A Sophomoric[*] Introduction to Shared-Memory Parallelism and Concurrency

Dan Grossman
Adapted to C++ by Steve Wolfman

Version of March 30, 2012

Contents

---

[*]As in intended for second-year students, not as in immaturely pretentious

1

1 Meta-Introduction: An Instructor's View of These Notes

1.1 Where This Material Fits in a Changing Curriculum

These notes teach parallelism and concurrency as part of an advanced sophomore-level data-structures course – the course that covers asymptotic complexity, balanced trees, hash tables, graph algorithms, sorting, etc.

Why parallelism and concurrency should be taught early:

Parallelism and concurrency are increasingly important topics in computer science and engineering. Traditionally, most undergraduates learned rather little about these topics and did so rather late in the curriculum: Senior-level operating-systems courses cover threads, scheduling, and synchronization. Early hardware courses have circuits and functional units with parallel parts. Advanced architecture courses discuss cache coherence. Electives might cover parallel algorithms or use distributed computing to solve embarrassingly parallel tasks.

Little of this scattered material emphasizes the essential concepts of parallelism and concurrency — and certainly not in a central place such that subsequent courses can rely on it. These days, most desktop and laptop computers have multiple cores. Modern high-level languages have threads built into them and standard libraries use threads (e.g., Java's Swing library for GUIs). It no longer seems reasonable to bring up threads "as needed" or delay until an operating-systems course that should be focusing on operating systems. There is no reason to introduce threads first in C or assembly when all the usual conveniences of high-level languages for introducing core concepts apply.

Why parallelism and concurrency should not be taught too early:

Conversely, it is tempting to introduce threads "from day one" in introductory programming courses before students learn "sequential habits." I suspect this approach is infeasible in most curricula. For example, it may make little sense in programs where most students in introductory courses do not end up majoring in computer science. "Messing with intro" is a high-risk endeavor, and introductory courses are already packed with essential concepts like variables, functions, arrays, linked structures, etc. There is probably no room.

So put it in the data-structures course:

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

- There is room: We 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 will 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 concurrency 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 C++ in the course. The C++11 standard makes threads, concurrency constructs, and the shared memory model used in these notes standard and available across platforms. The freely available OpenMP and Cilk frameworks support fork-join parallelism.

On the other hand, instructors wishing to introduce message passing or distributed computation will have to consider whether it makes sense in this course. I focus on shared memory, only mentioning other models. I do not claim shared memory is "better" (that's an endless argument on which I have no firm opinion), only that it is an important model and a good one to start with pedagogically. I also do not emphasize asynchrony and masking I/O latency. Such topics are probably better covered in a course on systems programming, though one could add them to these notes.

1.2 Six Theses On A Successful Approach to this Material

In summary, these notes rest 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 (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 hard 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.)

If you strongly disagree with these theses, you will probably not like these notes — but you might still try them.

1.3 How to Use These Notes — And Improve Them

These notes were originally written for CSE332 at the University of Washington (http://www.cs.washington.edu/education/courses/cse332). They account for 3 weeks of a required 10-week course (the University uses a quarter system). Alongside these notes are PowerPoint slides, homework assignments, and a programming project. In fact, these notes were the last aspect to be written — the first edition of the course went great without them and students reported parallelism to be their favorite aspect of the course.

Surely these notes have errors and explanations that could be improved. *Please* let me know of any problems you find. I am a perfectionist: if you find a typo I would like to know. Ideally you would first check that the most recent version of the notes does not already have the problem fixed.

As for permission, do with these notes whatever you like. Seriously: "steal these notes." The LaTeX sources are available so you can modify them however you like. I would *like* to know if you are using these notes and how. It motivates me to improve them and, frankly, it's not bad for my ego. Constructive criticism is also welcome. That said, I don't expect any thoughtful instructor to agree completely with me on what to cover and how to cover it.

Contact me at the email djg and then the at-sign and then cs.washington.edu. The current home for these notes and related materials is http://www.cs.washington.edu/homes/djg/teachingMaterials. Students are more than welcome to contact me: who better to let me know where these notes could be improved.

1.4 Acknowledgments

2 Introduction

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 most of the programs you have written have been sequential: Execution starts at the beginning of the program and proceeds one assignment / call / return / arithmetic operation at a time.[1]

---

[1]If you've written functional programs—such as in Scheme—this "one-at-a-time" model may seem wrong, and indeed it is! We can often automatically or trivially convert functional programs to exploit parallelism, and we'll discuss using a functional approach as good style for parallel programming later on.

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. These notes 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

- How to coordinate access to shared objects so that multiple threads using the same data do not produce the wrong answer

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 won't be able to 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 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 these notes cover 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.

2.2 Parallelism vs. Concurrency

These notes are organized around a fundamental distinction between *parallelism* and *concurrency*. Unfortunately, the way we define these terms is not entirely standard, so you should not assume that everyone uses these terms as we will. Nonetheless, most computer scientists agree that this distinction is important.

Parallel programming 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. 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's 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.

Concurrent programming is about correctly and efficiently controlling access by multiple threads to shared resources.

As a canonical example, suppose we have a dictionary implemented as a hashtable with operations `insert`, `lookup`, and `delete`. Suppose that inserting an item already in the table is supposed to 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 suppose different threads use the *same* hashtable, potentially at the same time. Suppose two threads even try to `insert` the same key at the same time. What might 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 C++, 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 whole problem 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.

Comparing Parallelism and Concurrency

We have emphasized here how parallel programming and concurrent programming 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.

In practice, the distinction between parallelism and concurrency is not absolute. Many programs have aspects of each. 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. 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. You probably found it easier to understand how to use parallelism to add up array elements than understanding why the while-loop for checking the oven was wrong. (And if you still don't understand the latter, don't worry, later sections will explain similar examples line-by-line.) So we will start with parallelism (Sections 3, 4, 5), getting comfortable with multiple things happening at once. Then we will switch our focus to concurrency (Sections 6, 7, 8, 9, 10) and shared resources (using memory instead of ovens), learn many of the subtle problems that arise, and present programming guidelines to avoid them.

2.3 Basic Threads and Shared Memory

Before writing any parallel or concurrent 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 programming constructs you know are for sequential programs, which do only one thing at once. Before showing any C++ specifics, we need to explain the *programming model*.

The model we will assume is *explicit threads* with *shared memory*[2]. 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/concurrent programming, but it is the only one we will use. The next section briefly mentions other models that a full course on parallel/concurrent 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 C++ runtime, 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 C++ 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's helpful to start by enumerating the key pieces that a *sequential* program has *while it is running*:

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. Global and static variables.

4. Heap-Allocated Data. Memory allocated by invoking `new` is not part of the call stack. We call the memory that holds all such data the *heap*. This use of the word "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 global or static variables, and the programmer is responsible for managing the lifetime of memory allocated on the heap (using `delete`).

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 global and static variables and one heap. (Warning: in C++, one thread can access another's call stack if pointers or references to these locations leak between them!)

---

[2] Specifically, Concurrent-Read Exclusive-Write (CREW) memory. It's OK for many threads to read a piece of memory at the same time, but it's only OK for one thread to write if no other thread is either reading or writing at the same time.

- 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.

- 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 shared memory works. 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 heap-allocated objects *could* be shared among threads, most are not. In fact, just as having global variables is often poor style, having lots of objects shared among threads is often poor style. We may need some shared objects to support communication among the threads, but we'll try to avoid them as much as possible. The C++ threading libraries are designed to help with that, e.g., supporting passing information into a new thread without any persistent sharing.

We finish this section with some specifics for exactly how to create a new thread in C++. In addition to creating threads, we will need other language constructs for coordinating them. For example, for one thread to read the result that another thread wrote as its answer, the reader often needs to know that the writer is done. We will present such primitives as we need them. The details vary in different languages. In fact, because C++11 deferred official support for the fork-join parallelism style that we use later in these notes, we will often refer to a different C++ library with slightly different specifics.

To create a new thread in C++, instantiate a `std::thread` object, passing it a function to use as the starting point for that thread (like its `main`) and any values you want to pass to the thread:

Here is a complete (but slightly erroneous) example of a useless C++ program that creates 20 more threads on top of its main thread:

```
// Compile with the C++11, roughly available with -std=c++0x in recent g++ compilers.
#include <thread>
#include <iostream>
#include <vector>
#include <ios> // for sync_with_stdio


void hiBye(int i) {
  std::cout << "Thread " << i << " says hi!" << std::endl;
  std::cout << "Thread " << i << " says bye!" << std::endl;
}

int main() {
  // Makes cin/cout thread-safe without extra care.
  std::ios_base::sync_with_stdio(true);

  std::vector<std::thread> threads;
  for (int i = 1; i <= 20; i++)
    threads.push_back(std::thread(&hiBye, i));

  return 0;
}
```

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

```
    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 C++ runtime 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.

Worse yet, you may see parts of lines interspersed with each other. For example:

```
Thread Thread 5 says bye!
4 says hi!
```

Nothing stops one thread interrupting another between `<<` operators. In fact, without the `sync_with_stdio(true)` command, this code can cause an error because `cout` in one thread isn't required to "play nice" (avoid *data races* discussed in Section 7) with `cout` in another thread. Even with this command, C++ doesn't even guarantee that you won't see interleaving *below* the level of the stream operators, e.g., one character from one thread followed by one character from the next. In general, concurrency introduces new questions about how code should and does behave.

So is any possible ordering of the 40 output lines possible? No. Each thread still runs sequentially. So, you will always see the `hi`—or at least `h` and later `i`—printed by thread 13 before seeing the `bye` printed by the same thread, even though there may be other threads' messages in between.

Finally, you should see an error message at the bottom of your program such as:

```
terminate called without an active exception
Abort
```

The error indicates that a thread was forcibly terminated when the thread object—and the vector containing it—was destroyed at the end of `main`. How do we stop threads without this error? We'll call the `join` method, which forces the current thread to wait until the thread it is `join`ing finishes. That gives us just what we need to avoid the error *and* eliminate the danger of interspersed lines:

```
for (int i = 1; i <= 20; i++) {
  threads.push_back(std::thread(&hiBye, i));
  threads.back().join();
}
```

This code prints the 40 lines in order with no error. Is that a good thing? Not really. The main thread sends each new thread off to do its work and then waits for that thread to finish before proceeding. In other words, only one thing is really happening at a time. We may as well have had the main thread write all the messages! Instead, we'll use:

```
for (int i = 1; i <= 20; i++)
  threads.push_back(std::thread(&hiBye, i));
for (int i = 0; i < threads.size(); i++)
  threads[i].join();
```

Now we get no error message and interspersed lines. The threads are working in parallel. There's much more that can be done with C++11 to support threads. See Appendix A for a tiny taste.

2.4 Other Models

While these notes focus on using threads and shared memory to handle parallelism and concurrency, it would be misleading to suggest that this is the only model for parallel/concurrent 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 Section 7.

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.

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 runtime keeps track of automatically.

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.

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 C++ to achieve the same idea requires a bit more work (Section 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 3.2) and show how to use recursive divide-and-conquer instead (Section 3.3). Because C++'s threads are not (yet) engineered for this style of programming, we switch to the OpenMP framework which is designed for our needs (Section 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 3.5) as well as data structures other than arrays (Section 3.6).

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 len) {
```

```
    int ans = 0;
    for(int i=0; i < len; 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 can do 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 any of 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.[3]

```
int sum(int arr[], int len) {
  results = new int[4];
  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) {
    int result = 0;
    for(j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

Fortunately, both the OpenMP and Cilk frameworks provide a FORALL construct.[4] However, it's worth understanding how we could accomplish this with bare threads, particularly since our naïve approach here will prepare us for a more savvy approach later. We can encode this programming pattern explicitly using threads as follows:

---

[3]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.

[4]In OpenMP, precede the loop by the line #pragma omp for plus some extra syntax to ensure the right memory is shared across threads.

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 the appropriate slot of the `results` array.

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

3. Combine the results by reading the answers out of the the array slots set in step 1.

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

Here is our WRONG attempt:

```
int sumRange(int * result, int arr[], int lo, int hi) {
  *result = 0;
  for (int j = lo; j < hi; j++) {
    *result += arr[j];
  }
}

int sum(int arr[], int len) {
  int results[4];
  std::vector<std::thread> workers;
  for (int i = 0; i < 4; i++)
    workers.push_back(std::thread(&sumRange, &results[i], arr,
                                  (i*len)/4, ((i+1)*len)/4));

  int result = 0;
  for (int i = 0; i < 4; i++) {
    result += results[0];
  }

  return result;
}
```

The code above gets most of the pieces right. The `sum` function creates four threads. Each thread takes as arguments the data it needs to do its work: 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.

Each thread also *shares* its entry in the `results` array with the main thread so that it can communicate results back. (Using a reference parameter rather than a pointer *will not work* here because of the way parameters are passed to the `thread` constructor. Try it and see!) This is our first real use of shared memory. In particular, for the thread created on the `i`th iteration of the for-loop in `sum`, the parameter `result` points to `results[i]`. The main thread can sum the entries in the `results` array to produce the final answer. Having two ways to refer to the same memory location is called *aliasing*. Aliasing is often difficult to reason about—read: causes many bugs—but it is the way we communicate in shared-memory programs!

The bug in this code has to do with synchronization and, as in our first example, C++ will help us find it by indicating that at least one thread was forcibly terminated rather than being `joined`. The main thread does not wait for the helper threads to finish before it sums its results. Remember that the `thread` constructor 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 answer. (The answer is usually too small... but surprisingly not always. Can it really be *too large*? Yes. Be afraid and see Section 7.)

We need to delay the second for-loop until the helper threads are done using the same technique as in our first example:

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

```
    int results[4];
    std::vector<std::thread> workers;
    for (int i = 0; i < 4; i++)
      workers.push_back(std::thread(&sumRange, arr, &results[i],
                                    (i*len)/4, ((i+1)*len)/4));

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

    int result = 0;
    for (int i = 0; i < 4; i++)
      result += results[0];

    return result;
}
```

Again, the `join` method forces the calling thread to wait until the thread being `join`ed finishes its work. Notice that the main thread calls `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 without 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 a superior solution. 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` function:

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

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

```
for(int i=0; i < 4; i++) {
    workers[i].join();
    result += results[i];
}
```

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 result—its shared memory—until at least that helper thread has terminated.

This example shows many of the key concepts of fork-join parallelism, but Section 3.2 will explain why it is a poor approach that can lead to suboptimal performance. Sections 3.3 and 3.4 will then present a similar but better approach.

