Chapter 1

Introduction

The topic of this book might best be described as **problem solving with** $\underset{\wedge}{\textbf{\textit{parallel}}}$ **computers**. The idea is you have some problem to solve (e.g. finding the shortest path from your room to your first class), and you want to use your computer to solve it for you. Your primary concern is probably that your answer is correct (e.g. you would likely be unhappy to find yourself at the wrong class). However, you also care that you can get the answer reasonably quickly (e.g. it would not be useful if your computer had to think about it until next semester).

This book is therefore about different aspects of problem solving with computers. It is about defining precisely the problem you want to solve. It is about learning the different techniques that can be used to solve such a problem, and about designing algorithms using these techniques. It is about designing abstract data types that can be used in these algorithms, and data structures that implement these types. And, it is about analyzing the cost of those algorithms and comparing them based on their cost.

Unlike traditional books on algorithms and data structures, which are concerned with sequential algorithms (ones that are correct and efficient on sequential computers), in this book we are concerned with parallel algorithms (ones that are correct and efficient on parallel computers). However, in the approach we take in this book , sequential and parallel algorithms are not that different. Indeed the book covers most of what is covered in a traditional sequential algorithms course. In the rest of this chapter we discuss why it is important to study parallelism, why it is important to separate interfaces from implementations, and outline some algorithm-design techniques.

1.1 Parallelism

The term “parallelism” or “parallel computing” refers the ability to run multiple computations (tasks) at the same time.

Parallel systems. Today parallelism is available in all computer systems, and at many different scales starting with parallelism in the nano-circuits that implement individual instructions, and working the way up to parallel systems that occupy large data centers. Since the early 2000s hardware manufacturers have been placing multiple processing units, often called "cores", onto a single chip. These cores can be general purpose processors, or more special purpose processors, such as those found in ***Graphics Processing Units*** (GPUs). Each core can run in parallel with the others. At the larger scale many such chips can be connected by a network and used together to solve large problems. For example, when you perform a simple search on the Internet, you engage a data center with thousands of computers in some part of the world, likely near your geographic location. Many of these computers (perhaps as many as hundreds, if not thousands) take up your query and sift through data to give you an accurate response as quickly as possible.

There are several reason for why such parallel systems and thus parallelism has become so important. First, parallelism is simply more powerful than sequential computing, where only one computation can be run at a time, because it enables solving more complex problems in shorter time. For example,an Internet search is not as effective if it cannot be completed at "interactive speeds", completing in several milliseconds. Similarly, a weather-forecast simulation is essentially useless if it cannot be completed in time.

The second reason is efficiency in terms of energy usage. As it turns out, performing a computation twice as fast sequentially requires eight times as much energy. Precisely speaking, energy consumption is a cubic function of clock frequency (speed). With parallelism we don't need more energy to speed up a computation, at least in principle. For example, to perform a computation in half the time, we need to divide the computation into two parallel sub-computations, perform them in parallel and combine their results. This can require as little as half the time as the sequential computation while consuming the same amount of energy. In reality, there are some overheads and we will need more energy, for example, to divide the computation and combine the results. Such overheads are usually small, e.g., constant fraction over sequential computation, but can be larger. These two factors—time and energy—have become increasingly important in the last decade, catapulting parallelism to the forefront of computing.

Example 1.1. As is historically popular in explaining algorithms, we can establish an analogy between parallel algorithms and cooking. As in a kitchen with multiple cooks, in parallel algorithms you can do things in parallel for faster turnaround time. For example, if you want to prepare 3 dishes with a team of cooks you can do so by asking each cook to prepare one. Doing so will often be faster that using one cook. But there are some overheads, for example, the work has to be divided as evenly as possible. Obviously, you also need more resources, e.g., each cook might need their own kitchen utensils.

Parallel software. The important advantage of using a parallel instead of a sequential algorithm is the ability to perform sophisticated computations quickly enough to make them practical or relevant, without consuming large amounts of energy.

One way to quantify this advantage is to measure the performance improvements of parallelism. Example 1.2 illustrates the sort of performance improvements that can achieved today. These times are on a 32 core commodity server machine. In the table, the sequential timings

Example 1.2. Example timings (reported in seconds) for some algorithms on 1 and on 32 cores.

	Sequential	**Parallel**	
		1-core	32-core
Sorting 10 million strings	2.9	2.9	.095
Remove duplicates for 10 million strings	.66	1.0	.038
Minimum spanning tree for 10 million edges	1.6	2.5	.14
Breadth first search for 10 million edges	.82	1.2	.046

use sequential algorithms while the parallel timings use parallel algorithms. Notice that the ***speedup*** for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (minimum spanning tree) to approximately 32 (sorting).

