S/T_P is usually the speed-up of interest and will be lower, due to constant factors like the time to create recursive tasks, than the definition of speed-up T_1/T_P. One measure of the overhead of using multiple threads is simply T_1/S, which is usually greater than 1.

As a final definition, we call T_1/T_∞ the *parallelism* of an algorithm. It is a measure of

how much improvement one could possibly hope for since it should be at least as great as the speed-up for any P. For our parallel reductions where the work is $\Theta(n)$ and the span is $\Theta(\log n)$, the parallelism is $\Theta(n/\log n)$. In other words, there is exponential available parallelism (n grows exponentially faster than $\log n$), meaning with enough processors we can hope for an exponential speed-up over the sequential version.

The ForkJoin Framework Bound Under some important assumptions we will describe below, algorithms written using the ForkJoin framework, in particular the divide-and-conquer algorithms in this chapter, have the following *expected* time bound:

$$T_P \text{ is } O(T_1/P + T_\infty)$$

The bound is *expected* because internally the framework uses randomness, so the bound can be violated from "bad luck" but such "bad luck" is exponentially unlikely, so it simply will not occur in practice. This is exactly like the expected-time running-time guarantee for the sequential quicksort algorithm when a pivot element is chosen randomly. Because this chapter does not describe the framework's implementation, we will not see where the randomness arises.

Notice that, ignoring constant factors, this bound is optimal: Given only P processors, no framework can expect to do better than T_1/P or better than T_∞. For small P, the term T_1/P is likely to be dominant and we can expect roughly linear speed-up. As P grows, the span becomes more relevant and the limit on the run-time is more influenced by T_∞.

Constant factors can be relevant, and it is entirely possible that a hand-crafted parallel algorithm in terms of some fixed P could do better than a generic library that has no idea what sort of parallel algorithm it is running. But just like we often use asymptotically optimal data structures even if hand-crafted ones for our task might be a little faster, using a library such as this is often an excellent approach.

Thinking in terms of the program-execution dag, it is rather amazing that a library can achieve this optimal result. While the program is running, it is the framework's job to

choose among all the threads that *could* run next (they are not blocked waiting for some other thread to finish) and assign P of them to processors. For simple parallel reductions, the choice hardly matters because all paths to the bottom of the dag are about the same length, but for arbitrary dags it seems important to work on the longer paths. Yet it turns out a much greedier algorithm that just picks randomly among the available threads will do only a constant factor worse. But this is all about the library's internal scheduling algorithm (which is not actually totally random) and we, as library users, can just rely on the provided bound.

However, as mentioned above, the bound holds only under a couple assumptions. The first is that all the threads you create to do subproblems do approximately the same amount of work. Otherwise, if a thread with much-more-work-to-do is scheduled very late, other processors will sit idle waiting for this laggard to finish. The second is that all the threads do a small but not tiny amount of work. This again helps with load balancing. The documentation for the library suggests aiming for approximately 5000 basic computational steps (additions, method calls, etc.) and that the exact figure can be off by a factor of 10 or so without problem. In other words, just avoid threads that do millions of operations as well as threads that do dozens.

To summarize, as a user of a library like this, your job is to pick a good parallel algorithm, implement it in terms of divide-and-conquer with a reasonable sequential cut-off, and analyze the expected run-time in terms of the provided bound. The library's job is to give this bound while trying to maintain low constant-factor overheads. While this library is particularly good for *this* style of programming, this basic division is common: application writers develop good algorithms and rely on some underlying *thread scheduler* to deliver reasonable performance.

7.4.2 Amdahl's Law

So far we have analyzed the running time of a parallel algorithm. While a parallel algorithm could have some "sequential parts" (a part of the dag where there is a long linear

sequence of nodes), it is common to think of an execution in terms of some entirely parallel parts (e.g., maps and reductions) and some entirely sequential parts. The sequential parts could simply be algorithms that have not been parallelized or they could be inherently sequential, like reading in input. As this section shows, even a little bit of sequential work in your program drastically reduces the speed-up once you have a significant number of processors.

This result is really a matter of very basic algebra. It is named after Gene Amdahl, who first articulated it. Though almost all computer scientists learn it and understand it, it is all too common to forget its implications. It is, perhaps, counterintuitive that just a little non-parallelism has such a drastic limit on speed-up. But it's a fact, so learn and remember Amdahl's Law!