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 suboptimal, likely leading 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 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 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. If 3 processors are available, our 4-thread program will take approximately twice as long as with 4 processors, but a 3-thread version would run only about 33% slower. 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 C++ 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:

```
int sum(int arr[], int len, int numParts) {
  int results[numParts];
  std::vector<std::thread> workers;
  for (int i = 0; i < numParts; i++)
    workers.push_back(std::thread(&sumRange, arr, &results[i],
                                  (i*len)/numParts, ((i+1)*len)/numParts));

  int result = 0;
  for (int i = 0; i < NUM_PARTS; i++) {
    workers[i].join();
    result += results[i];
  }

  return result;
}
```

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 C++ runtime 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 3.3 and 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 `len`/1000, which is $\Theta(n)$, results to combine. Combining the results with a sequential for-loop produces a $\Theta(n)$ 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. Although the details of threading in C++11 are implementation-specific, it is very unlikely that the thread support used was designed for small tasks like adding 1000 numbers. The threads will work and produce the correct answer, but the constant-factor overheads of creating each thread is far too large. A 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 an implementation of threads that is designed for this kind of fork/join programming.

3.3 Divide-And-Conquer Parallelism

This section presents the idea of divide-and-conquer parallelism using C++'s threads. Then Section 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
```

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 this tree is $\log n$ for a range of size n. With enough processors, the total 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 C++ 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 C++:

```cpp
void sumWorker(int * result, int arr[], int lo, int hi) {
  if (hi - lo == 1)
    *result = arr[lo];
  else {
```

```
    int leftSum, rightSum;
    std::thread left(&sumWorker, &leftSum, arr, lo, lo+(hi-lo)/2);
    std::thread right(&sumWorker, &rightSum, arr, lo+(hi-lo)/2, hi);
    left.join();
    right.join();
    *result = leftSum + rightSum;
  }
}

int sum(int arr[], int len) {
  int result;
  sumWorker(&result, arr, 0, len);
  return result;
}
```

Each thread creates two helper threads `left` and `right` and then waits for them to finish. Crucially, the call to `join` on `left` comes after creating `right`; otherwise, the algorithm would have no effective parallelism whatsoever. It would still produce the correct answer, but so would the original much simpler sequential program.

Notice that `sumWorker` is just a normal function. The "main" `sum` method calls it directly. As such, this is an ordinary function call like you have used since you started programming; the caller and callee are part of the same 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.

```
void sumWorker(int * result, int arr[], int lo, int hi) {
  const int SEQUENTIAL_CUTOFF = 1000;
  if (hi - lo < SEQUENTIAL_CUTOFF) {
    *result = 0;
    for (int i = lo; i < hi; i++)
      *result += arr[i];
  }
  else {
    int leftSum, rightSum;
    std::thread left(&sumWorker, &leftSum, arr, lo, lo+(hi-lo)/2);
    std::thread right(&sumWorker, &rightSum, arr, lo+(hi-lo)/2, hi);
    left.join();
    right.join();
    *result = leftSum + rightSum;
  }
}

int sum(int arr[], int len) {
  int result;
  sumWorker(&result, arr, 0, len);
  return result;
}
```

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 insertionsort for small arrays ($n < 50$ or so). Therefore, it is

17

common to have the recursive quicksort switch over to insertionsort 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 OpenMP 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 `sumWorker` function directly, which recall is just an ordinary method call unlike the "magic" thread constructor.

```
void sumWorker(int * result, int arr[], int lo, int hi) {
  const int SEQUENTIAL_CUTOFF = 1000;
  if (hi - lo < SEQUENTIAL_CUTOFF) {
    *result = 0;
    for (int i = lo; i < hi; i++)
      *result += arr[i];
  }
  else {
    int leftSum, rightSum;
    std::thread left(&sumWorker, &leftSum, arr, lo, lo+(hi-lo)/2);
    sumWorker(&rightSum, arr, lo+(hi-lo)/2, hi);
    left.join();
    *result = leftSum + rightSum;
  }
}


int sum(int arr[], int len) {
  int result;
  sumWorker(&result, arr, 0, len);
  return result;
}
```

The code above creates only one helper thread. It then does the right half of the work itself with a recursive call. 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.

Unfortunately, even with these optimizations, the code above will run poorly in practice, especially if given a large array. The implementation of C++'s threads (on most platforms) 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 C++ runtime 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 C++'s threads when we learn concurrency because the synchronization operations we will use work with C++'s threads (although the OpenMP framework also has synchronization operations).

3.4   The C++ OpenMP framework

Starting with version 3.0, the widely adopted OpenMP framework (for C, C++ and Fortran) includes facilities designed

exactly for the kind of fine-grained fork-join parallel computing these notes use. OpenMP supports lightweight threads (actually "tasks", a slightly different concept) that are small enough that even a million of them should not overwhelm the system. Good OpenMP implementations should also include a scheduler and run-time system with provably optimal expected-time guarantees, as described in Section 4. Similar libraries for other languages include Intel's Threading Building Blocks and CilkPlus, 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 C++ 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.

As long as you are using GCC 4.4 or newer, you should be able to compile OpenMP programs successfully using the -fopenmp flag (for both compilation and linking). The main web site for the library is http://openmp.org/wp/ with a handy C++ quick summary sheet at http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf. Note that you can trivially compile and run your application without threads by leaving out the -fopenmp flag, which can be very handy!

We now convert our finished program above into a version that uses OpenMP. As with our final "C++ threads"-based version, this uses a sequential cut-off and only one helper task at each recursive subdivision.

```
int sumWorker(int arr[], int lo, int hi) {
  static const int SEQUENTIAL_CUTOFF = 1000;
  if (hi - lo < SEQUENTIAL_CUTOFF) {
    int result = 0;
    for (int i = lo; i < hi; i++)
      result += arr[i];
    return result;
  }
  else {
    int leftSum, rightSum;

#pragma omp task untied shared(leftSum)
    leftSum = sumWorker(arr, lo, lo+(hi-lo)/2);

    rightSum = sumWorker(arr, lo+(hi-lo)/2, hi);
#pragma omp taskwait

    return leftSum + rightSum;
  }
}

int sum(int arr[], int len) {
  int result;
#pragma omp parallel
#pragma omp single
  result = sumWorker(arr, 0, len);
  return result;
}
```

Most differences from our code above are straightforward translations into the new framework. One larger, pleasant change is that sumWorker returns its result rather than taking a pointer to shared memory. We instead use OpenMP's shared clause to indicate that the underlying memory of the variable leftSum is shared with the newly created task.

Perhaps the most bizarre and confusing (and easy to forget) change that we make is beginning the sum function using the OpenMP compiler directive #pragma omp parallel with the OpenMP single compiler directive nested

directly inside of it. The `parallel` directive "kicks off" parallelism, spinning up an appropriately-sized pool of threads. The `single` directive constrains execution of its contained statement—`result = sumWorker(arr, 0, len);`—to a single thread. If we didn't use `parallel`, we'd get no parallelism. On the other hand, if we didn't use `single` inside of `parallel`, we'd perform the whole calculation many times, once for each actual thread OpenMP decided to allocate. Neither one is at all what we want! (Note that variables present when we reach a `parallel` directive default to being shared; so, we need not explicitly instruct OpenMP to share the variable `result`.)

Once we get past these issues, other changes include:

- `sum` uses `sumWorker`'s return value rather than passing a pointer, because of the change already described above.

- Under the `SEQUENTIAL_CUTOFF`, `sumWorker` declares and returns a local variable result rather than using memory passed in, again because of the change described above.

- We use the OpenMP compiler directive `task` to fork the left branch as its own task rather than the `std::thread` constructor.[5] Note that only the next line is within the `task`. If we want multiple lines to be affected, we can enclose them in braces `{...}`, just as within a `for` or `if` statement.

- As described above, we use the OpenMP `shared` clause and the return value of `sumWorker` rather than passing a pointer to the result.

- We use the OpenMP compiler directive `taskwait` rather than `join`. There's a subtle difference in that `join` waits on a particular thread whereas `taskwait` waits on *all* child tasks of the current task. In this example there is only one child task, but if we returned to the version where we fork a task for both the left and right part of the sum, we'd still use only one `taskwait`, and we'd have no way to wait instead on one "half" or the other.

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.

---

[5]If you do anything fancy with OpenMP, understand why the `untied` clause can cause you trouble, particularly if you use a `threadprivate` clause.

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:

```
#include <functional>

template <typename T>
T fold(T arr[], int len, std::function<T(T,T)> binop, T initialValue) {
  T ans = initialValue;
  for (int i = 0; i < len; i++) {
    ans = binop(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 operator `binop` to the current answer and the next element of the array.

Without any additional information about what `binop` 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 binary operation 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, $binop(a, binop(b,c))$ is the same as $binop(binop(a,b),c)$.[6] 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 OpenMP framework has a `reduction` clause 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 pattern, 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:

```
// result, arr1, and arr2 should all have length len.
void add(int result[], int arr1[], int arr2[], len) {
  FORALL(int i=0; i < arr1.length; i++)
    result[i] = arr1[i] + arr2[i];
}
```

Coding up this algorithm in OpenMP is straightforward: Each task in the parallel divide-and-conquer will have a reference to these arrays but will work on different portions of them. (Again, OpenMP supports doing this even more

---

[6]We can even remove the constraint that `binop` return the same type as the array elements. Instead, it can take some independent type—which may be the same as the array element type—and an array element and return that independent type.

straightforwardly with its `parallel for` directive, but we're looking at what it takes to implement such facilities efficiently.) 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 OpenMP framework or Hadoop to do the actual scheduling of the parallelism.

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.

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 OpenMP 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.

4.1 Work and Span

4.1.1 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) where:

- Nodes are pieces of work the program performs. Each node will be a constant, i.e., $O(1)$, 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.

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:
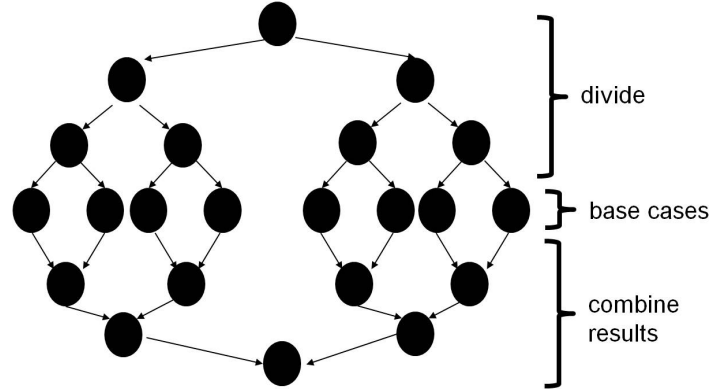
The root node represents the computation that divides the array into two equal halves. The bottom node represents the computation that adds together the two sums from the halves to produce the final answer. The base cases represent reading from a one-element range assuming no sequential cut-off. A sequential 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 $\log n$).

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

4.1.2 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.

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.

24

4.1.3 The OpenMP framework Bound

With a good implementation[7] and under some important assumptions we will describe below, algorithms written using the OpenMP framework, in particular the divide-and-conquer algorithms in these notes, have the following *expected* time bound:

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

The bound is *expected* because internally the framework uses randomness to make choices just like quicksort, in order to get a true expected-time guarantee, should use randomness to select a pivot. Because these notes do 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 tasks that *could* run next (they are not blocked waiting for some other task 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 tasks 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 tasks you create to do subproblems do approximately the same amount of work. Otherwise, if a task 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 tasks do a small but not tiny amount of work. This again helps with load balancing. How little is too little? We suggest aiming for approximately 5000 basic computational steps (additions, method calls, etc.), based on advice for the ForkJoin library in Java 7 and give or take a factor of 10. In other words, just avoid tasks that do millions of operations as well as tasks 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.

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 the person 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,

---

[7]OpenMP does not specify conforming schedulers in enough detail to be sure it will get this bound, but this is the state-of-the-art using scheduling algorithms developed for the Cilk framework. We believe that to be guaranteed the good time (and space) bounds offered by a Cilk-style implementation of the scheduler, an OpenMP divide-and-conquer algorithm must *either* use the `untied` clause *or* fork both the left and right branches using separate `task` directives.

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 only works 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.

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, 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.

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 couple examples that represent a couple of the many key parallel-programming patterns.

As is common when studying algorithms, we will not show full C++ implementations. It should be clear at this point that one could code up the algorithms using the OpenMP framework, though, as is common with sophisticated data structures and algorithms, it may not be entirely easy.

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*.[8] A $\Theta(n)$ sequential solution is trivial:

```
void prefixSum(int output[], int input[], int len) {
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1] + input[i];
}
```

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 1 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 $[0, n)$.[9]

- 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 $[n/2, n)$.

---

[8]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.

[9]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[10] 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. 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 bottom, *passing "down" as an argument the sum of the array indices to the left of the node*. Figure 2 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 ith leaf. Convincing yourself this algorithm is correct will likely require working through a 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`.

- 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 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.

---

[10]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.
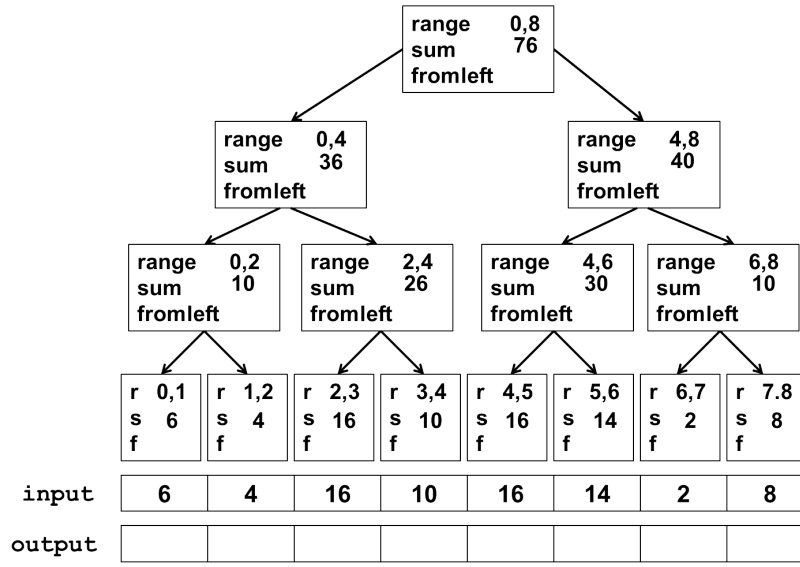
Figure 1: Example result of the first pass of the parallel prefix-sum algorithm: we have a binary tree where each node holds the sum of a range of elements of the input. We can build this tree bottom-up with $\Theta(n)$ work and $\Theta(\log n)$ span because a node's sum is just the sum of its children.
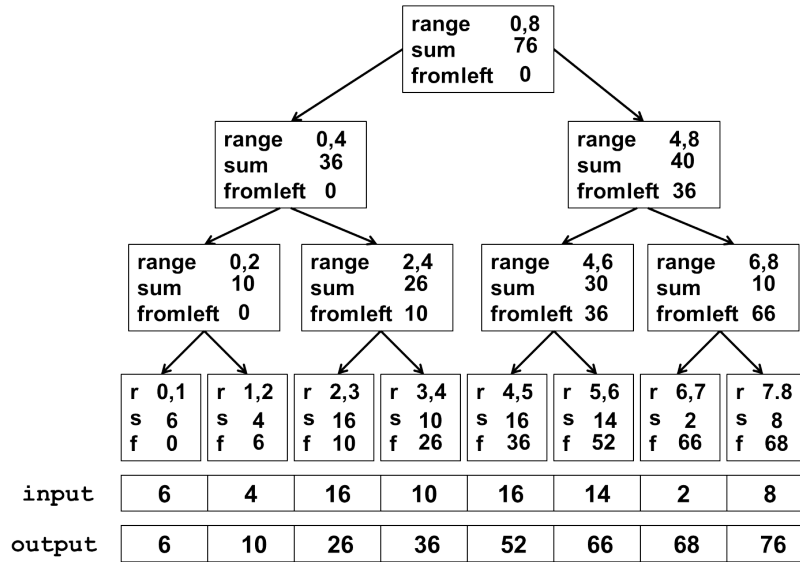
Figure 2: Example result (after the second pass) of the parallel prefix-sum algorithm: the second pass is a $\Theta(n)$ work, $\Theta(\log n)$ span top-down algorithm that passes a "fromleft" value from parents to children, starting with 0 at the root. At a leaf, this value holds the sum of values of all indices to the left, which is what we need for the output. A node passes "fromleft" unchanged to its left child and "fromleft" plus the sum of its left child to its right child.