Challenges of parallel software. It would be convenient to use sequential algorithms on parallel computers, but this does not work well because parallel computing requires a different way of organizing the computation. The fundamental difference between sequential and parallel computation is that in the latter certain computations will be performed at the same time but this is possible only if the computations are actually ***independent***, i.e., do not depend on each other. Thus when designing a parallel algorithm, we have to identify the underlying dependencies in the computation to be performed and avoid creating unnecessary dependencies.

Example 1.3. Going back to or cooking example, suppose that we want to make a frittata in our kitchen with 3 cooks. Making a frittata is quite a bit more involved than just boiling eggs. We have to be careful about the dependencies between various tasks. For example, vegetables cannot be sauteed before they are washed and chopped the eggs cannot be added to the meal before being broken or before the vegetables are sauteed, etc.

An important challenge is therefore to design algorithms that minimize the dependencies so that more things can run in parallel. This design challenge is a primary focus of this book.

Another important challenge concerns the coding and usage of a parallel algorithm in the real world. The many forms of parallelism, ranging from small to large scale, and from general to special purpose, has led to many different programming languages and system for coding parallel algorithms. These different programming languages systems often target a particular kind of hardware, and even a particular kind of problem domain. For example, there are separate systems for coding parallel numerical algorithms on shared memory hardware, for coding graphics algorithms on Graphical Processing Units (GPUs), and for coding data-analytics software on a distributed system. Each such system tends to have its own programming interface, its own cost model, and its own optimizations, making it practically impossible to take a parallel algorithm and code it once and for all for all possible applications. As it turns out, one can easily spend weeks or even months optimizing a parallel sorting algorithm on specific parallel hardware, such as a GPU.

Maximizing speedup by coding and optimizing an algorithm is not the goal of this book. Instead, our goal is to cover general design principles for parallel algorithms that can be applied in essentially all parallel systems, from the data center to the multicore chips on mobile phones. We will learn to think about parallelism at a high-level, learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. The focus is on understanding when things can run in parallel, and when not due to dependencies. There is much more to learn about parallelism, and we hope you continue studying this subject.

1.2 Work and Span

In this book we analyze the cost of algorithms in terms of two measures: ***work*** and ***span***. Together these measures capture both the sequential time and the parallelism available in an algorithm. We typically analyze both of these asymptotically, using for example the big-O notation, which will be described in more detail in Chapter 4.

The ***work*** of an algorithm corresponds to the total number of primitive operations performed by an algorithm. If running on a sequential machine, it corresponds to the sequential time. On a parallel machine, however, work can be split among multiple processors and thus reduce the time.

The interesting question is to what extent can the work be shared. Ideally we would like the work to be evenly shared. If we had W work and P processors to work on it in parallel, then even sharing would imply each processor does $\frac{W}{P}$ work, and hence the total time is $\frac{W}{P}$. An algorithm that achieves such ideal sharing is said to have ***perfect speedup***. Perfect speedup, however, is not always possible. If our algorithm is fully sequential (each operation depends on prior operations, leaving no room for parallelism), for example, we can only take advantage of one processor, and the time would not be improved at all by adding more. There is no sharing—-at least in parallel. More generally, when executing an algorithm in parallel, we cannot break dependencies, if a task depends on another task, we have to complete them in order.

The second measure, ***span***, enables analyzing to what extent the work of an algorithm can be split among processors. The ***span*** of an algorithm basically corresponds to the longest sequence of dependences in the computation. It can be thought of the time an algorithm would take if we had an unlimited number of processors on an ideal machine.

As we shall see in Section 4.2.4, even though work and span, are abstract machine-independent models of cost, they can be used to predict the run-time on any number of processors. Specifically, if for an algorithm the work dominates, i.e., is much larger than, span, then we expect the algorithm to deliver good speedups.

Example 1.4. As an example, consider the parallel `mergeSort` algorithm for sorting a sequence of length n. The work is the same as the sequential time, which you might know is

$$W(n) = O(n \lg n).$$

In Chapter 8 we will see that the span for `mergeSort` is

$$S(n) = O(\lg^2 n).$$

Thus, when sorting a million keys, work is $10^6 \lg(10^6) > 10^7$, and span is $\lg^2(10^6) < 500$. This means that we would expect to get good (close to perfect) speedups when using a small to moderate number of processors, e.g., couple of tens or hundreds, because the work term will dominate. We should note that in practice, the numbers might be more conservative due to natural overheads of parallel execution.