With that introduction, here is the full derivation of Amdahl's Law: Suppose the work T_1 is 1, i.e., the total program execution time on one processor is 1 "unit time." Let S be the portion of the execution that cannot be parallelized and assume the rest of the execution $(1 - S)$ gets perfect linear speed-up on P processors for any P. Notice this is a charitable assumption about the parallel part equivalent to assuming the span is $O(1)$. Then:

$$T_1 = S + (1 - S) = 1$$

$$T_P = S + (1 - S)/P$$

Notice all we have assumed is that the parallel portion $(1 - S)$ runs in time $(1 - S)/P$. Then the speed-up, by definition is:

$$\text{Amdahl's Law:} \quad T_1/T_P = 1/(S + (1 - S)/P)$$

As a corollary, the parallelism is just the simplified equation as P goes to ∞:

$$T_1/T_\infty = 1/S$$

The equations may look innocuous until you start plugging in values. For example, if 33% of a program is sequential, then a billion processors can achieve a speed-up of at most 3. That is just common sense: they cannot speed-up 1/3 of the program, so even if the rest of the program runs "instantly" the speed-up is only 3.

The "problem" is when we expect to get twice the performance from twice the computational resources. If those extra resources are processors, this works only if most of the execution time is still running parallelizable code. Adding a second or third processor can often provide significant speed-up, but as the number of processors grows, the benefit quickly diminishes.

Recall that from 1980–2005 the processing speed of desktop computers doubled approximately every 18 months. Therefore, 12 years or so was long enough to buy a new computer and have it run an old program 100 times faster. Now suppose that instead in 12 years we have 256 processors rather than 1 but all the processors have the same speed. What percentage of a program would have to be perfectly parallelizable in order to get a speed-up of 100? Perhaps a speed-up of 100 given 256 cores seems easy? Plugging into Amdahl's Law, we need:

$$100 \leq 1/(S + (1 - S)/256)$$

Solving for S reveals that at most 0.61% of the program can be sequential.

Given depressing results like these — and there are many, hopefully you will draw some possible-speedup plots as homework exercises — it is tempting to give up on parallelism as a means to performance improvement. While you should never forget Amdahl's Law, you should also not entirely despair. Parallelism does provide real speed-up for performance-critical parts of programs. You just do not get to speed-up *all* of your code by buying a faster computer. More specifically, there are two common workarounds to the fact-of-life that is Amdahl's Law:

1. We can find new parallel algorithms. Given enough processors, it is worth parallelizing

something (reducing span) even if it means more total computation (increasing work). Amdahl's Law says that as the number of processors grows, span is more important than work. This is often described as "scalability matters more than performance" where scalability means can-use-more-processors and performance means run-time on a small-number-of-processors. In short, large amounts of parallelism can change your algorithmic choices.

2. We can use the parallelism to solve new or bigger problems rather than solving the same problem faster. For example, suppose the parallel part of a program is $O(n^2)$ and the sequential part is $O(n)$. As we increase the number of processors, we can increase n with only a small increase in the running time. One area where parallelism is very successful is computer graphics (animated movies, video games, etc.). Compared to years ago, it is not so much that computers are rendering the same scenes faster; it is that they are rendering more impressive scenes with more pixels and more accurate images. In short, parallelism can enable new things (provided those things are parallelizable of course) even if the old things are limited by Amdahl's Law.

7.4.3 Comparing Amdahl's Law and Moore's Law

There is another "Law" relevant to computing speed and the number of processors available on a chip. Moore's Law, again named after its inventor, Gordon Moore, states that the number of transistors per unit area on a chip doubles roughly every 18 months. That increased transistor density used to lead to faster processors; now it is leading to more processors.

Moore's Law is an observation about the semiconductor industry that has held for decades. The fact that it has held for so long is entirely about empirical evidence — people look at the chips that are sold and see that they obey this law. Actually, for many years it has been a self-fulfilling prophecy: chip manufacturers expect themselves to continue Moore's Law and they find a way to achieve technological innovation at this pace. There is no inherent mathematical theorem underlying it. Yet we expect the number of processors

to increase exponentially for the foreseeable future.

On the other hand, Amdahl's Law is an irrefutable fact of algebra.

7.5 Fancier Fork-Join Algorithms: Prefix, Pack, Sort

This section presents a few more sophisticated parallel algorithms. The intention is to demonstrate (a) sometimes problems that seem inherently sequential turn out to have efficient parallel algorithms, (b) we can use parallel-algorithm techniques as building blocks for other larger parallel algorithms, and (c) we can use asymptotic complexity to help decide when one parallel algorithm is better than another. The study of parallel algorithms could take an entire course, so we will pick just a few examples that represent some of the many key parallel-programming patterns.

As is common when studying algorithms, we will not show full Java implementations. It should be clear at this point that one could code up the algorithms using the ForkJoin framework even if it may not be entirely easy to implement more sophisticated techniques.