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:

```
// caller is responsible for deallocating the output array
void greaterThanTen(int *& output, int & outputLen, int input[], int len) {
  outputLen = 0;
  for (int i=0; i < len; i++)
    if (input[i] > 10)
      outputLen++;
  output = new int[outputLen];
  int index = 0;
  for (int i=0; i < len; i++)
    if (input[i] > 10)
      output[index++] = input[i];
  assert(index==outputLen);
  return output;
}
```

Writing a generic version is really no more difficult; as in Section 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` and `bitsumLen`:

   ```
   int outputLen = bitsum[bitsumLen-1];
   output = new int[outputLen];
   FORALL(int i=0; i < inputLen; 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...

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

If the partition is roughly balanced, then the two recursive calls solve problems of approximately half the size. In the best case, the partition would be exactly balanced, leading to a recurrence for the running time of $T(n) = O(n) + 2T(n/2)$, which is $O(n \log n)$. It turns out if pivots are chosen 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 recurrence for the span to $T(n) = O(n) + 1T(n/2)$, which is $O(n)$. 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 a billion elements about 30 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.

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 recurrence is still $T(n) = O(n) + 2T(n/2) = O(n \log n)$ where the $O(n)$ now includes the two pack operations. The span recurrence is now $T(n) = O(\log n) + 1T(n/2)$ because the span for the pack operations is $O(\log n)$. The solution to this recurrence is $O(\log^2 n)$, 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.

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 recurrence for this algorithm is $T(n) = 2T(n/2) + O(n)$ 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.

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 $T(n) = 1T(n/2) + O(n)$, 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 (modulo rounding), 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 $O(1)$.

- 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 $3n/4$. Therefore, the worst-case span is $T(n) = T(3n/4) + O(\log n)$; the $T(n/4)$ does not appear because it can be done in parallel with the $T(3n/4)$ and is expected to finish first. It turns out the solution to this recurrence is $O(\log^2 n)$. The work is $T(n) = T(3n/4) + T(n/4) + O(\log n)$, which is $O(n)$.

Recall this analysis was just for the merging step. Adding this back into the overall mergesort algorithm, we get a span of $T(n) = 1T(n/2) + O(\log^2 n)$, which is $O(\log^3 N)$, and a work of $T(n) = 2T(n/2) + O(n)$, 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.

---

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

6 Basic Shared-Memory Concurrency

We now leave fork-join parallelism behind to focus on concurrent programming, which is about correctly and efficiently controlling access by multiple threads to shared resources. As described in Section 2, the line between parallel and concurrent programming is not clear, and many programs have aspects of both, but it is best at first to study each by itself.

We will still have threads and shared memory. We will use C++11's built-in threads (`std::thread`), not the OpenMP framework. The "shared resources" will be memory locations (fields of objects) used by more than one thread. We will learn how to write code that provides properly synchronized access to shared resources even though it may not be known what order the threads may access the data. In fact, multiple threads may *try* to access the data at the same time. We will see why this cannot be allowed and how programmers must use programming-language features to avoid it. The features we will focus on involve *mutual-exclusion locks*, introduced in Section 6.3.

Here are some simple high-level examples of shared resources where we need to control concurrent access:

1. We are writing banking software where we have an object for each back account. Different threads (e.g., one per bank teller or ATM) deposit or withdraw funds from various accounts. What if two threads try to manipulate the same account at the same time?

2. We have a hashtable storing a cache of previously-retrieved results. For example, maybe the table maps file names to file contents so that we can avoid reopening files we have already accessed. What if two threads try to insert files — maybe even the same file — at the same time?

3. Suppose we have a *pipeline*, which is like an assembly line, where each *stage* is a separate thread that does some processing and then gives the result to the subsequent stage. To pass these results, we could use a queue where the result-provider enqueues and the result-receiver dequeues. So an n-stage pipeline would have $n-1$ queues; one queue between each pair of adjacent stages. What if an enqueue and a dequeue happen at the same time? What if a dequeue is attempted when there are no elements in the queue (yet)?

6.1 The Programming Model

As the examples above hint at, in concurrent programming we have multiple threads that are "largely doing their own thing" but occasionally need to coordinate since they are accessing shared resources. It is like different cooks working in the same kitchen — easy if they are using different utensils and stove burners, but more difficult if they need to share things. The cooks may be working toward a shared goal like producing a meal, but while they are each working on a different recipe, the shared goal is not the focus.

In terms of programming, the basic model comprises multiple threads that are running in a mostly uncoordinated way. We might create a new thread when we have something new to do. The operations of each thread are *interleaved* (running alongside, before, after, or at the same time) with operations by other threads. This is very different from fork-join parallelism. In fork-join algorithms, one thread creates a helper thread to solve a specific subproblem and the creating thread waits (by calling join) for the other thread to finish. Here, we may have 4 threads processing bank-account changes as they arrive. While it is unlikely that two threads would access the same account at the same time, it is possible and we must be correct in this case.

One thing that makes concurrent programs very difficult to debug is that the bugs can be very unlikely to occur. If a program exhibits a bug and you re-run the program another million times with the same inputs, the bug may not appear again. This is because what happens can depend on the order that threads access shared resources, which is not entirely under programmer control. It can depend on how the threads are scheduled onto the processors. Therefore, concurrent programming is *non-deterministic*, the output does not depend only on the input. Because testing concurrent programs is so difficult, it is exceedingly important to design them well using well-understood idioms (see Section 8) from the beginning.

As an example, suppose a bank-account class has methods `deposit` and `withdraw`. Suppose the latter throws an exception if the amount to be withdrawn is larger than the current balance. If one thread deposits into the account and another thread withdraws from the account, then whether the withdrawing thread throws an exception could depend on the order the operations occur. And that is still assuming all of one operation completes before the other starts. The

next couple sections show how to ensure the operations themselves are not interleaved, but even after ensuring this, the program can still be non-deterministic.

As a contrast, we can see how fork-join parallelism made it relatively easy to avoid having two threads access the same memory at the same time. Consider a simple parallel reduction like summing an array. Each task (and thus each thread) accesses a disjoint portion of the array, so there is no sharing like there potentially is with bank accounts. The sharing that does occur is essentially "one-way" for any particular memory location. One thread "parcels out" inputs (like the array location and range to consider) to a newly created helper thread. The helper thread writes results that the other thread reads after the helper thread terminates. The *synchronization* here is accomplished entirely via (1) thread creation (not calling the `thread` constructor or invoking the `task pragma` until the correct fields are written) and (2) joins (waiting to read results until the other thread has terminated using `join` or the `taskwait pragma`). But in our concurrent programming model, this form of synchronization will not work because we will not wait for another thread to finish running before accessing a shared resource like a bank account.

It is worth asking why anyone would use this difficult programming model. It would certainly be simpler to have one thread that does everything we need to do. There are several reasons:

- *Parallelism:* Despite conceptual differences between parallelism and concurrency, it may well be that a parallel algorithm needs to have different threads accessing some of the same data structures in an unpredictable way. For example, we could have multiple threads search through a graph, only occasionally crossing paths.

- *Responsiveness:* Many programs, including operating systems and programs with user interfaces, want/need to respond to external events quickly. One way to do this is to have some threads doing the program's expensive computations while other threads are responsible for "listening for" events like buttons being clicked or typing occurring. The listening threads can then (quickly) write to some fields that the computation threads later read.

- *Processor utilization:* If one thread needs to read data from disk (e.g., a file), this will take a very long time relatively speaking. In a conventional single-threaded program, the program will not do anything for the milliseconds it takes to get the information. But this is enough time for another thread to perform millions of instructions. Hence by having other threads the program can do useful work while waiting for I/O. This use of multithreading is called *masking I/O latency*.

- *Failure/performance isolation:* Sometimes having multiple threads is simply a more convenient way to structure a program. In particular, when one thread throws an exception or takes too long to compute a result, it affects only what code is executed by that thread. If we have multiple independent pieces of work, some of which might (due to a bug, a problem with the data, or some other reason) cause an exception or run for too long, the other threads can still continue executing. There are other approaches to these problems, but threads often work well.

It is common to hear that threads are useful *only* for performance. This is true only if "performance" includes responsiveness and isolation, which stretches the definition of performance.

6.2   The Need for Synchronization

This section presents in detail an initial example to demonstrate why we should prevent multiple threads from simultaneously performing operations on the same memory. We will focus on showing *possible interleavings* that produce the wrong answer. However, the example also has *data races*, which Section 7 explains must be prevented. We will show that using "ordinary" C++ features to try to prevent bad interleavings simply does not work. This motivates learning about *locks*, which are primitives provided by C++ and other programming languages that provide us what we need. We will not learn how to *implement* locks, since the techniques use low-level features more central to courses in operating systems and computer architecture.

Consider this code for a `BankAccount` class:

```
class BankAccount {
  int balance;
public:
  struct insufficientFunds : virtual std::exception {};
```

```
  virtual int getBalance() { return balance; }
  virtual void setBalance(int newBalance) { balance = newBalance; }

  virtual void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw insufficientFunds();
    setBalance(b - amount);
  }
  // ... other operations like deposit, destructor, etc.
};
```

This code is correct in a single-threaded program. But suppose we have two threads, one calls `x->withdraw(100)` and the other calls `y->withdraw(100)`. "Most likely" these calls are not a problem:

- If `x` and `y` are not aliases, meaning they refer to distinct bank accounts, then there is no problem because the calls are using different memory. This is like two cooks using different pots at the same time.

- If one call finishes before the other starts, then the behavior is like in a sequential program. This is like one cook using a pot and then another cook using the same pot. When this is not the case, we say the calls *interleave*. Note that interleaving can happen even with one processor because a thread can be *pre-empted* at any point, meaning the thread scheduler stops the thread and runs another one.

So let us consider two interleaved calls to `withdraw` on the same bank account. We will "draw" interleavings using a vertical timeline (earlier operations closer to the top) with each thread in a separate column. There are many possible interleavings since even the operations in a helper method like `getBalance` can be interleaved with operations in other threads. But here is one incorrect interleaving. Assume initially the `balance` field holds 150 and both withdrawals are passed 100.

```
Thread 1                        Thread 2
--------                        --------
int b = getBalance();
                                int b = getBalance();
                                if(amount > b)
                                  throw insufficientFunds();
                                setBalance(b - amount);
if(amount > b)
  throw insufficientFunds();
setBalance(b - amount);
```

If this interleaving occurs, the resulting `balance` will be 50 and no exception will be thrown. But two `withdraw` operations were supposed to occur — there *should* be an exception. Somehow we "lost" a withdraw, which would not make the bank happy. The problem is that `balance` changed after Thread 1 retrieved it and stored it in `b`.

When first learning concurrent programming there is a natural but almost-always WRONG attempted fix to this sort of problem. It is tempting to rearrange or repeat operations to try to avoid using "stale" information like the value in `b`. Here is a WRONG idea:

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw insufficientFunds();
  // maybe balance changed, so get the new balance
  setBalance(getBalance() - amount);
}
```

The idea above is to call `getBalance()` a second time to get any updated values. But there is *still* an incorrect interleaving. Just because `setBalance(getBalance() - amount)` is on one line in our source code does not mean it happens all at once. This code still (a) calls `getBalance`, then (b) subtracts `amount` from the result, then (c) calls `setBalance`. Moreover (a) and (c) may consist of multiple steps. In any case, the balance can change between (a) and (c), so we have not really accomplished anything except we might now produce a negative balance without raising an exception.

The sane way to fix this sort of problem, which arises whenever we have concurrent access to shared memory that might change (i.e., the contents are *mutable*), is to enforce *mutual exclusion*: allow only one thread to access any particular account at a time. The idea is that a thread will "hang a do-not-disturb sign" on the account before it starts an operation such as `withdraw` and not remove the sign until it is finished. Moreover, all threads will check for a "do-not-disturb sign" before trying to hang a sign and do an operation. If such a sign is there, a thread will *wait* until there is no sign so that it can be sure it is the only one performing an operation on the bank account. We call the work done "while the sign is hanging" a *critical section*; it is critical that such operations not be interleaved with other conflicting ones. (Critical sections often have other technical requirements, but this simple informal definition will suffice for our purposes.)

Here is one WRONG way to try to implement this idea. You should never write code that tries to do this manually, but it is worth understanding why it is WRONG:

```
class BankAccount {
  int balance;
  bool busy;
public:
  virtual void withdraw(int amount) {
    while (busy) { /* spin-wait */ }
    busy = true;
    int b = getBalance();
    if (amount > b)
      throw insufficientFunds();
    setBalance(b - amount);
    busy = false;
  }
  // deposit would spin on same boolean
};
```

The idea is to use the `busy` variable as our do-not-disturb sign. All the other account operations would use the `busy` variable in the same way so that only one operation occurs at a time. The while-loop is perhaps a little inefficient since it would be better not to do useless work and instead be notified when the account is "available" for another operation, but that is not the real problem. The `busy` variable does not work, as this interleaving shows:

```
Thread 1                        Thread 2
--------                        --------
while(busy) { }
                                while(busy) { }
busy = true;
                                busy = true;
int b = getBalance();
                                int b = getBalance();
                                if(amount > b)
                                  throw insufficientFunds();
                                setBalance(b - amount);
if(amount > b)
  throw insufficientFunds();
setBalance(b - amount);
```

36

Essentially we have the same problem with `busy` that we had with `balance`: we can check that `busy` is false, but then it might get set to true before we have a chance to set it to true ourselves. We need to check there is no do-not-disturb sign *and* put our own sign up while *we still know there is no sign*. Hopefully you can see that using a variable to see if `busy` is busy is only going to push the problem somewhere else again.