In this book we calculate the work and span of algorithms in a very simple way that just involves composing costs across subcomputations. Basically we assume that sub-computations are either composed sequentially (one must be performed after the other) or in parallel (they can be performed at the same time). We then calculate the work as the sum of the work of the subcomputations and the span as the sum of the span of sequential subcomputations or maximum of the work of the parallel subcomputations. More concretely, given two subcomputations, we can calculate the work and the span of their sequential and parallel composition as follows.

	W **(Work)**	S **(span)**
Sequential composition	$1 + W_1 + W_2$	$1 + S_1 + S_2$
Parallel composition	$1 + W_1 + W_2$	$1 + \max(S_1, S_2)$

In the table, W_1 and S_1 are the work and span of the first subcomputation and W_2 and S_2 of the second. The 1 that is added to each rule is the cost of composing the subcomputations.

The intuition behind these rules is that work simply adds, whether we perform computations sequentially or in parallel. The span, however, only depends on the span of the maximum of the two parallel computations. It might help to think of work as the total energy consumed by a computation and span as the minimum possible time that the computation requires. Regardless of whether computations are performed serially or in parallel, energy is equally required; time, however, is determined only by the slowest computation.

Example 1.5. Suppose that we have 30 eggs to cook using 3 cooks. Whether all 3 cooks to do the cooking or just one, the total work remains unchanged: 30 eggs need to be cooked. Assuming that cooking an egg takes 5 minutes, the total work therefore is 150 minutes. The span of this job corresponds to the longest sequence of dependences that we must follow. Since we can, in principle, cook all the eggs at the same time, span is 5 minutes.

Given that we have 3 cooks, how much time do we actually need? The greedy scheduling principle tells us that we need no more that $150/3 + 5 = 55$ minutes. That is almost a factor 3 speedup over the 150 that we would need with just one cook.

How do we actually realize the greedy schedule? In this case, this is simple, all we have to do is divide the eggs equally between our cooks.

If algorithm A has less work than algorithm B, but has greater span then which algorithm is better? In analyzing sequential algorithms there is only one measure so it is clear when one algorithm is asymptotically better than another, but now we have two measures. In general the work is more important than the span. This is because the work reflects the total cost of the computation (the processor-time product). Therefore typically the goal is to first reduce the work and then reduce the span by designing asymptotically work-efficient algorithms that perform no work than the best sequential algorithm for the same problem. However, sometimes it is worth giving up a little in work to gain a large improvement in span.

Definition 1.6. [Work Efficiency] We say that a parallel algorithm is ***asymptotically work efficient*** or, simply ***work efficient***, if the work is asymptotically the same as the time for an optimal sequential algorithm that solves the same problem.

For example, the parallel mergeSort described in Example 1.4 is work efficient since it does $O(n \log n)$ work, which optimal time for comparison based sorting. In this course we will try to develop work-efficient or close to work-efficient algorithms.

1.3 Specification, Problem, Implementation

Problem solving in computer science requires reasoning precisely about problems being studied and the properties of solutions. To facilitate such reasoning, in this book, we define problems by specifying them and describe the desired properties of solutions at different levels of abstraction, such as the cost and the implementation of the solution.

In this book, we are usually interested in two distinct classes of problems: algorithms problems and data structures problems.

Algorithm Specification. We specify an algorithm by describing what is expected of the algorithm via an ***algorithm specification***. For example, we can specify a sorting algorithm for sequences with respect to a given comparison function as follows.

Algorithm Specification 1.7. [Comparison Sorting] Given a sequence A of n elements taken from a totally ordered set with comparison operator $\leq$, return a sequence B containing the same elements but such that $B[i] \leq B[j]$ for $0 \leq i < j < n$.

The specification describes ***what*** the algorithm should do but it does not describe ***how*** it achieves what is asked. This is intentional because there can be many algorithms that meet a specification. A crucial property of any algorithm is its resource requirements or its ***cost***. For example, of the many ways algorithms for sorting a sequence, we may prefer some over the others. We specify the cost of class of algorithms with a ***cost specification***. For example, the following cost specification states that a particular class of parallel sorting algorithms performs $O(n \log n)$ work and $O(\log^2 n)$ span.

Cost Specification 1.8. [Comparison Sort: Work-Efficient and Parallel] Assuming the comparison function $<$ does constant work, the cost for parallel comparison sorting a sequence of length n is $O(n \log n)$ work and $O(\log^2 n)$ span.

There can be many cost specifications for sorting. For example, if we are not interested in parallelism, we can specify $O(n \log n)$ work but no bounds on the span. Here is another specification that requires even smaller span but allows for more work.