7.5.1 Parallel-Prefix Sum

Consider this problem: Given an array of n integers `input`, produce an array of n integers `output` where `output[i]` is the sum of the first `i` elements of `input`. In other words, we are computing the sum of *every* prefix of the input array and returning all the results. This is called the *prefix-sum problem*.[6] Figure 7.8 shows an example input and output. A $\Theta(n)$ sequential solution is trivial:

```
int[] prefix_sum(int[] input) {
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1] + input[i];
```

[6]It is common to distinguish the inclusive-sum (the first `i` elements) from the exclusive-sum (the first `i-1` elements); we will assume inclusive sums are desired.

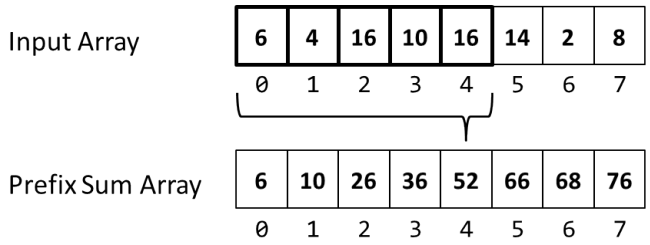


Figure (7.8) Example input and output for computing a prefix sum. Notice the sum of the first 5 elements is 52.

```
  return output;
}
```

It is not at all obvious that a good parallel algorithm, say, one with $\Theta(\log n)$ span, exists. After all, it seems we need `output[i-1]` to compute `output[i]`. If so, the span will be $\Theta(n)$. Just as a parallel reduction uses a totally different algorithm than the straightforward sequential approach, there is also an efficient parallel algorithm for the prefix-sum problem. Like many clever data structures and algorithms, it is not something most people are likely to discover on their own, but it is a useful technique to know.

The algorithm works in two passes. We will call the first pass the "up" pass because it builds a binary tree from bottom to top. We first describe the resulting tree and then explain how it can be produced via a fork-join computation. Figure 7.9 shows an example.

- Every node holds the sum of the integers for some range of the `input` array.
- The root of the tree holds the sum for the entire range $[0, n)$.[7]
- A node's left child holds the sum for the left half of the node's range and the node's right child holds the sum for the right half of the node's range. For example, the root's left child is for the range $[0, n/2)$ and the root's right child is for the range $[n/2, n)$.

[7]As before, we describe ranges as including their left end but excluding their right end.

- Conceptually, the leaves of the tree hold the sum for one-element ranges. So there are n leaves. In practice, we would use a sequential cut-off and have the leaves store the sum for a range of, say, approximately 500 elements.

To build this tree — and we do mean here to build the actual tree data-structure[8] because we need it in the second pass — we can use a straightforward fork-join computation:

- The overall goal is to produce the node for the range $[0, n)$.
- To build the node for the range $[x, y)$:
  - If $x == y - 1$ (or more generally $y - x$ is below the sequential cut-off), produce a node holding `input[x]` (or more generally the sum of the range $[x, y)$ computed sequentially).
  - Else recursively in parallel build the nodes for $[x, (x + y)/2)$ and $[(x + y)/2, y)$. Make these the left and right children of the result node. Add their answers together for the result node's sum.

In short, the result of the divide-and-conquer is a tree node and the way we "combine results" is to use the two recursive results as the subtrees. So we build the tree "bottom-up," creating larger subtrees from as we return from each level of the recursion. Figure 7.9 shows an example of this bottom-up process, where each node stores the range it stores the sum for and the corresponding sum. The "fromleft" field is blank — we use it in the second pass.

Convince yourself this algorithm is $\Theta(n)$ work and $\Theta(\log n)$ span.

Now we are ready for the second pass called the "down" pass, where we use this tree to compute the prefix-sum. The essential trick is that we process the tree from top to

[8] As a side-note, if you have seen an array-based representation of a complete tree, for example with a binary-heap representation of a priority queue, then notice that *if* the array length is a power of two, then the tree we build is also complete and therefore amenable to a compact array representation. The length of the array needs to be $2n - 1$ (or less with a sequential cut-off). If the array length is not a power of two and we still want a compact array, then we can either act as though the array length is the next larger power of two or use a more sophisticated rule for how to divide subranges so that we always build a complete tree.