To get out of this conundrum, we rely on *synchronization primitives* provided by the programming language. These typically rely on special hardware instructions that can do a little bit more "all at once" than just read or write a variable. This turns out to be enough to implement mutual exclusion properly. The particular kind of synchronization we will use is *locking*.

6.3 Locks

We can define a *mutual-exclusion lock*, also known as just a *lock* (or sometimes called a *mutex*), as an abstract datatype designed for concurrent programming and supporting 3 operations:

- `new` creates a new lock that is initially "not held"

- `acquire` takes a lock and blocks until it is currently "not held" (this could be right away, i.e., not blocking at all, since the lock might already be not held). It sets the lock to "held" and returns.

- `release` takes a lock and sets it to "not held."

This is exactly the "do-not-disturb" sign we were looking for, where the `acquire` operation blocks (does not return) until the caller is the thread that most recently hung the sign. It is up to the lock implementation to ensure that it always does the right thing no matter how many acquires and/or releases different threads perform simultaneously. For example, if there are three acquire operations at the same time for a "not held" lock, one will "win" and return immediately while the other two will block. When the "winner" calls `release`, one other thread will get to hold the lock next, and so on.

The description above is a general notion of what locks are. Section 6.4 will describe how to use the locks that are part of the C++ language. But using the general idea, we can write this PSEUDOCODE (C++ does not actually have a `Lock` class), which is almost correct (i.e., still WRONG but close):

```
class BankAccount {
  int balance;
  Lock lk;
public:
  virtual void withdraw(int amount) {
    lk.aqcuire(); /* may block */
    int b = getBalance();
    if (amount > b)
      throw insufficientFunds();
    setBalance(b - amount);
    lk.release();
  }
  // deposit would also acquire/release the same Lock object lk
};
```

Though we show only the `withdraw` operation, remember that all operations accessing `balance` and whatever other fields the account has should acquire and release the lock `lk`. That way only one thread will perform any operation on a particular account at a time. Interleaving a `withdraw` and a `deposit`, for example, is incorrect. If any operation does not use the lock correctly, it is like threads calling that operation are ignoring the do-not-disturb sign. Because each account uses a different lock, different threads can still operate on different accounts at the same time. But because locks work correctly to enforce that only one thread holds any given lock at a time, we have the mutual exclusion we need for each account.

Notice that using a single lock for all accounts would also be correct; we would still have mutual exclusion. However, we might have much worse performance since only one thread would be able to manipulate any account at a time.

What, then, is wrong with the pseudocode? The biggest problem occurs if `amount > b`. In this case, an exception is thrown and therefore the lock is not released. So no thread will ever be able to perform another operation on this bank account and any thread that attempts to do so will be blocked *forever* waiting for the lock to be released. This situation is like someone leaving without removing the do-not-disturb sign, so nobody else ever enters the room.

A fix in this case would be to add `lk.release` in the branch of the if-statement before the throw-statement. But more generally we have to worry about any exception that might be thrown, even while some other method called from within the critical section is executing. So more generally it would be safer to use a catch-statement to make sure the lock is always released. Fortunately, as Section 6.4 describes, when we use C++'s locks instead of our pseudocode locks, this will not be necessary.

A second problem, which we can also avoid in C++, is more subtle. Notice that `withdraw` calls `getBalance` and `setBalance` while holding the lock for the account. These methods might also be called directly from outside the bank-account implementation, assuming they are public. Therefore, they should also acquire and release the lock before accessing the account's fields like `balance`. But if `withdraw` calls these helper methods *while holding the lock*, then as we have described it, the thread would block forever — waiting for "some thread" to release the lock when in fact the thread itself already holds the lock.

There are two solutions to this problem. The first solution is more difficult for programmers: we can define two versions of `getBalance` and `setBalance`. One would acquire the lock and one would assume the caller already holds it. The latter could be private, perhaps, though there are any number of ways to manually keep track in the programmer's head (and hopefully documentation!) what locks the program holds where.

The second solution is to change our definition of the `acquire` and `release` operations so that it is okay for a thread to (re)acquire a lock it already holds. Locks that support this are called *re-entrant locks*. (The name fits well with our do-not-disturb analogy.) To define re-entrant locks, we need to adjust the definitions for the three operations by adding a *count* to track how many times the current holder has reacquired the lock (as well as the identity of the current holder):

- `new` creates a new lock with no current holder and a count of 0

- `acquire` blocks if there is a current holder *different from the thread calling it*. Else if the current holder is the thread calling it, do not block and increment the counter. Else there is no current holder, so set the current holder to the calling thread.

- `release` only releases the lock (sets the current holder to "none") if the count is 0. Otherwise, it decrements the count.

In other words, a lock is released when the number of `release` operations by the holding thread equals the number of `acquire` operations.

6.4 Locks in C++11

The C++ language has built-in support for locks that is different than the pseudocode in the previous section, but easy to understand in terms of what we have already discussed. The main change is to separate out the object that we lock on—called a `recursive_mutex`—from the object we use to manage the lock—called a `lock_guard`.[12] (Note that the "recursive" part in `recursive_mutex` means the lock is re-entrant as described above.)

Although a `recursive_mutex` has a `lock` and `unlock` function corresponding to `acquire` and `release`, we won't use them. That's because the `lock_guard` class solves for us the problem of releasing the lock if an exception is thrown. When we create the `lock_guard`, we give it the mutex object we want to lock, and it locks (acquires) the mutex for us. When the `lock_guard` is destroyed, it unlocks (releases) the mutex for us.

Why is this handy? Consider this template for using `lock_guard`:

---

[12]In fact several other types of `mutex`es and locking objects exist, but we'll focus on these two.

```
class SomeClass {
  ...
public:
  std::recursive_mutex m;
  ...
  void doSomething() {
    std::lock_guard<std::recursive_mutex> guard(m);
    ...
  }
}
```

On each call to doSomething, the guard object will lock m. If some other thread is already holding the lock, that will block until the lock is available. (It won't block if this thread is holding the lock since we used the recursive version of a mutex.)

Then, the rest of the function executes while this thread holds the lock on m.

Finally, we want to release the lock when the function finishes. Fortunately, guard is a local variable inside doSomething. C++ guarantees it will be destroyed when the function finishes execution, whether it finishes normally or because of an exception. When that happens, guard's destructor will unlock m for us automatically. So, by the time the function returns, this thread has unlocked m, just as it should. (You can also use enclosing { ... } to create a tighter locking scope.)

Here is an implementation of our bank account class using C++'s locks. Notice that we save ourselves some typing and confusion by using typedef to give short names to the particular type of mutex and lock we want to use.

```
class BankAccount {
  int balance;
public:
  struct insufficientFunds : virtual std::exception {};
  typedef std::recursive_mutex mutex;  // short name for the mutex
  typedef std::lock_guard<mutex> lock; // short name for the lock
  mutex m;

  int getBalance() {
    lock guard(m);
    return balance;
  }
  void setBalance(int newBalance) {
    lock guard(m);
    balance = newBalance;
  }
  void withdraw(int amount) {
    lock guard(m);
    int b = getBalance();
    if (amount > b)
      throw insufficientFunds();
    setBalance(b - amount);
    assert(getBalance() == b - amount);
  }
  // deposit and other operations would also lock on m.
};
```

Because we use a re-entrant mutex, it is no problem for withdraw to call setBalance and getBalance. Because our lock guard automatically releases the lock, the exception in withdraw is not a problem.

39

While it may seem naturally good style to make `m` a `private` field since it is an implementation detail (clients need not care *how* instances of `BankAccount` provide mutual exclusion), this choice prevents clients from writing their own critical sections involving bank accounts. For example, suppose a client wants to double an account's balance. In a sequential program, another class could include this correct method:

```
void doubleBalance(BankAccount & acct) {
  acct.setBalance(acct.getBalance()*2);
}
```

But this code is subject to a bad interleaving: the balance might be changed by another thread after the call to `getBalance`. If `m` were instead `public`, then clients could use it—and the convenience of re-entrant locks—to write a proper critical section:

```
void doubleBalance(BankAccount & acct) {
  BankAccount::lock guard(acct.m);
  acct.setBalance(acct.getBalance()*2);
}
```

In this way, `doubleBalance` acquires the same lock that the methods of `acct` do, ensuring mutual exclusion. That is, `doubleBalance` will not be interleaved with other operations on the same account.

As long as critical sections only access the state of a single shared object, simply guarding each method in the shared object's class may work well. For some other cases, see the guidelines in Section 8.

Finally, note that recursive mutexes and lock guards (and joins) are far from the only support for concurrent programming in C++. In Section 10.2 we will show C++'s support for condition variables. The standard libraries `thread` and `future` also have many useful classes and functions for different tasks.

Unfortunately, while C++ specifies that its container classes will work when different elements are modified in parallel[13], it gives no guarantees about simultaneous modification of the container as a whole or to a single element. Nonetheless, you probably should *not* implement your own concurrent versions of container data structures like hash tables and queues. There are many carefully optimized correct implementations available, such as in Intel's (open source) Threading Building Blocks library.

7   Race Conditions: Bad Interleavings and Data Races

A *race condition* is a mistake in your program (i.e., a bug) such that whether the program behaves correctly or not depends on the order that the threads execute. The name is a bit difficult to motivate: the idea is that the program will "happen to work" if certain threads during program execution "win the race" to do their next operation before some other threads do their next operations. Race conditions are very common bugs in concurrent programming that, by definition, do not exist in sequential programming. This section defines two kinds of race conditions and explains why you must avoid them. Section 8 describes programming guidelines that help you do so.

One kind of race condition is a *bad interleaving*, sometimes called a *higher-level race* (to distinguish it from the second kind). Section 6 showed some bad interleavings and this section will show several more. The key point is that "what is a bad interleaving" *depends entirely on what you are trying to do*. Whether or not it is okay to interleave two bank-account withdraw operations depends on some specification of how a bank account is supposed to behave.

A *data race* is a specific kind of race condition that is better described as a "simultaneous access error" although nobody uses that term. There are two kinds of data races:

- When one thread might *read* a memory location at the same moment that another thread *writes* the same memory location.

- When one thread might *write* a memory location at the same moment that another thread also *writes* the same memory location.

---

[13]Specifically, there will be no data race when different contained elements are concurrently modified. The exception to this is `vector`s of Boolean values, presumably so that they can be stored as bitvectors.

Notice it is *not an error* for two threads to both *read* the same memory location at the same time.

As Section 7.2 explains, *your programs must never have data races even if it looks like a data race would not cause an error — if your program has data races, the execution of your program is allowed to do very strange things.*

To better understand the difference between bad interleavings and data races, consider the final version of the BankAccount class from Section 6. If we accidentally omit the locking statement from the withdraw definition, the program will have many bad interleavings, such as the first one we considered in Section 6. But since getBalance and setBalance are still guarded by a locking statement, there are no data races: Every piece of code that reads or writes balance does so while holding the appropriate lock. So there can never be a simultaneous read/write or write/write of this field.

Now suppose withdraw is guarded by its locking statement but we accidentally omitted the locking statement from getBalance. Data races now exist and the program is wrong regardless of what a bank account is supposed to do. Interestingly, simultaneous calls to getBalance do *not* cause a data race because simultaneous reads are not an error. But if one thread calls getBalance at the same time another thread calls setBalance, then a data race occurs. Even though setBalance is guarded, getBalance is not, so they might read and write balance at the same time. If your program has data races, it is infeasible to reason about what might happen. (In other words, literally *anything* is allowed to happen!)

7.1 Bad Interleavings: An Example with Stacks

A useful way to reason about bad interleavings is to think about *intermediate states* that other threads must not "see." Data structures and libraries typically have *invariants* necessary for their correct operation. For example, a size field may need to store the correct number of elements in a data structure. Operations typically violate invariants temporarily, restoring the invariant after making an appropriate change. For example, an operation might add an element to the structure *and* increment the size field. No matter what order is used for these two steps, there is an intermediate state where the invariant does not hold. In concurrent programming, our synchronization strategy often amounts to using mutual exclusion to ensure that no thread can observe an invariant-violating intermediate state produced by another thread. What invariants matter for program correctness depends on the program.

Let's consider an extended example using a basic implementation of a bounded-size stack. (This is not an interesting data structure; the point is to pick a small example so we can focus on the interleavings. Section 10.2 discusses an alternative to throwing exceptions for this kind of structure.)

```
template <typename E>
class Stack {
  E * array;
  int size, index;
public:
  struct stackException : virtual std::exception {};
  struct stackEmptyException : virtual stackException {};
  struct stackFullException : virtual stackException {};
  typedef std::recursive_mutex mutex;
  typedef std::lock_guard<mutex> guard;
  mutex m;

  virtual ~Stack() { delete [] array; }
  Stack(int size) : array(new E[size]), size(size) { }
  bool isEmpty() {
    guard lock(m);
    return index == 0;
  }
  void push(E val) {
    guard lock(m);
    if (index == size)
```

```
        throw stackFullException();
      array[index++] = val;
    }
    E pop() {
      guard lock(m);
      if (isEmpty())
        throw stackEmptyException();
      return array[--index];
    }
};
```

The key invariant is that if index>0 then index-1 is the position of the most recently pushed item that has not yet been popped. Both push and pop temporarily violate this invariant because they need to modify index and the contents of the array. Even though this is done in one C++ statement, it does not happen all at once. By guarding each method on a single mutex, no thread using these operations can see an incorrect intermediate state. It effectively ensures there is some global order of calls to isEmpty, push, and pop. That order might differ across different executions because scheduling is non-deterministic, but the calls will not be interleaved.

Also notice there are no data races because array and size are accessed only while holding the lock. Not holding the lock in the constructor is fine because a new object is not yet reachable from any other thread. The new stack that is returned will have to be assigned to some thread-shared location before a second thread could use it, and such an assignment will not happen until after the constructor finishes executing.[14]

Now suppose we want to implement a new operation for our stack called peek (also known as top) that returns the newest not-yet-popped element in the stack without popping it. A correct implementation would be:

```
    E peek() {
      guard lock(m);
      if(index==0)
        throw stackEmptyException();
      return array[index-1];
    }
```

If we omit the locking statement, then this operation would cause data races with simultaneous push or pop operations.