Cost Specification 1.9. [Comparison Sort: Fast Parallel] Assuming the comparison function $<$ does constant work, the cost for parallel comparison sorting a sequence of length n is $O(n^2)$ work and $O(\log n)$ span.

As we discussed, we usually care more about work and thus would prefer the first specification; there might, however, be cases where the second specification is preferable.

Data Structure Specification. We specify a data structure by describing what is expected of the data structure via an ***Abstract Data Type (ADT) specification***. For example, we can specify a priority queue ADT as follows.

Abstract Data Type 1.10. [Priority Queue] A priority queue consists of a priority queue type and supports three operations on values of this type. The operation `empty` returns an empty queue. The operation `insert` inserts a given value with a priority into the queue and returns the queue. The operation `removeMin` removes the value with the smallest priority from the queue and returns it.

As with algorithms, we usually give cost specifications to data structures. The following cost specification describes a class of basic priority queue data structures.

Cost Specification 1.11. [Priority Queue: Basic] The work and span of a priority queue operations are as follows.

- `create`: $O(1)$, $O(1)$.
- `insert`: $O(\log n)$, $O(\log n)$.
- `removeMin`: $O(\log n)$, $O(\log n)$.

Problem. A ***problem*** requires meeting an algorithm or an ADT specification and a corresponding cost specification. Since we allow specifying algorithms and data structures, we can distinguish between algorithms problems and data-structure problems. An ***algorithms problem*** requires designing an algorithm that satisfies the given algorithm specification and cost specification if any. A ***data-structures problem*** requires meeting an ADT specification by designing a data structure that can support the desired operations with the required efficiency specified by the cost specification. The difference between an algorithms problem and a data-structures problem is that the latter involves designing a data structure and a collection of algorithms, one for each operation, that operate on that data structure.

When we consider problems, it is usually clear from the context whether we are talking about algorithms or data structures. In such cases, we use the simpler terms ***specification*** and ***problem*** to refer to the algorithm/ADT specification and the corresponding problem respectively.

Implementation. We can solve an algorithms or a data-structures problem by presenting an ***implementation***. The term ***algorithm*** refers to an implementation that solves an algorithms problem and the term ***data structure*** to refer to an implementation that solves a data-structures problem. We note that while the distinction between problems and algorithms is common in the literature, the distinction between abstract data types and data structures is less so.

We describe an algorithm by using the pseudo-code notation based on SPARC, the language used in this book. For example, we can specify the classic insertion sort algorithm as follows.

Algorithm 1.12. [Insertion Sort]

```
insSort $f$ $s$ =
  if $|s|$ = 0 then $\langle\,\rangle$
  else insert $f$ $s$[0]  (insSort $f$ (s[1,...,n-1]))
```

In the algorithms, f is the comparison function and s is the input sequence. The algorithm uses a function (`insert`$f x s$) that takes the comparison function f, an element x, and a sequence s sorted by f, and inserts x in the appropriate place. Inserting into a sorted sequence is itself an algorithms problem, since we are not specifying how it is implemented, but just specifying its functionality. We might also be given a cost specification for `insert`, e.g., for a sequence of length n the cost of `insert` should be $O(n)$ work and $O(\log n)$ span. Given this cost we can determine the overall asymptotic cost of `sort` using our composition rules described in the last section. Since the code uses `insert` sequentially and since there are n inserts, the algorithm `insSort` has $n \times O(n) = O(n^2)$ work and $n \times O(\log n) = O(n \log n)$ span.

Similarly, we can specify a data structure by specifying the data type used by the implementation, and the algorithms for each operation. For example, we can implement a priority queue with a binary heap data structure and describe each operation as an algorithm that operates on this data structure. In other words, a data structure can be viewed as a collection of algorithms that operate on the same organization of the data.

Remark 1.13. [On the importance of specification] Several reasons underline the importance of distinguishing between specification and implementation. First, we want to be able to use a specification without knowing the details of an implementation that matches that specification. In many cases the specification of a problem is quite simple, but an efficient algorithm or data structure that solves it, i.e., the implementation, is complicated. Specifications allow us abstract from implementation details. Second, we want to be able to change or improve implementations over time. As long as each implementation matches the same specification, and the user relied only on the specification, then he or she can continue using the new implementation without worrying about their code breaking. Third, when we compare the performance of different algorithms or data structures it is important that we are not comparing apples with oranges. We have to make sure the algorithms we compare are solving the same problem, because subtle differences in the problem specification can make a significant difference in how efficiently that problem can be solved.

1.4 Problems

1-1 Essence of parallelism
When designing a parallel algorithm, we have to be careful in identifying the computations that can be performed in parallel. What is the key property of such computations?