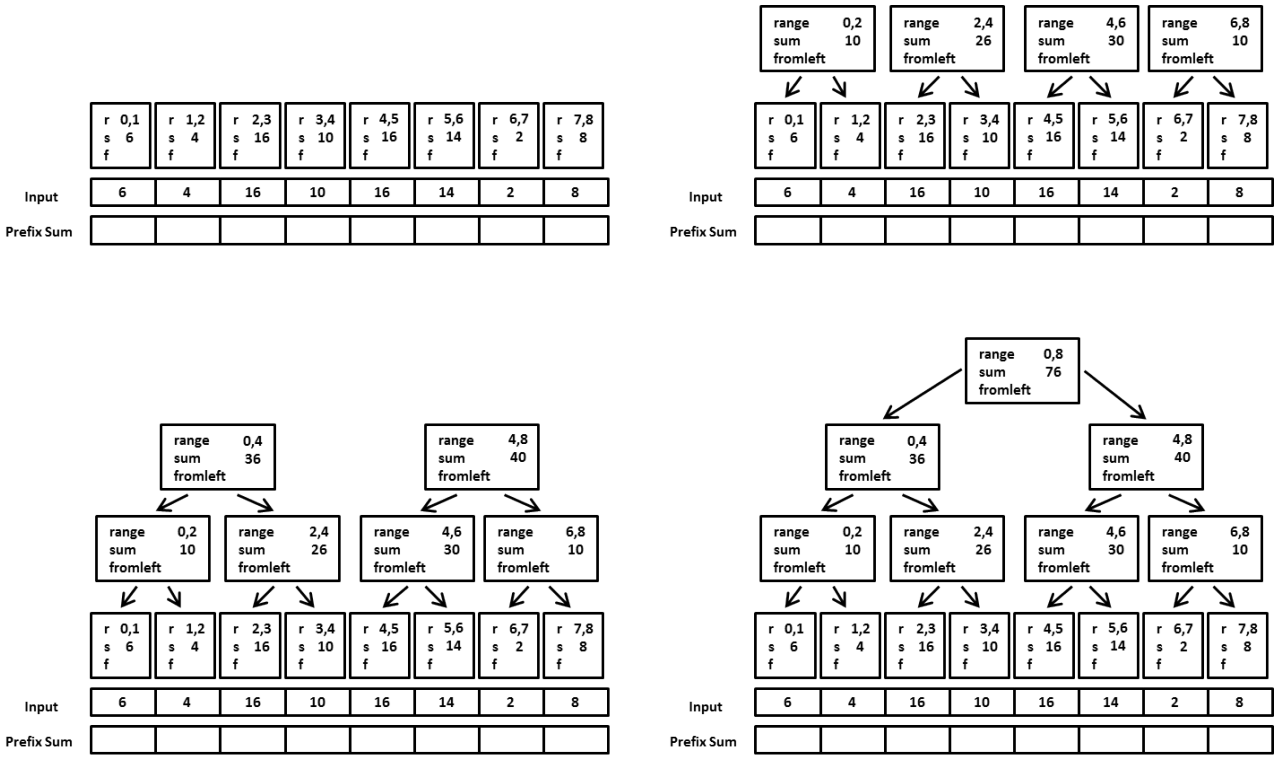


Figure (7.9) Example of the first pass of the parallel prefix-sum algorithm: the overall result (bottom-right) is a binary tree where each node holds the sum of a range of elements of the input. Each node holds the index range for which it holds the sum (two numbers for the two endpoints) and the sum of that range. At the lowest level, we write r for range and s for sum just for formatting purposes. The fromleft field is used in the second pass. We can build this tree bottom-up with $\Theta(n)$ work and $\Theta(\log n)$ span because a node's sum is just the sum of its children.

bottom, *passing "down" as an argument the sum of the array indices to the left of the node.* Figure 7.10 shows an example. Here are the details:

- The argument passed to the root is 0. This is because there are no numbers to the left of the range $[0, n)$ so their sum is 0.

- The argument passed to a node's left child is the same argument passed to the node. This is because the sum of numbers to the left of the range $[x, (x+y)/2)$ is the sum of numbers to the left of the range $[x, y)$.

- The argument passed to a node's right child is the argument passed to the node *plus* the sum stored at the node's left child. This is because the sum of numbers to the left of the range $[(x+y)/2, y)$ is the sum to the left of x plus the sum of the range $[x, (x+y)/2)$. This is why we stored these sums in the up pass!

When we reach a leaf, we have exactly what we need: `output[i]` is `input[i]` plus the value passed down to the i^{th} leaf. Convincing yourself this algorithm is correct will likely require working through Figure 7.10 in detail or creating your own short example while drawing the binary tree.