Consider instead this alternate also-correct implementation:

```
    E peek() {
      guard lock(m);
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
```

This version is perhaps worse style, but it is certainly correct. Additionally, a helper method *outside* the class could take this approach (but not the first approach since array and index are private fields):

```
template <typename E>
E myPeekHelper(Stack<E> & s) {
  Stack<E>::lock guard(s.m);
  E ans = s.pop();
  s.push(ans);
  return ans;
}
```

---

[14]It would be wrong for the constructor to "expose" itself by doing something like someObject.f = this;, but it is almost always a horrible and strange idea to do something like that anyway.

Notice this version could not be written if stacks used some private inaccessible lock. Also notice that it relies on re-entrant locks. However, we will consider instead this WRONG version where the helper method omits its own locking statement:

```
template <typename E>
E myPeekHelperWrong(Stack<E> & s) {
  E ans = s.pop();
  s.push(ans);
  return ans;
}
```

Notice this version has no data races because the `pop` and `push` calls still acquire and release the appropriate lock. Also notice that `myPeekHelperWrong` uses the stack operators exactly as it is supposed to. Nonetheless, it is incorrect because a peek operation is not supposed to modify the stack. While the *overall result* is the same stack if the code runs without interleaving from other threads, the code produces an *intermediate state* that other threads should not see or modify. This intermediate state would not be observable in a single-threaded program or in our correct version that made the entire method body a critical section.

To show why the wrong version can lead to incorrect behavior, we can demonstrate interleavings with other operations such that the stack does the wrong thing. Writing out such interleavings is good practice for reasoning about concurrent code and helps determine what needs to be in a critical section. As it turns out in our small example, `myPeekHelperWrong` causes race conditions with all other stack operations, including itself.

`myPeekHelperWrong` **and** `isEmpty`:

If a stack is not empty, then `isEmpty` should return `false`. Suppose two threads share a stack `stk` that has one element in it. If one thread calls `isEmpty` and the other calls `myPeekHelperWrong`, the first thread can get the wrong result:

```
Thread 1                      Thread 2 (calls myPeekHelperWrong)
--------                      ----------------------------------
                              E ans = stk.pop();

bool b = isEmpty();

                              stk.push(ans);
                              return ans;
```

Notice there is nothing wrong with what "C++ is doing" here. It is the programmer who wrote `myPeekHelperWrong` such that it does not behave like a peek operation should. Yet the code that gets the wrong answer is `isEmpty`, which is another reason debugging concurrent programs is difficult.

`myPeekHelperWrong` **and** `push`:

Another key property of stacks is that they return elements in last-in-first-out order. But consider this interleaving, where one of Thread 1's `push` operations happens in the middle of a concurrent call to `myPeekHelperWrong`.

```
Thread 1                      Thread 2 (calls myPeekHelperWrong)
--------                      ----------------------------------
stk.push(x);
                              E ans = stk.pop();
stk.push(y);
                              stk.push(ans);
                              return ans;
E z = stk.pop();
```

This interleaving has the effect of reordering the top two elements of the stack, so `z` ends up holding the wrong value.

`myPeekHelperWrong` **and** `pop`:

Similar to the previous example, a `pop` operation (instead of a `push`) that views the intermediate state of `myPeekHelperWrong` can get the wrong (not last-in-first-out) answer:

```
Thread 1                        Thread 2 (calls myPeekHelperWrong)
--------                        --------------------------------
stk.push(x);
stk.push(y);
                                E ans = stk.pop();
E z = stk.pop();

                                stk.push(ans);
                                return ans;
```

myPeekHelperWrong **and** myPeekHelperWrong:

Finally, two threads both calling myPeekHelperWrong causes race conditions. First, if stk initially has one element, then a bad interleaving of the two calls could raise a StackEmptyException, which should happen only when a stack has zero elements:

```
Thread 1                        Thread 2 (calls myPeekHelperWrong)
--------                        --------------------------------
E ans = stk.pop();
                                E ans = stk.pop(); // exception!

stk.push(ans);
return ans;
```

Second, even if the stack has more than 1 element, we can extend the interleaving above to swap the order of the top two elements, which will cause later pop operations to get the wrong answer.

```
Thread 1                        Thread 2 (calls myPeekHelperWrong)
--------                        --------------------------------
E ans = stk.pop();
                                E ans = stk.pop();

stk.push(ans);
return ans;
                                stk.push(ans);
                                return ans;
```

In general, bad interleavings can involve any number of threads using any operations. The defense is to define large enough critical sections such that every possible thread schedule is correct. Because push, pop, and isEmpty are guarded with locks, we do not have to worry about calls to these methods being interleaved with each other. To implement peek correctly, the key insight is to realize that its intermediate state must not be visible and therefore we need its calls to pop and push to be part of *one* critical section instead of two separate critical sections. As Section 8 discusses in more detail, making your critical sections large enough — but not too large — is an essential task in concurrent programming.

7.2 Data Races: Wrong Even When They Look Right

This section provides more detail on the rule that data races — simultaneous read/write or write/write of the same field — must be avoided even if it "seems like" the program would be correct anyway.

For example, consider the Stack implementation from Section 7.1 and suppose we omit synchronization from the isEmpty method. Given everything we have discussed so far, doing so seems okay, if perhaps a bit risky. After all, isEmpty only reads the index field and since all the other critical sections write to index at most once, it cannot be that isEmpty observes an intermediate state of another critical section: it would see index either before or after it was updated — and both possibilities remain when isEmpty is properly synchronized.

Nonetheless, omitting synchronization introduces data races with concurrent push and pop operations. As soon as a C++ program has a data race, it is *impossible* reason about what will happen: any program with a data race is as

44

wrong as a program with an array-bounds violation. So data races must be avoided. The rest of this section gives some indication as to *why* and a little more detail on *how* to avoid data races.

7.2.1 Inability to Reason In the Presence of Data Races

Let's consider an example that is simpler than the stack implementation but is so simple that we won't motivate why anyone would write code like this:

```
struct S {
  int x;
  int y;
  S() { x = 0; y = 0; }
  void f() {
    x = 1; // line A
    y = 1; // line B
  }
  void g() {
    int a = y; // line C
    int b = x; // line D
    assert(b >= a);
  }
}
```

Notice `f` and `g` are not synchronized, leading to potential data races on fields `x` and `y`. Therefore, it turns out the assertion in g can fail. But there is no interleaving of operations that justifies the assertion failure! Here we prove that all interleavings are correct. Then Section 7.2.2 explains why the assertion can fail anyway.

- If a call to g begins after at least one call to `f` has ended, then `x` and `y` are both 1, so the assertion holds.

- Similarly, if a call to g completes before any call to `f` begins, then `x` and `y` are both 0, so the assertion holds.

- So we need only concern ourselves with a call to g that is interleaved with the first call to `f`. One way to proceed is to consider all possible interleavings of the lines marked A, B, C, and D and show the assertion holds under all of them. Because A must precede B and C must precede D, there are only 6 such interleavings: ABCD, ACBD, ACDB, CABD, CADB, CDAB. Manual inspection of all six possibilities completes the proof.

  A more illuminating approach is a proof by contradiction: Assume the assertion fails, meaning `!(b>=a)`. Then `a==1` and `b==0`. Since `a==1`, line B happened before line C. Since (1) A must happen before B, (2) C must happen before D, and (3) "happens before" is a transitive relation, A must happen before D. But then `b==1` and the assertion holds.

There is nothing wrong with the proof except its assumption that we can reason in terms of "all possible interleavings" or that everything happens in certain orders. We can reason this way *only* if the program has no data races.

7.2.2 Partial Explanation of Why Data Races Are Disallowed

To understand why one cannot always reason in terms of interleavings requires taking courses in compilers and/or computer architecture.

Compilers typically perform *optimizations* to execute code faster without changing its meaning. If we required compilers to never change the possible interleavings, then it would be too difficult for compilers to be effective. Therefore, language definitions allow compilers to do things like execute line B above before line A. Now, in this simple example, there is no reason why the compiler *would* do this reordering. The point is that it is *allowed to*, and in more complicated examples with code containing loops and other features there are reasons to do so. Once such a reordering is allowed, the assertion is allowed to fail. Whether it will ever do so depends on exactly how a particular C++ implementation works.

Similarly, in hardware, there isn't *really* one single shared memory containing a single copy of all data in the program. Instead there are various caches and buffers that let the processor access some memory faster than other memory. As a result, the hardware has to keep track of different copies of things and move things around. As it does so, memory operations might not become "visible" to other threads in the order they happened in the program. As with compilers, requiring the hardware to expose all reads and writes in the exact order they happen is considered too onerous from a performance perspective.

To be clear, compilers and hardware cannot just "do whatever they want." All this reordering is *completely hidden from you* and you never need to worry about it *if* you avoid data races. This issue is irrelevant in single-threaded programming because with one thread you can never have a data race.

7.2.3 The Grand Compromise

One way to summarize the previous section is in terms of a "grand compromise" between programmers who want to be able to reason easily about what a program might do and compiler/hardware implementers who want to implement things efficiently and easily:

- The programmer promises not to write data races.

- The implementers promise that if the programmer does his/her job, then the implementation will be indistinguishable from one that does no reordering of memory operations. That is, it will preserve the illusion that all memory operations are interleaved in a global order.

Why is this compromise the state of the art rather than a more extreme position of requiring the implementation to preserve the "interleaving illusion" even in the presence of data races? In short, it would make it much more difficult to develop compilers and processors. And for what? Just to support programs with data races, which are almost always bad style and difficult to reason about anyway! Why go to so much trouble just for ill-advised programming style? On the other hand, if your program accidentally has data races, it might be more difficult to debug.

7.2.4 Avoiding Data Races

To avoid data races, we need synchronization. Given C++'s locks which we already know about, we could rewrite our example from Section 7.2.1 as follows:

```
struct S {
  std::recursive_mutex m;
  typedef std::lock_guard<std::recursive_mutex> guard;
  int x;
  int y;
  S() { x = 0; y = 0; }
  void f() {
    { guard lock(m); x = 1; } // line A
    { guard lock(m); y = 1; } // line B
  }
  void g() {
    int a,b;
    { guard lock(m); a = y; } // line C
    { guard lock(m); b = x; } // line D
    assert(b >= a);
  }
}
```

In practice you might choose to implement each method with one critical section instead of two, but using two is still sufficient to eliminate data races, which allows us to reason in terms of the six possible interleavings, which ensures the assertion cannot fail. It would even be correct to use two different locks, one to protect x (lines A and D) and another

to protect y (lines B and C). The bottom line is that by avoiding data races we can reason in terms of interleavings (ignoring reorderings) because we have fulfilled our obligations under The Grand Compromise.

There is a second way to avoid data races when writing tricky code that depends on the exact ordering of reads and writes to fields.

Instead of using locks, we can use `atomic` types. By *definition*, accesses to atomic objects *do not count as data races*, so programmers using atomics are still upholding their end of the grand compromise. For a given implementation, reading/writing an atomic object is probably less efficient than reading/writing a regular value, but more efficient than acquiring and releasing a lock. For our example, the code looks like this:

```
struct S {
  std::atomic<int> x;
  std::atomic<int> y;
  S() { x.store(0); x.store(0); }
  void f() {
    x.store(1); // line A
    y.store(1); // line B
  }
  void g() {
    int a = y.load(); // line C
    int b = x.load(); // line D
    assert(b >= a);
  }
};
```

Atomic objects are typically used only by concurrency experts. They can be convenient when concurrent code is sharing only a single field. Usually with more than one field you need critical sections longer than a single memory access, in which case the right tool is a lock not an atomic object.

7.2.5 A More Likely Example

Since the previous discussion focused on a strange and unmotivated example (who would write code like f and g?), this section concludes with a data-race example that would arise naturally.

Suppose we want to perform some expensive iterative computation, for example, rendering a really ugly monster for a video game, until we have achieved a perfect solution or another thread informs us that the result is "good enough" (it's time to draw the monster already) or no longer needed (the video-game player has left the room with the monster). We could have a boolean `stop` field that the expensive computation checks periodically:

```
class MonsterDraw {
  bool stop = false;
  void draw(...) {
    while(!stop) {
      ... keep making uglier ...
    }
    ...
  }
}
```

Then one thread would execute `draw` and other threads could set `stop` to true as necessary. Notice that doing so introduces data races on the `stop` field. To be honest, even though this programming approach is wrong, with most C++ implementations it will probably work. But if it does not, it is the programmer's fault. Worse, the code might work for a long time, but when the C++ implementation changes or the compiler is run with different pieces of code, the idiom might stop working. Most likely what would happen is the `draw` method would never "see" that `stop` was changed and it would not terminate.

The simplest fix to the code above is to declare `stop` as an atomic type. In a language like Java, you might instead use the capacity to interrupt other threads. C++11 does *not* support interrupting threads in this way; so, if you do want to be able to interrupt another thread, a design like this is not unreasonable.

8 Concurrency Programming Guidelines

This section does not introduce any new primitives for concurrent programming. Instead, we acknowledge that writing correct concurrent programs is difficult — just saying, "here is how locks work, try to avoid race conditions but still write efficient programs" would leave you ill-equipped to write non-trivial multithreaded programs. Over the decades that programmers have struggled with shared memory and locks, certain guidelines and methodologies have proven useful. Here we sketch some of the most widely agreed upon approaches to getting your synchronization correct. Following them does not make concurrency easy, but it does make it much easier. We focus on how to design/organize your code, but related topics like how to use tools to test and debug concurrent programs are also important.

8.1 Conceptually Splitting Memory in Three Parts

Every memory location in your program (for example an object field), should meet *at least one* of the following three criteria:

1. It is *thread-local*: Only one thread ever accesses it.

2. It is *immutable*: (After being initialized,) it is only read, never written.

3. It is *synchronized*: Locks—or other synchronization primitives—are used to ensure there are no race conditions.

Put another way, memory that is thread-local or immutable is irrelevant to synchronization. You can ignore it when considering how to use locks and where to put critical sections. So the more memory you have that is thread-local or immutable, the easier concurrent programming will be. It is often worth changing the data structures and algorithms you are using to choose ones that have less sharing of mutable memory, leaving synchronization only for the unavoidable communication among threads.