This second pass is also amenable to a parallel fork-join computation. When we create a subproblem, we just need the value being passed down and the node it is being passed to. We just start with a value of 0 and the root of the tree. This pass, like the first one, is $\Theta(n)$ work and $\Theta(\log n)$ span. So the algorithm overall is $\Theta(n)$ work and $\Theta(\log n)$ span. It is *asymptotically* no more expensive than computing just the sum of the whole array. The parallel-prefix problem, surprisingly, has a solution with exponential parallelism!

Perhaps the prefix-sum problem is not particularly interesting. But just as our original sum-an-array problem exemplified the parallel-reduction pattern, the prefix-sum problem exemplifies the more general parallel-prefix pattern. Here are two other general problems that can be solved the same way as the prefix-sum problem; you can probably think of more.

- Let `output[i]` be the minimum (or maximum) of all elements to the left of `i`.

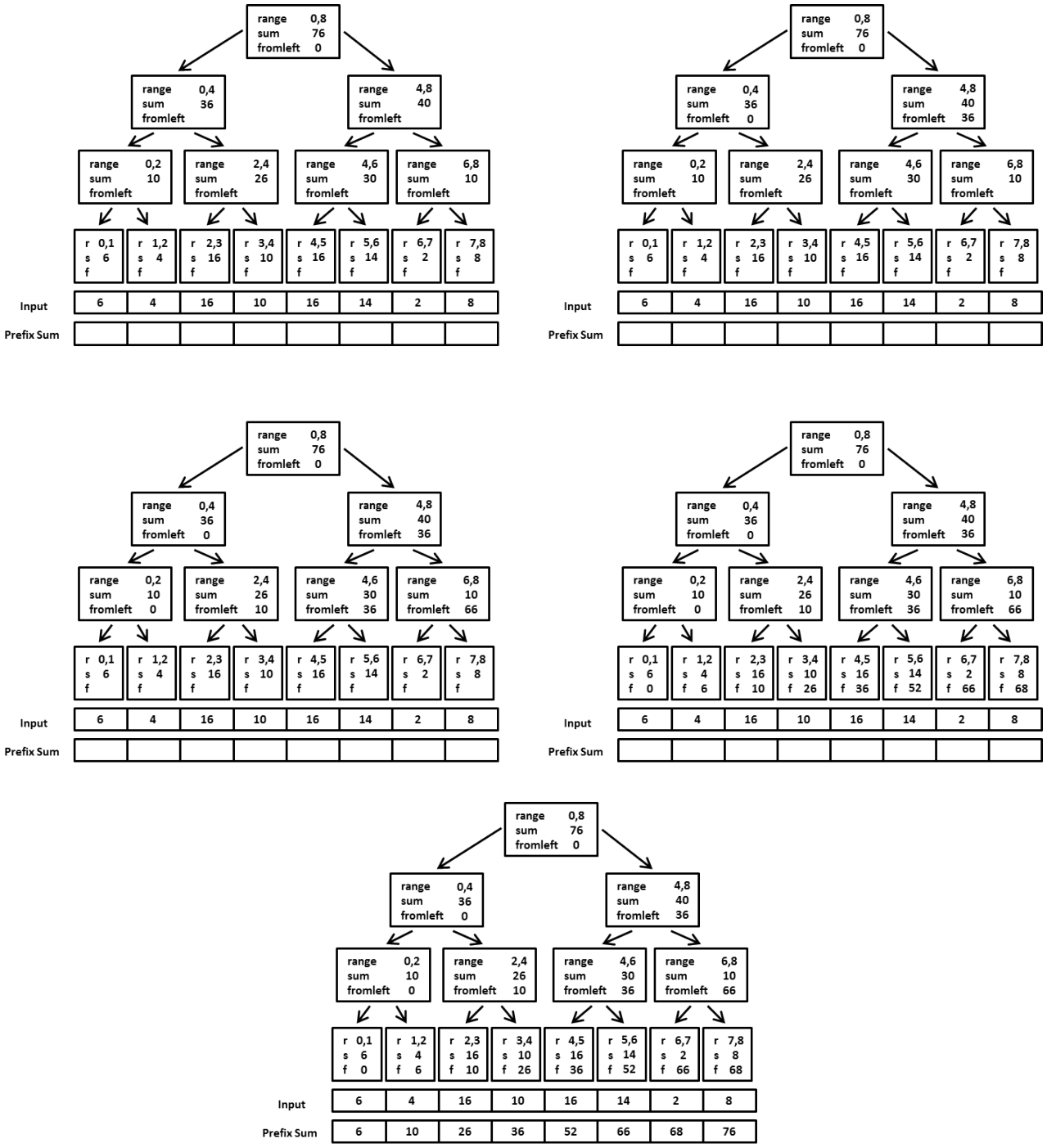


Figure (7.10) Example of the second pass of the parallel prefix-sum algorithm. Starting with the result of the first pass and a "fromleft" value at the root of 0, we proceed down the tree filling in fromleft fields in parallel, propagating the same fromleft value to the left-child and the fromleft value plus the left-child's sum to the right-value. At the leaves, the fromleft value plus the (1-element) sum is precisely the correct prefix-sum value. This pass is $\Theta(n)$ work and $\Theta(\log n)$ span.

- Let `output[i]` be a count of how many elements to the left of `i` satisfy some property.

Moreover, many parallel algorithms for problems that are not "obviously parallel" use a parallel-prefix computation as a helper method. It seems to be "the trick" that comes up over and over again to make things parallel. Section 7.5.2 gives an example, developing an algorithm on top of parallel-prefix sum. We will then use *that* algorithm to implement a parallel variant of quicksort.

7.5.2 Pack

This section develops a parallel algorithm for this problem: Given an array `input`, produce an array `output` containing only those elements of `input` that satisfy some property, and in the same order they appear in `input`. For example if the property is, "greater than 10" and `input` is {`17,4,6,8,11,5,13,19,0,24`}, then `output` is {`17,11,13,19,24`}. Notice the length of `output` is unknown in advance but never longer than `input`. A $\Theta(n)$ sequential solution using our "greater than 10" example is:

```
int[] greaterThanTen(int[] input) {
   int count = 0;
   for(int i=0; i < input.length; i++)
     if(input[i] > 10)
       count++;
   int[] output = new int[count];
   int index = 0;
   for(int i=0; i < input.length; i++)
     if(input[i] > 10)
       output[index++] = input[i];
   assert(index==count);
   return output;
}
```

Writing a generic version is really no more difficult; as in Section 7.3.5 it amounts to a judicious use of generics and higher-order programming. In general, let us call this pattern a *pack* operation, adding it to our patterns of maps, reduces, and prefixes. However, the term *pack* is not standard, nor is there a common term to our knowledge. *Filter* is also descriptive, but underemphasizes that the order is preserved.

This problem looks difficult to solve with effective parallelism. Finding which elements should be part of the output is a trivial map operation, but knowing what output index to use for each element requires knowing how many elements to the left also are greater than 10. But that is exactly what a prefix computation can do!

We can describe an efficient parallel pack algorithm almost entirely in terms of helper methods using patterns we already know. For simplicity, we describe the algorithm in terms of 3 steps, each of which is $\Theta(n)$ work and $\Theta(\log n)$ span; in practice it is straightforward to do the first two steps together in one fork-join computation.

1. Perform a parallel map to produce a *bit vector* where a 1 indicates the corresponding input element is greater than 10. So for {`17,4,6,8,11,5,13,19,0,24`} this step produces {`1,0,0,0,1,0,1,1,0,1`}.

2. Perform a parallel prefix sum on the bit vector produced in the previous step. Continuing our example produces {`1,1,1,1,2,2,3,4,4,5`}.

3. The array produced in the previous step provides the information that a final parallel map needs to produce the packed output array. In pseudocode, calling the result of step (1) `bitvector` and the result of step (2) `bitsum`:

```
int output_length = bitsum[bitsum.length-1];
int[] output = new int[output_length];
FORALL(int i=0; i < input.length; i++) {
   if(bitvector[i]==1)
     output[bitsum[i]-1] = input[i];
}
```

Note it is also possible to do step (3) using only `bitsum` and not `bitvector`, which would allow step (2) to do the prefix sum *in place*, updating the `bitvector` array. In either case, each iteration of the FORALL loop either does not write anything or writes to a different element of `output` than every other iteration.

Just as prefix sum was surprisingly useful for pack, pack turns out to be surprisingly useful for more important algorithms, including a parallel variant of quicksort...

7.5.3 Parallel Quicksort

Recall that sequential quicksort is an in-place sorting algorithm with $O(n \log n)$ best- and expected-case running time. It works as follows:

1. Pick a pivot element ($O(1)$)
2. Partition the data into: ($O(n)$)
   (a) Elements less than the pivot
   (b) The pivot
   (c) Elements greater than the pivot
3. Recursively sort the elements less than the pivot
4. Recursively sort the elements greater than the pivot

Let's assume for simplicity the partition is roughly balanced, so the two recursive calls solve problems of approximately half the size. If we write $R(n)$ for the running time of a problem of size n, then, except for the base cases of the recursion which finish in $O(1)$ time, we have $R(n) = O(n) + 2R(n/2)$ due to the $O(n)$ partition and two problems of half the size.[9] It turns out that when the running time is $R(n) = O(n) + 2R(n/2)$, this works out to be $O(n \log n)$, but we do not show the derivation of this fact here. Moreover, if pivots are chosen

[9]The more common notation is $T(n)$ instead of $R(n)$, but we will use $R(n)$ to avoid confusion with work T_1 and span T_∞.

randomly, the expected running time remains $O(n \log n)$. For simplicity, the analysis below will continue to assume the chosen pivot is magically exactly the median. As with sequential quicksort, the expected-time asymptotic bounds are the same if the pivot is chosen uniformly at random, but the proper analysis under this assumption is more mathematically intricate.