In terms of a Venn diagram, we can view memory as containing these sets:
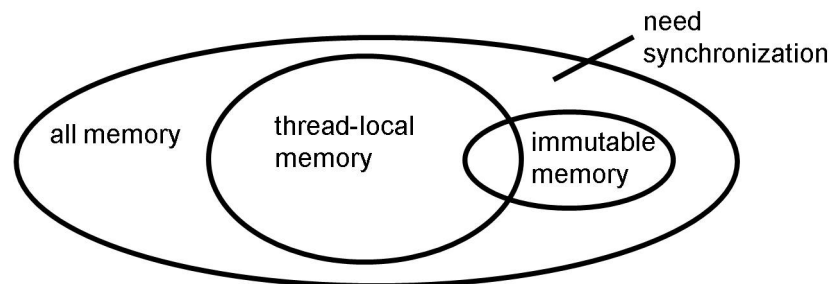
In these terms, the suggestion is to make the thread-local memory and immutable memory sets as large as possible. We consider approaches for the two sets separately:

More Thread-Local Memory

Whenever possible, do not share resources. Instead of having one resource, have a separate (copy of the) resource for each thread. For example, there is no need to have one shared `Random` object for generating pseudorandom numbers, since it will be just as (pseudo)random for each thread to use its own object. Dictionaries storing already-computed results can also be thread-local: This has the disadvantage that one thread will not insert a value that another thread can then use, but the decrease in synchronization is often worth this trade-off.

Note that the `std::thread` constructor forces you to pass pointers or explicitly annotate reference parameters to create sharing of local variables. Consider this a reminder to minimize sharing of data between threads! (C++11

also includes an explicit `thread_local` specifier that can be used to turn a global or static variable into multiple thread-local copies. As with all memory locations, however, the programmer can also explicitly make these accessible to other threads.)

Overall, the vast majority of objects in a typical concurrent program should and will be thread-local. It may not seem that way in these notes, but only because we are focusing on the shared objects, which are the difficult ones. In conclusion, do not share objects unless those objects really have the *explicit purpose* of enabling shared-memory communication among threads.

More Immutable Memory

Whenever possible, do not update fields of objects: Make new objects instead. This guideline is a key tenet of *functional programming*, which unfortunately is typically not taught in the same course as data structures or concurrency. Even in single-threaded programs, it is often helpful to avoid mutating fields of objects because other too-often-forgotten-about *aliases* to that object will "see" any mutations performed.

Let's consider a small example that demonstrates some of the issues. Suppose we have a dictionary such as a hashtable that maps student ids to names where the name class is represented like this:

```
struct Name {
  std::string first;
  std::string middle;
  std::string last;
  Name(std::string f, std::string m, std::string l) {
    first  = f;
    middle = m;
    last   = l;
  }
};
std::ostream & operator<<(std::ostream & out, const Name & n) {
  out << n.first << " " << n.middle << " " << n.last;
}
```

Suppose we want to write a method that looks up a student by id in some table and returns the name, but using a middle initial instead of the full middle name. The following direct solution is probably the BEST STYLE:

```
    Name & n = table.lookup(id);
    std::stringstream sstream;
    sstream << n.first << " " << n.middle[0] << " " << n.last;
    return sstream.str();
```

Notice that the computation is almost the same as that already done by the stream insertion operator. So if the computation were more complicated than just a few string concatenations, it would be tempting and arguably reasonable to try to reuse the stream insertion operator. This approach is a BAD IDEA:

```
    Name & n = table.lookup(id);
    n.middle = n.middle.substr(0,1);
    std::stringstream sstream;
    sstream << n;
    return sstream.str();
```

While this code produces the right answer, it has the *side-effect* of changing the name's `middle` field. Worse, the Name object is *still in the table*. Actually, this depends on how the `lookup` method is implemented. We assume here it returns a reference to the object in the table, rather than creating a new Name object. Under this assumption, the full middle name for some student has been *replaced* in the table by the middle initial, which is presumably a bug.

In a SINGLE-THREADED PROGRAM, the following workaround is correct but POOR STYLE:

```
Name & n = table.lookup(id);
std::string m = n.middle;
n.middle = m.substr(0,1);
std::stringstream sstream;
sstream << n;
n.middle = m;
return sstream.str();
```

Again, this is total overkill for our example, but the idea makes sense: undo the side effect before returning.

But if the table is shared among threads, the APPROACH ABOVE IS WRONG. Simultaneous calls using the same id would have data races. Even if there were no data races, there would be interleavings where other operations might see the intermediate state where the `middle` field held only the initial. The point is that for shared memory, you cannot perform side effects and undo them later: later is too late in the presence of concurrent operations. This is a major reason that the functional-programming paradigm is even more useful with concurrency.

The solution in this case is simple: make a copy of the object by not declaring n to be a reference and, if necessary, mutate the (thread-local) copy. So this WORKS:

```
Name n = table.lookup(id);
n.middle = m.substr(0,1);
std::stringstream sstream;
sstream << n;
return sstream.str();
```

Note that it is no problem for multiple threads to lookup up the same Name object at the same time. They will all *read* the same fields of the same object (they all have aliases to it), but simultaneous reads never lead to race conditions. It is like multiple pedestrians reading the same street sign at the same time: They do not bother each other.

8.2 Approaches to Synchronization

Even after maximizing the thread-local and immutable objects, there will remain some shared resources among the threads (else threads have no way to communicate). The remaining guidelines suggest ways to think about correctly synchronizing access to these resources.

Guideline #0: Avoid data races

Use locks to ensure that two threads never simultaneously read/write or write/write the same field. While guideline #0 is *necessary* (see Section 7.2), it is not *sufficient* (see Section 7.1 and other examples of wrong programs that have no data races).

Guideline #1: Use consistent locking

For each location that needs synchronization, identify a lock that is always held when accessing that location. If some lock l is always held when memory location m is read or written, then we say that l *guards* the location m. It is extremely good practice to document (e.g., via comments) for each field needing synchronization what lock guards that location.

Note that each location would have exactly one guard, but one lock can guard multiple locations. A simple example would be multiple fields of the same object, but any mapping from locks to locations is okay. For example, perhaps all nodes of a linked list are protected by one lock. Another way to think about consistent locking is to imagine that all the objects needing synchronization are *partitioned*. Each piece of the partition has a lock that guards all the locations in the piece.

Consistent locking is *sufficient* to prevent data races, but is *not sufficient* to prevent bad interleavings. For example, the extended example in Section 7.1 used consistent locking.

Consistent locking is *not necessary* to prevent data races nor bad interleavings. It is a guideline that is typically the default assumption for concurrent code, violated only when carefully documented and for good reason. One good

reason is when the program has multiple conceptual "phases" and all threads globally coordinate when the program moves from one phase to another. In this case, different phases can use different synchronization strategies.

For example, suppose in an early phase multiple threads are inserting different key-value pairs into a dictionary. Perhaps there is one lock for the dictionary and all threads acquire this lock before performing any dictionary operations. Suppose that at some point in the program, the dictionary becomes fixed, meaning no more inserts or deletions will be performed on it. Once all threads *know* this point has been reached (probably by reading some other synchronized shared-memory object to make sure that all threads are done adding to the dictionary), it would be correct to perform subsequent lookup operations *without synchronization* since the dictionary has *become immutable*.

Guideline #2: Start with coarse-grained locking and move to finer-grained locking only if contention is hurting performance

This guideline introduces some new terms. *Coarse-grained locking* means using fewer locks to guard more objects. For example, one lock for an entire dictionary or for an entire array of bank accounts would be coarse-grained. Conversely, *fine-grained locking* means using more locks, each of which guards fewer memory locations. For example, using a separate lock for each bucket in a chaining hashtable or a separate lock for each bank account would be fine-grained. Honestly, the terms "coarse-grained" and "fine-grained" do not have a strict dividing line: we can really only say that one locking strategy is "more coarse-grained" or "more fine-grained" than another one. That is, *locking granularity* is really a continuum, where one direction is coarser and the other direction is finer.

Coarse-grained locking is typically easier. After grabbing just one lock, we can access many different locations in a critical section. With fine-grained locking, operations may end up needing to acquire multiple locks (using nested synchronized statements). It is easy to forget to acquire a lock or to acquire the wrong lock. It also introduces the possibility of deadlock (Section 9).

But coarse-grained locking leads to unnecessary blocking. In the extreme, the coarsest strategy would have just one lock for the entire program! Under this approach, it is obvious what lock to acquire, but no two operations on shared memory can proceed in parallel. Having one lock for many, many bank accounts is probably a bad idea because with enough threads, there will be multiple threads attempting to access bank accounts at the same time and they will be unable to. *Contention* describes the situation where threads are blocked waiting for each other: They are *contending* (in the sense of competing) for the same resource, and this is hurting performance. If there is little contention (it is rare that a thread is blocked), then coarse-grained locking is sufficient. When contention becomes problematic, it is worth considering finer-grained locking, realizing that changing the code without introducing race conditions is difficult. A good execution profiler should provide information on where threads are most often blocked waiting for a lock to be released.

Guideline #3: Make critical sections large enough for correctness but no larger. Do not perform I/O or expensive computations within critical sections.

The granularity of critical sections is completely orthogonal to the granularity of locks (guideline #2). Coarse-grained critical sections are "large" pieces of code whereas fine-grained critical sections are "small" pieces of code. However, what matters is *not* how many lines of code a critical section occupies. What matters is how long a critical section takes to execute (longer means larger) and how many shared resources such as memory locations it accesses (more means larger).

If you make your critical sections too short, you are introducing more possible interleavings and potentially incorrect race conditions. For example, compare:

```
{ lock guard(lk); /* do first thing */  }
/* do second thing */
{ lock guard(lk); /* do third thing */  }
```

with:

```
{
  lock guard(lk);
  /* do first thing */
  /* do second thing */
```

```
  /* do third thing */
}
```

The first version has smaller critical sections. Smaller critical sections lead to less contention. For example, other threads can use data guarded by `lk` while the code above is doing the "second thing." But smaller critical sections expose more states to other threads. Perhaps only the second version with one larger critical section is correct. This guideline is therefore a tough one to follow because it requires moderation: make critical sections no longer or shorter than necessary.

Basic operations like assignment statements and method calls are so fast that it is almost never worth the trouble of splitting them into multiple critical sections. However, expensive computations inside critical sections should be avoided wherever possible for the obvious reason that other threads may end up blocked waiting for the lock. Remember in particular that reading data from disk or the network is typically orders of magnitude slower than reading memory. Long running loops and other computations should also be avoided.

Sometimes it is possible to reorganize code to avoid long critical sections by repeating an operation in the (hopefully) rare case that it is necessary. For example, consider this large critical section that replaces a table entry by performing some computation on its old value:

```
{
  lock guard(lk);
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

We assume the program is incorrect if this critical section is any smaller: We need other threads to see either the old value in the table or the new value. And we need to compute v2 using the current v1. So this variation would be WRONG:

```
{
  lock guard(lk);
  v1 = table.lookup(k);
}
v2 = expensive(v1);
{
  lock guard(lk);
  table.remove(k);
  table.insert(k,v2);
}
```

If another thread updated the table to map k so some v3 while this thread was executing `expensive(v1)`, then we would not be performing the correct computation. In this case, either the final value in the table should be v3 (if that thread ran second) or the result of `expensive(v3)` (if that thread ran first), but never v2.

However, this more complicated version would WORK in this situation, under the CAVEAT described below:

```
bool loop_done = false;
while(!loop_done) {
  {
    lock guard(lk);
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  {
    lock guard(lk);
```

```
    if(table.lookup(k)==v1) {
      loop_done = true;
      table.remove(k);
      table.insert(k,v2);
    }
  }
}
```

The essence of the trick is the second critical section. If `table.lookup(k)==v1`, then we know what `expensive(v1)` would compute, having already computed it! So the critical section is small, but does the same thing as our original large critical section. But if `table.lookup(k)!=v1`, then our precomputation did useless work and the while-loop starts over. If we expect concurrent updates to be rare, then this approach is a "good performance bet" while still correct in all cases.

As promised, there is an important caveat to mention. This approach does *not* ensure the table entry for key k was unmodified during the call to `expensive`. There could have been any number of concurrent updates, removals, insertions, etc. with this key. All we know is that when the second critical section executes, the key k has value v1. Maybe it was unchanged or maybe it was changed and then changed back.

In our example, we assumed it did not matter. There are many situations where it *does* matter and in such cases, checking the value is wrong because it does not ensure the entry has not been modified. This is known as an "A-B-A" problem, meaning the value started as A, changed to B, and changed back to A, causing some incorrect code to conclude, wrongly, that it was never changed.

Guideline #4: Think in terms of what operations need to be *atomic*. Determine a locking strategy after you know what the critical sections are.

An operation is *atomic*, as in indivisible, if it executes either entirely or not at all, with no other thread able to observe that it has partly executed. (In the databases literature, this would be called atomic and *isolated*, but it is common in concurrent programming to conflate these two ideas under the term *atomic*.) Ensuring necessary operations appear to be atomic is exactly why we have critical sections. Since this is the essential point, this guideline recommends thinking first in terms of what the critical sections are *without thinking about locking granularity*. Then, once the critical sections are clear, how to use locks to implement the critical sections can be the next step.

In this way, we ensure we actually develop the software we want—the correct critical sections—without being distracted prematurely by the implementation details of how to do the locking. In other words, think about atomicity (using guideline #3) first and the locking protocol (using guidelines #0, #1, and #2) second.

Unfortunately, one of the most difficult aspects of lock-based programming is when software needs change. When new critical sections are needed in "version 2.0" of a program, it may require changing the locking protocol, which in turn will require carefully modifying the code for many other already-working critical sections. Section 9 even includes an example where there is really no good solution. Nonetheless, following guideline #4 makes it easier to think about why and how a locking protocol may need changing.

Guideline #5: Do not implement your own concurrent data structures. Use carefully tuned ones written by experts and provided in standard libraries.

Widely used libraries for popular programming languages already contain many reusable data structures that you should use whenever they meet your needs. For example, there is little reason in C++ to implement your own hashtable since the `std::unordered_map` class has been carefully tuned and tested. For concurrent programming, the advice to "reuse existing libraries" is even more important, precisely because writing, debugging, and testing concurrent code is so difficult. Experts who devote their lives to concurrent programming can write tricky fine-grained locking code once and the rest of us can benefit. For example, Intel's Threading Building Blocks library includes the `concurrent_hash_map` class implementing a hashtable that can be safely used by multiple threads with very little contention. It uses some techniques more advanced than discussed in these notes, but clients to the library have such trickiness hidden from them. For basic things like queues and dictionaries, do not implement your own.

If standard libraries are so good, then why learn about concurrent programming at all? For the same reason that educated computer scientists need a basic understanding of sequential data structures in order to pick the right ones

and use them correctly (e.g., designing a good hash function), concurrent programmers need to understand how to debug and performance tune their use of libraries (e.g., to avoid contention). Moreover, standard libraries do not typically handle all the necessary synchronization for an application. For example, `concurrent_hash_map` does not support atomically removing one element and inserting two others. If your application needs to do that, then you will need to implement your own locking protocol to synchronize access to a shared data structure. Understanding race conditions is crucial. While the examples in these notes consider race conditions on simple data structures like stacks, in larger applications using standard libraries for concurrent data structures will still often have bad interleavings at higher levels of abstraction.

9 Deadlock

There is another common concurrency bug that you must avoid when writing concurrent programs: If a collection of threads are blocked forever, all waiting for another thread in the collection to do something (which it won't because it's blocked), we say the threads are *deadlocked*. Deadlock is a different kind of bug from a race condition, but unfortunately preventing a potential race condition can cause a potential deadlock and vice-versa. Deadlocks typically result only with some, possibly-rare, thread schedules (i.e., like race conditions they are non-deterministic). One advantage compared to race conditions is that it is easier to tell a deadlock occurred: Some threads never terminate because they are blocked.

As a canonical example of code that could cause a deadlock, consider a bank-account class that includes a method to transfer money to another bank-account. With what we have learned so far, the following WRONG code skeleton looks reasonable:

```
class BankAccount {
  ...
  void withdraw(int amt) { lock guard(m); ... }
  void deposit(int amt) { lock guard(m); ... }
  void transferTo(int amt, BankAccount & other) {
    lock guard(m);
    withdraw(amt);
    other.deposit(amt);
  }
}
```

This class uses a fine-grained locking strategy where each account has its own lock. There are no data races because the only methods that directly access any fields are (we assume) synchronized like `withdraw` and `deposit`, which acquire the correct lock. More subtly, the `transferTo` method appears atomic. For another thread to see the intermediate state where the withdraw has occurred but not the deposit would require operating on the withdrawn-from account. Since `transferTo` is synchronized, this cannot occur.

Nonetheless, there are interleavings of two `transferTo` operations where neither one ever finishes, which is clearly not desired. For simplicity, suppose x and y are global variables each holding a `BankAccount`. Consider this interleaving in PSEUDOCODE:

```
Thread 1: x.transferTo(1,y)      Thread 2: y.transferTo(1,x)
-------------------------         --------------------------
acquire lock for x
withdraw 1 from x

                                 acquire lock for y
                                 withdraw 1 from y
                                 block on lock for x

block on lock for y
```

In this example, one thread transfers from one account (call it "A") to another (call it "B") while the other thread transfers from B to A. With just the wrong interleaving, the first thread holds the lock for A and is blocked on the lock

54

for B while the second thread holds the lock for B and is blocked on the lock for A. So both are waiting for a lock to be released by a thread that is blocked. They will wait forever.

More generally, a deadlock occurs when there are threads $T_1, T_2, ..., T_n$ such that:

- For $1 \leq i \leq (n-1)$, T_i is waiting for a resource held by T_{i+1}

- T_n is waiting for a resource held by T_1

In other words, there is a cycle of waiting.

Returning to our example, there is, unfortunately, no obvious way to write an atomic `transferTo` method that will not deadlock. Depending on our needs we could write:

```
void transferTo(int amt, BankAccount & other) {
  withdraw(amt);
  other.deposit(amt);
}
```

All we have done is make `transferTo` not be synchronized. Because `withdraw` and `deposit` are still synchronized, the only negative effect is that an intermediate state is potentially exposed: Another thread that retrieved the balances of the two accounts in the transfer could now "see" the state where the withdraw had occurred but not the deposit. If that is okay in our application, then this is a sufficient solution. If not, another approach would be to resort to coarse-grained locking where all accounts use the same lock.

Either of these approaches avoids deadlock because they ensure that no thread ever holds more than one lock at a time. This is a sufficient but not necessary condition for avoiding deadlock because it cannot lead to a cycle of waiting. A much more flexible sufficient-but-not-necessary strategy that subsumes the often-unworkable "only one lock at a time strategy" works as follows: For all pairs of locks x and y, either do not have a thread ever (try to) hold both x and y simultaneously *or* have a globally agreed upon order, meaning either x is always acquired before y or y is always acquired before x.

This strategy effectively defines a conceptual partial order on locks and requires that a thread tries to acquire a lock only if all locks currently held by the thread (if any) come earlier in the partial order. It is merely a programming convention that the entire program must get right.

To understand how this ordering idea can be useful, we return to our `transferTo` example. Suppose we wanted `transferTo` to be atomic. A simpler way to write the code that still has the deadlock problem we started with is:

```
void transferTo(int amt, BankAccount & other) {
  lock guard(m);
  lock guard(other.m);
  withdraw(amt);
  other.deposit(amt);
}
```

Here we make explicit that we need the locks guarding both accounts and we hold those locks throughout the critical section. This version is easier to understand and the potential deadlock — still caused by another thread performing a transfer to the accounts in reverse order — is also more apparent. (Note in passing that a deadlock involving more threads is also possible. For example, thread 1 could transfer from A to B, thread 2 from B to C, and thread 3 from C to A.)

The critical section has two locks to acquire. If all threads acquired these locks in the same order, then no deadlock could occur. In our deadlock example where one thread calls x.transferTo(1,y) and the other calls y.transferTo(1,x), this is exactly what goes wrong: one acquires x before y and the other y before x. We want one of the threads to acquire the locks in the other order.

In general, there is no clear way for us to do this, but sometimes our data structures have some unique identifier that lets us pick an arbitrary order to prevent deadlock. Suppose, as is the case in actual banks, that every `BankAccount` already has an `acctNumber` field of type `int` that is distinct from every other `acctNumber`. Then we can use these numbers to require that threads always acquire locks for bank accounts in increasing order. (We could also require decreasing order, the key is that all code agrees on one way or the other.) We can then write `transferTo` like this:

```
class BankAccount {
  ...
  int acctNumber; // must be unique
public:
  void transferTo(int amt, BankAccount & other) {
    if(acctNumber < other.acctNumber) {
      lock guard(m);
      lock guard(other.m);
      withdraw(amt);
      other.deposit(amt);
    }
    else {
      lock guard(other.m);
      lock guard(m);
      withdraw(amt);
      other.deposit(amt);
    }
  }
}
```

While it may not be obvious that this code prevents deadlock, it should be obvious that it follows the strategy described above. And *any* code following this strategy will not deadlock because we cannot create a cycle if we acquire locks according to the agreed-upon partial order.

Fortunately, there are other situations where preventing deadlock amounts to fairly simple rules about the order that locks are acquired. Here are two examples:

- If you have two different types of objects, you can order the locks by the types of data they protect. For example, the documentation could state, "When moving an item from the hashtable to the work queue, never try to acquire the queue's lock while holding the hashtable's lock (the other order is acceptable)."

- If you have an acyclic data structure like a tree, you can use the references in the data structure to define the locking order. For example, the documentation could state, "If holding a lock for a node in the tree, do not acquire other nodes' locks unless they are descendants in the tree."

At this point, we pause to repeat some of our most important advice: reasoning about concurrency is hard; so, take advantage of solutions that have already been created for you. C++11 offers the std::lock function to lock one or more locks without deadlock. Use it!

10   Additional Synchronization Primitives

So far, the synchronization primitives we have seen are (reentrant) locks, join, and atomic fields. This section describes two more useful primitives — reader/writer locks and condition variables — in detail, emphasizing how they provide facilities that basic locks do not. Then Section 10.3 briefly mentions some other primitives.

10.1   Reader/Writer Locks

These notes have emphasized multiple times that it is *not* an error for multiple threads to read the same information at the same time. Simultaneous reads of the same field by any number of threads is not a data race. Immutable data does not need to be accessed from within critical sections. But as soon as there might be concurrent writes, we have used locks that allow only one thread at a time. *This is unnecessarily conservative.* Considering all the data guarded by a lock, it would be fine to allow multiple simultaneous *readers* of the data provided that any *writer* of the data does so exclusively, i.e., while there are neither concurrent readers nor writers. In this way, we still prevent any data race or bad interleaving resulting from read/write or write/write errors.

As an example where this would be useful, consider a dictionary such as a hashtable protected by a single lock (i.e., simple, coarse-grained locking). Suppose that lookup operations are common but insert or delete operations are very rare. As long as lookup operations do not actually mutate any elements of the dictionary, it is fine to allow them to proceed in parallel and doing so avoids unnecessary contention. (If on the other hand, lookups cause data-structure manipulations, as they might with a move-to-front list or a splay tree, then this will not work. This is another example of the advantage of not mutating shared memory unnecessarily.) When an insert or delete operation does arrive, it would need to block until there were no other operations active (lookups or other operations that mutate the dictionary), but (a) that is rare and (b) that is what we expect from coarse-grained locking. In short, we would like to support simultaneous concurrent reads. Fine-grained locking might help with this, but only indirectly and incompletely (it still would not help with reads of the same dictionary key), and it is much more difficult to implement.

Instead a *reader/writer lock* provides exactly what we are looking for as a primitive. It has a different interface than a regular lock: A reader/writer lock is acquired either "to be a reader" or "to be a writer." The lock allows multiple threads to hold the lock "as a reader" at the same time, but only one to hold it as a writer and only if there are no holders as readers. In particular, here are the operations:

- `new` creates a new lock that initially has "0 readers" and is "unheld for writing"

- `acquire_write` blocks if the lock is "held for writing" there are currently any readers or writers, else it makes the lock "held for writing"

- `release_write` returns the lock to its initial state

- `acquire_read` blocks if the lock is currently "held for writing" else it increments the number of readers

- `release_read` decrements the number of readers, which may or may not return the lock to its initial state

A reader/writer lock will block acquire operations as necessary to uphold the following invariants, where we write $|r|$ and $|w|$ for the number of threads holding the lock for reading and writing, respectively.

- $0 \le |w| \le 1$

- $|w| * |r| = 0$

Note that the name *reader/writer lock* is really a misnomer. Nothing about the definition of the lock's operations has anything to do with reading or writing. A better name might be a *shared/exclusive lock*: Multiple threads can hold the lock in "shared mode" but only one can hold it in "exclusive mode" and only when no threads hold it in "shared mode." It is up to the programmer to use the lock correctly, i.e., to ensure that it is okay for multiple threads to proceed simultaneously when they hold the lock for reading. And the most sensible way to ensure this is to read memory but not write it. (Intel's `concurrent_hash_map` uses method overloading and C++'s `const` specifier to enforce the correspondence between the "shared lock" and reader mode, although it's easy to accidentally grab the exclusive lock even in code that doesn't intend to change data.)

Returning to our dictionary example, using a reader/writer lock is entirely straightforward: The lookup operation would acquire the coarse-grained lock for reading, and other operations (insert, delete, resize, etc.) would acquire it for writing. If these other operations are rare, there will be little contention.

There are a couple details related to the semantics of reader/writer locks worth understanding. To learn how a particular library resolves these questions, you need to read the documentation. First, some reader/writer locks give *priority* to writers. This means that an "acquire for writing" operation will succeed before any "acquire for reading" operations that *arrive later*. The "acquire for writing" still has to wait for any threads that *already* hold the lock for reading to release the lock. This priority can prevent writers from *starving* (blocking forever) if readers are so common or have long enough critical sections that the number of threads wishing to hold the lock for reading is never 0.

Second, some libraries let a thread *upgrade* from being a reader to being a writer without releasing the lock. Upgrading might lead to deadlock though if multiple threads try to upgrade.

Third, just like regular locks, a reader/writer lock may or may not be reentrant. Aside from the case of upgrading, this is an orthogonal issue.

Interestingly, while C++'s Boost library—the proving ground for many of C++11's features—included reader/writer locks using `shared_mutex` and `shared_lock`, C++11 does *not* include them for various reasons including performance concerns. On the one hand, locking with the shared locks is significantly more expensive *even for readers*. On the other hand, cache coherency concerns—keeping cached data consistent across multiple processors—may reduce performance further or effectively serialize readers.[15]

10.2   Condition Variables

Condition variables are a mechanism that lets threads *wait* (block) until another thread *notifies* them that it might be useful to "wake up" and try again to do whatever could not be done before waiting. (We assume the thread that waited chose to do so because some *condition* prevented it from continuing in a useful manner.) It can be awkward to use condition variables correctly, but most uses of them are within standard libraries, so arguably the most useful thing is to understand when and why this wait/notification approach is desirable.

To understand condition variables, consider the canonical example of a *bounded buffer*. A bounded buffer is just a queue with a maximum size. Bounded buffers that are shared among threads are useful in many concurrent programs. For example, if a program is conceptually a pipeline (an assembly line), we could assign some number of threads to each stage and have a bounded buffer between each pair of adjacent stages. When a thread in one stage produces an object for the next stage to process, it can enqueue the object in the appropriate buffer. When a thread in the next stage is ready to process more data, it can dequeue from the same buffer. From the perspective of this buffer, we call the enqueuers *producer threads* (or just *producers*) and the dequeuers *consumer threads* (or just *consumers*).

Naturally, we need to synchronize access to the queue to make sure each object is enqueued/dequeued exactly once, there are no data races, etc. One lock per bounded buffer will work fine for this purpose. More interesting is what to do if a producer encounters a full buffer or a consumer encounters an empty buffer. In single-threaded programming, it makes sense to throw an exception when encountering a full or empty buffer since this is an unexpected condition that cannot change. But here:

- If the thread simply waits and tries again later, a producer may find the queue no longer full (if a consumer has dequeued) and a consumer may find the queue no longer empty (if a producer has enqueued).

- It is not unexpected to find the queue in a "temporarily bad" condition for continuing. The buffer is for managing the handoff of data and it is entirely natural that at some point the producers might get slightly ahead (filling the buffer and wanting to enqueue more) or the consumers might get slightly ahead (emptying the buffer and wanting to dequeue more).

The fact that the buffer is bounded is useful. Not only does it save space, but it ensures the producers will not get too far ahead of the consumers. Once the buffer fills, we want to stop running the producer threads since we would rather use our processors to run the consumer threads, which are clearly not running enough or are taking longer. So we really do want waiting threads to stop using computational resources so the threads that are not stuck can get useful work done.

With just locks, we can write a version of a bounded buffer that never results in a wrong answer, but it will be INEXCUSABLY INEFFICIENT because threads will not stop using resources when they cannot proceed. Instead they keep checking to see if they can proceed:

```
template <typename E>
class Buffer {
  // not shown: an array of fixed size for the queue with two indices
  // for the front and back, along with methods isEmpty() and isFull()
  // and a recursive_mutex m belonging to the class plus a typedef
  // for the preferred lock type as "lock".
  void enqueue(E elt) {
    while(true) {
```

---

[15] Anthony Williams discusses this rationale on Boost's listserv here: `http://permalink.gmane.org/gmane.comp.lib.boost.devel/211180`.