How should we parallelize this algorithm? The first obvious step is to perform steps (3) and (4) in parallel. This has no effect on the work, but changes the running time for the span to $R(n) = O(n) + 1R(n/2)$ (because we can solve the two problems of half the size simultaneously), which works out to be $O(n)$. Therefore, the parallelism, T_1/T_∞, is $O(n \log n)/O(n)$, i.e., $O(\log n)$. While this is an improvement, it is a far cry from the exponential parallelism we have seen for algorithms up to this point. Concretely, it suggests an infinite number of processors would sort billions of elements dozens of times faster than one processor. This is okay but underwhelming.

To do better, we need to parallelize the step that produces the partition. The sequential partitioning algorithm uses clever swapping of data elements to perform the in-place sort. To parallelize it, we will sacrifice the in-place property. Given Amdahl's Law, this is likely a good trade-off: use extra space to achieve additional parallelism. All we will need is one more array of the same length as the input array.

To partition into our new extra array, all we need are two pack operations: one into the left side of the array and one into the right side. In particular:

- Pack all elements less than the pivot into the left side of the array: If x elements are less than the pivot, put this data at positions 0 to $x - 1$.

- Pack all elements greater than the pivot into the right side of the array: If x elements are greater than the pivot, put this data at positions $n - x$ to $n - 1$.

The first step is exactly like the pack operation we saw earlier. The second step just works down from the end of the array instead of up from the beginning of the array. After performing both steps, there is one spot left for the pivot between the two partitions. Figure 7.11 shows an example of using two pack operations to partition in parallel.

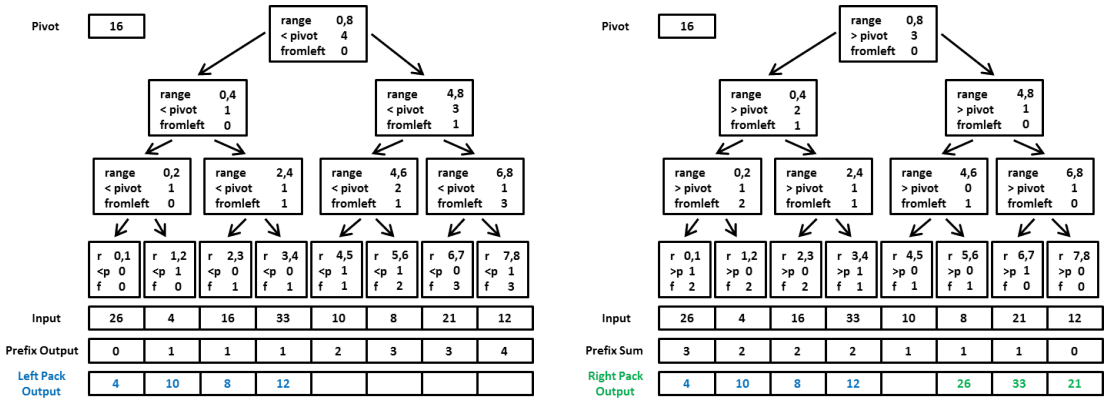


Figure (7.11) Example using two pack operations to partition in parallel

Each of the two pack operations is $O(n)$ work and $O(\log n)$ span. The fact that we look at each element twice, once to decide if it is less than the pivot and once to decide if it is greater than the pivot is only a constant factor more work. Also note that the two pack operations can be performed in parallel, though this is unnecessary for the asymptotic bound.

After completing the partition, we continue with the usual recursive sorting (in parallel) of the two sides. Rather than create another auxiliary array, the next step of the recursion can reuse the original array. Sequential mergesort often uses this same space-reuse trick.