```
      lock guard(m);
      if(!isFull()) {
        ... do enqueue as normal ...
        return;
      }
    }
  }
  E dequeue() {
    while(true) {
      lock guard(m);
      if(!isEmpty()) {
        E ans = ... do dequeue as normal ...
        return ans;
      }
    }
  }
};
```

The purpose of condition variables is to avoid this *spinning*, meaning threads checking the same thing over and over again until they can proceed. This spinning is particularly bad because it causes contention for exactly the lock that other threads need to acquire in order to do the work before the spinners can do something useful. Instead of constantly checking, the threads could "sleep for a while" before trying again. But how long should they wait? Too little waiting slows the program down due to wasted spinning. Too much waiting is slowing the program down because threads that could proceed are still sleeping.

It would be best to have threads "wake up" (unblock) exactly when they might usefully continue. In other words, if producers are blocked on a full queue, wake them up when a dequeue occurs and if consumers are blocked on an empty queue, wake them up when an enqueue occurs. This cannot be done correctly with just locks, which is exactly why condition variables exist.

C++11 provides a class `condition_variable` that supports the operations described above.[16] There are three key methods of the `condition_variable` class that we focus on:

- `wait` takes a `unique_lock` that must be held by the caller. The call will atomically (a) block the thread, (b) release the lock, and (c) register the thread as "waiting" to be notified. (This is the step you cannot reasonably implement on your own; there has to be no "gap" between the lock-release and registering for notification else the thread might "miss" being notified and never wake up.) When the thread is later woken up it will re-acquire the same lock before `wait` returns. (If other threads also seek the lock, then this may lead to more blocking. There is no guarantee that a notified thread gets the lock next.) Notice the thread then continues as part of a *new* critical section; the entire point is that the state of shared memory has changed in the meantime while it did not hold the lock.

- `notify_one` wakes up *one* thread that is blocked on this condition variable (i.e., has called `wait` on the same object). If there are no such threads, then `notify_one` has no effect, but this is fine. (And this is why there has to be no "gap" in the implementation of `wait`.) If there are multiple threads blocked, there is no guarantee which one is notified.

- `notify_all` is like `notify_one` except it wakes up all the threads blocked on this condition variable.

The term *wait* is standard, but the others vary in different languages. Sometimes *notify_one* is called *signal* or *pulse*. Sometimes *notify_all* is called *broadcast* or *pulseAll*.

Here is a WRONG use of these primitives for our bounded buffer. It is close enough to get the basic idea. Identifying the bugs is essential for understanding how to use condition variables correctly.

---

[16]Warning: `condition_variable` cannot be used with `lock_guard`, since the lock used with a condition variable must be releasable, and `lock_guards` are not. Use the very similar `unique_lock` instead. The same warning applies for `condition_variable_any`, even though it can be used with more types than `condition_variable`.

```
template <typename E>
class Buffer {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   typedef std::recursive_mutex mutex;
   typedef std::unique_lock<mutex> lock;  // NOTE: not using lock_guard
   mutex cv_m;
   std::condition_variable cv;
   void enqueue(E elt) {
      while(true) {
        lock guard(m);
        if(!isFull())
          cv.wait(guard);
        ... do enqueue as normal ...
        if(... buffer was empty (i.e., now has 1 element) ...)
          cv.notify_one();
        return;
      }
   }
   E dequeue() {
      while(true) {
        lock guard(m);
        if(!isEmpty())
          cv.wait(guard);
        E ans = ... do dequeue as normal ...
        if(... buffer was full (i.e., now has room for 1 element) ...)
          cv.notify_one();
        return ans;
      }
   }
};
```

Enqueue waits if the buffer is full. Rather than spinning, the thread will consume no resources until awakened. While waiting, it does not hold the lock (per the definition of wait), which is crucial so other operations can complete. On the other hand, if the enqueue puts one item into an empty queue, it needs to call notify_one in case there are dequeuers waiting for data. Dequeue is symmetric. Again, this is the *idea*—wait if the buffer cannot handle what you need to do and notify others who might be blocked after you change the state of the buffer—but this code has two serious bugs. It may not be easy to see them because thinking about condition variables takes some getting used to.

The first bug is that we have to account for the fact that after a thread is notified that it should try again, but before it actually tries again, the buffer can undergo other operations by other threads. That is, there is a "gap" between the notification and when the notified thread actually re-acquires the lock to begin its new critical section. Here is an example bad interleaving with the enqueue in Thread 1 doing the wrong thing; a symmetric example exists for dequeue.

```
Initially the buffer is full (so Thread 1 blocks)

Thread 1 (enqueue)           Thread 2 (dequeue)        Thread 3 (enqueue)
------------------           ------------------        -------------------
if(isFull())
   cv.wait();
                             ... do dequeue ...
                             if(... was full ...)
```

```
                                      cv.notify_one();
                                                    if(isFull()) (not full: don't wait)

                                                    ... do enqueue ...

... do enqueue ... (wrong!)
```

Thread 3 "snuck in" and re-filled the buffer before Thread 1 ran again. Hence Thread 1 adds an object to a full buffer, when what it should do is wait again. The fix is that Thread 1, after (implicitly) re-acquiring the lock, must again check whether the buffer is full. A second if-statement does not suffice because if the buffer is full again, it will wait and need to check a third time after being awakened and so on. Fortunately, we know how to check something repeatedly until we get the answer we want: a while-loop. So this version fixes this bug, but is still WRONG:

```cpp
template <typename E>
class Buffer {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   typedef std::recursive_mutex mutex;
   typedef std::unique_lock<mutex> lock;  // NOTE: not using lock_guard
   mutex cv_m;
   std::condition_variable cv;
   void enqueue(E elt) {
      lock guard(m);
      while (!isFull())
        cv.wait(guard);
      ... do enqueue as normal ...
      if (... buffer was empty (i.e., now has 1 element) ...)
        cv.notify_one();
   }
   E dequeue() {
      lock guard(m);
      while (!isEmpty())
        cv.wait(guard);
      E ans = ... do dequeue as normal ...
      if (... buffer was full (i.e., now has room for 1 element) ...)
        cv.notify_one();
      return ans;
   }
};
```

The only change is using a while-loop instead of an if-statement for deciding whether to wait. *Calls to* wait *should always be inside while-loops that re-check the condition.*[17] Not only is this in almost every situation the right thing to do for your program, but technically for obscure reasons C++ is allowed to notify (i.e., wake up) a thread even if your program does not call a notify method! So it is mandatory to re-check the condition since it is possible that nothing changed. This is still much better than the initial spinning version because each iteration of the while-loop corresponds to being notified, so we expect very few (probably 1) iterations.

Unfortunately, the version of the code above is still wrong. It does not account for the possibility of multiple threads being blocked due to a full (or empty) buffer. Consider this interleaving:

Sending one notification on the first dequeue was fine at that point, but that still left a thread blocked. If the second dequeue happens before any enqueue, no notification will be sent, which leaves the second blocked thread waiting when it should not wait.

---

[17]This pattern is sufficiently common that C++11 provides direct support. wait can take a Predicate which it will then check repeatedly in exactly this style, returning only when the thread has the lock and the Predicate is true.

The simplest solution in this case is to change the code to use `notify_all` instead of `notify_one`. That way, whenever the buffer becomes non-empty, *all* blocked producers know about it and similarly whenever the buffer becomes non-full, *all* blocked consumers wake up. If there are n blocked threads and there are no subsequent operations like the second dequeue above, then $n-1$ threads will simply block again, but at least this is correct. So this is RIGHT:

```
template <typename E>
class Buffer {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   typedef std::recursive_mutex mutex;
   typedef std::unique_lock<mutex> lock;  // NOTE: not using lock_guard
   mutex cv_m;
   std::condition_variable cv;
   void enqueue(E elt) {
      lock guard(m);
      while (!isFull())
        cv.wait(guard);
      ... do enqueue as normal ...
      if (... buffer was empty (i.e., now has 1 element) ...)
        cv.notify_all();
   }
   E dequeue() {
      lock guard(m);
      while (!isEmpty())
        cv.wait(guard);
      E ans = ... do dequeue as normal ...
      if (... buffer was full (i.e., now has room for 1 element) ...)
        cv.notify_all();
      return ans;
   }
};
```

If we do not expect very many blocked threads, this `notify_all` solution is reasonable. Otherwise, it causes a lot of probably-wasteful waking up (just to have $n-1$ threads probably block again). Another tempting solution ALMOST works, but of course "almost" means "wrong." We could have every enqueue and dequeue call `notify_one`, not just if the buffer had been empty or full. (In terms of the code, just replace the if-statements with calls `cv.notify_one()`.) In terms of the interleaving above, the first dequeue would wake up one blocked enqueuer and the second dequeue would wake up another one. If no thread is blocked, then the `notify_one` call is unnecessary but does no harm.

The reason this is incorrect is subtle, but here is one scenario where the wrong thing happens. For simplicity, assume the size of the buffer is 1 (any size will do but larger sizes require more threads before there is a problem).

1. Assume the buffer starts empty.

2. Then two threads T_1 and T_2 block trying to dequeue.

3. Then one thread T_3 enqueues (filling the buffer) and calls `notify_one`. Suppose this wakes up T_1.

4. But before T_1 re-acquires the lock, another thread T_4 tries to enqueue and blocks (because the buffer is still full).

5. Now T_1 runs, emptying the buffer and calling `notify_one`, but *in a stroke of miserable luck this wakes up T_2 instead of T_4.*

6. T_2 will see an empty buffer and wait again. So now T_2 and T_4 are blocked even though T_4 should enqueue and then T_2 could dequeue.

The way to fix this problem is to use *two condition variables* — one for producers to notify consumers and one for consumers to notify producers. But we need both condition variables to be "associated with" the same lock, since we still need all producers and consumers to use one lock to enforce mutual exclusion.

To summarize, condition variables are important and subtle. Important because concurrent programs need ways to have threads wait without spinning until they are notified to try again. Locks are not the right tool for this. Subtle because bugs are more difficult to reason about and the correct programming idioms are less intuitive than with locks.

In practice, condition variables tend to be used in very stylized ways. Moreover, it is rare that you need to use condition variables explicitly rather than using a library that internally uses condition variables. For example, Intel's open source Threading Building Blocks library includes a class `concurrent_bounded_queue` that is exactly what we need for a bounded buffer. It is good to know that any blocking caused by calls to the library will be efficient thanks to condition variables, but we do not need to write or understand the condition-variable code.

10.3   Other

Locks and condition variables are not the only synchronization primitives. For one, do not forget that `join` synchronizes two threads in a particular way and is very convenient when it is the appropriate tool. You may encounter other primitives as you gain experience with concurrent programming. Your experience learning about locks should make it much easier to learn other constructs. Key things to understand when learning about synchronization are:

- What interleavings does a primitive prevent?

- What are the rules for using a primitive correctly?

- What are the standard idioms where a primitive proves most useful?

Another common exercise is to implement one primitive in terms of another, perhaps inefficiently. For example, it is possible to implement `join` in terms of locks: The helper thread can hold a lock until just before it terminates. The `join` primitive can then (try to) acquire and immediately release this lock. Such an *encoding* is poor style if `join` is provided directly, but it can help understand how `join` works and can be useful if you are in a setting where only locks are provided as primitives.

Here is an incomplete list of other primitives you may encounter:

- *Semaphores* are rather low-level mechanisms where two operations, called P and V for historical reasons, increment or decrement a counter. There are rules about how these operations are synchronized and when they block. Semaphores can be easily used to implement locks as well as barriers.

- *Barriers* are a bit more like `join` and are often more useful in parallel programming than concurrent programming. When threads get to a barrier, they block until n of them (where n is a property of the barrier) reach the barrier. Unlike `join`, the synchronization is not waiting for *one* other thread to *terminate*, but rather n other threads to *get to the barrier*.

- *Monitors* provide synchronization that corresponds more closely to the structure of your code. Like an object, they can encapsulate some private state and provide some entry points (methods). As synchronization, the basic idea is that only one thread can call any of the methods at a time, with the others blocking.

A   Fully C++11 Initial Thread Example

Many of the features in C++11 work together to make thread programming simpler. Here's the `main` function of the first threading example rewritten to illustrate this:

```
std::ios_base::sync_with_stdio(true);

std::vector<std::thread> threads;
for (int i = 1; i <= 20; i++)
```

```
    threads.push_back(std::thread([i]() {
        std::cout << "Thread " << i << " says hi!" << std::endl;
        std::cout << "Thread " << i << " says bye!" << std::endl;
    }));

  for (auto & thread : threads)
    thread.join();

  return 0;
```

Rather than passing in a function written separately, we use a lambda (an anonymous function) to create a new function at the call site. If you're familiar with lambdas, read on in this paragraph. We can specify the variables to close over and whether to close over them by value or by reference in the `[...]`. In addition, we can give a default semantics (value `[=]` or reference `[&]`), and the lambda will automatically close over any variables used but not explicitly specified using the default semantics given.

Rather than looping over the `threads` vector explicitly or even using the `algorithm` library's `for_each` algorithm, we use the new syntactic sugar for a for-each style loop. Also, rather than explicitly declaring the type of the iterator variable, we use `auto` to have C++ deduce it for us. Note that at least some g++ C++0x variants do not support this syntax, in which case you can fall back to the `for_each` algorithm.

Finally, we haven't mentioned it explicitly, but the fact that we can pass a temporary `std::thread` object to the `push_back` method also benefits from C++'s new rvalue reference parameters which allow the freshly created and about-to-be-destroyed thread object to be "moved" into the list rather than copied or shared.

And, extra finally, if you happen to have a C++ runtime that makes individual stream insertion operations on `std::cout` atomic like the one we're using, you can avoid breaking up lines in this example by using the `sstream` library's `std::stringstream` class to collect up everything you want to print before printing it:

```
        std::stringstream strout;
        strout << "Thread " << i << " says hi!" << std::endl;
        std::cout << strout.str();
```

Again, the C++11 specification makes no guarantees that this will actually make your printout atomic!

In fact, C++11 has substantial advanced support for launching parallel threads and managing concurrent access. Here's a simple example—based on the `async` documentation at `en.cppreference.com`—that uses several C++11 features to calculate a sum in parallel. We could call this on an array `arr` with `len` elements as `sum(arr, arr+len)` or on a `vector v` (or other iterable container) as `sum(begin(v), end(v))`.

```
// Include the future and algorithm headers for async and accumulate.
// The begin/end style is common in C++ algorithms.
template <typename Iter>
int sum(Iter begin, Iter end) {
  auto len = end - begin;
  if (len < SEQUENTIAL_CUTOFF)
    return std::accumulate(begin, end, 0);

  auto mid = begin + (end-begin)/2;
  auto pending_result = std::async(std::launch::async, &sum<Iter>, begin, mid);
  auto right = sum(mid, end);
  auto left = pending_result.get(); // blocks on the result of the first call
  return left + right;
}
```