Let us now re-analyze the asymptotic complexity of parallel quicksort using our parallel (but not in-place) partition and, for simplicity, assuming pivots always divide problems exactly in half. The work is still $R(n) = O(n) + 2R(n/2) = O(n \log n)$ where the $O(n)$ now includes the two pack operations. The span is now $R(n) = O(\log n) + 1R(n/2)$ because the span for the pack operations is $O(\log n)$. This turns out to be (again, not showing the derivation) $O(\log^2 n)$, not as good as $O(\log n)$, but much better than the $O(n)$ (in fact, $\Theta(n)$) we had with a sequential partition. Hence the available parallelism is proportional to $n \log n / \log^2 n = n / \log n$, an exponential speed-up.

7.5.4 Parallel Mergesort

As a final parallel algorithm, we develop a parallel version of mergesort. As with quicksort, achieving a little parallelism is trivial, but achieving a lot of parallelism requires much more cleverness. We first recall the sequential mergesort algorithm, which always has running time $O(n \log n)$ and is not in-place:

1. Recursively sort the left half and right half of the input.

2. Merge the sorted results into a new sorted array by repeatedly moving the smallest not-yet-moved element into the new array.

The running time for this algorithm is $R(n) = 2R(n/2) + O(n)$ because there are two subproblems of half the size and the merging is $O(n)$ using a single loop that progresses through the two sorted recursive results.[10] This is the same running time as sequential quicksort, and it works out to $O(n \log n)$.

The trivial first parallelization step is to do the two recursive sorts in parallel. Exactly like with parallel quicksort, this has no effect on the work and reduces the span to $R(n) = 1R(n/2) + O(n)$, which is $O(n)$. Hence the parallelism is $O(\log n)$.

To do better, we need to parallelize the merge operation. Our algorithm will take two sorted subarrays of length x and y and merge them. In sequential mergesort, the two lengths are equal (or almost equal when merging an odd number of elements), but as we will see, our recursive parallel merging will create subproblems that may need to merge arrays of different lengths. The algorithm is:[11]

- Determine the median element of the larger array. This is just the element in the middle of its range, so this operation is $O(1)$.

[10] As with the analysis for parallel quicksort, we are writing $R(n)$ instead of the more common $T(n)$ just to avoid confusion with our notation for work and span.

[11] The base case, ignoring a sequential cut-off, is when x and y are both ≤ 1, in which case merging is trivially $O(1)$.

- Use binary search to find the position j in the smaller array such that all elements to the left of j are less than the larger array's median. Binary search is $O(\log m)$ where m is the length of the smaller array.

- Recursively merge the left half of the larger array with positions 0 to j of the smaller array.

- Recursively merge the right half of the larger array with the rest of the smaller array.

The total number of elements this algorithm merges is $x + y$, which we will call n. The first two steps are $O(\log n)$ since n is greater than the length of the array on which we do the binary search. That leaves the two subproblems, which are not necessarily of size $n/2$. That best-case scenario occurs when the binary search ends up in the middle of the smaller array. The worst-case scenario is when it ends up at one extreme, i.e., all elements of the smaller array are less than the median of the larger array or all elements of the smaller array are greater than the median of the larger array.

But we now argue that the worst-case scenario is not that bad. The larger array has at least $n/2$ elements — otherwise it would not be the larger array. And we always split the larger array's elements in half for the recursive subproblems. So each subproblem has at least $n/4$ (half of $n/2$) elements. So the worst-case split is $n/4$ and $3n/4$. That turns out to be "good enough" for a large amount of parallelism for the merge. The full analysis of the recursive algorithm is a bit intricate, so we just mention a few salient facts here.

Because the worst-case split is $n/4$ and $3n/4$, the worst-case span is $R(n) = R(3n/4) + O(\log n)$: the $R(n/4)$ does not appear because it can be done in parallel with the $R(3n/4)$ and is expected to finish first (and the $O(\log n)$ is for the binary search). $R(n) = R(3n/4) + O(\log n)$ works out to be $O(\log^2 n)$ (which is not obvious and we do not show the derivation here). The work is $R(n) = R(3n/4) + R(n/4) + O(\log n)$, which works out to be $O(n)$ (again omitting the derivation).

Recall this analysis was just for the merging step. Adding $O(\log^2 n)$ span and $O(n)$ work for merging back into the overall mergesort algorithm, we get a span of $R(n) = 1R(n/2) +$

$O(\log^2 n)$, which is $O(\log^3 N)$, and a work of $R(n) = 2R(n/2) + O(n)$, which is $O(n \log n)$. While the span and resulting parallelism is $O(\log n)$ worse than for parallel quicksort, it is a worst-case bound compared to quicksort's expected case.

REFERENCES

[1] D. Grossman and R. E. Anderson, "Introducing parallelism and concurrency in the data structures course," in *43rd SIGCSE Technical Symposium on Computer Science Education*, Raleigh, NC, Mar. 2012.