Chapter 2

Parallelism

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

1 Parallel Hardware

Multicore Chips. 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. Today (in year 2018), multicore chips are used in essentially all computing devices ranging from mobile phones to desktop computers and servers.

Large-Scale Parallelism. At the larger scale, many computers 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. Because each computer can itself be parallel (e.g., built with multicore chips), the scale of parallelism can be quite large, e.g., in the thousands.

Fundamental Reasons for Why Parallelism Matters. There are several reasons for why parallelism has become prevalent over the past decade.

First, parallel computing is simply faster than sequential computing. This is important,

because many tasks must be completed quickly to be of use. For example, to be useful, an Internet search should complete in "interactive speeds" (usually below 100 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. Due to basic physics, performing a computation twice as fast sequentially requires eight times as much energy (energy consumption is a cubic function of clock frequency). With parallelism we don't need to use more energy than sequential computation, because energy is determined by the total amount of computation (work).

These two factors—time and energy—have become increasingly important in the last decade.

Example 2.1. Using two parallel computers, we can perform a computation in half the time of a sequential computer (operating at the same speed). To this end, 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. Because the total computation that we must do remains the same in both sequential and parallel cases, the total energy consumed is also the same.

The above reasoning holds in theory. In practice, there are overheads to parallelism: the speedup will be less than two-fold and more energy will be needed. For example, dividing the computation and combining the results could lead to additional overhead. Such overhead usually diminishes as the degree of parallelism increases but not always.

Example 2.2. 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 cooking pan.

Example 2.3 (Comparison to Sequential)**.** One way to quantify the advantages of parallelism is to compare its performance to sequential computation. The table below illustrates the sort of performance improvements that can achieved today. These timings are taken on a 32 core commodity server machine. In the table, the sequential timings 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).

Application	**Sequential**	**Parallel P = 1**	**Parallel P = 32**
Sort 10^7 strings	2.9	2.9	.095
Remove duplicates for 10^7 strings	.66	1.0	.038
Minimum spanning tree for 10^7 edges	1.6	2.5	.14
Breadth first search for 10^7 edges	.82	1.2	.046

2 Parallel Software

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 is that in parallel algorithms, computations must actually be ***independent*** to be performed in parallel. By independent we mean that computations do not depend on each other. Thus when designing a parallel algorithm, we have to identify the underlying dependencies in the computation and avoid creating unnecessary dependencies. This design challenge is an important focus of this book.

Example 2.4. Going back to our cooking example, suppose that we want to make a frittata in our kitchen with 4 cooks. Making a frittata is not easy. It involves cleaning and chopping vegetables, beating eggs, sauteeing, as well as baking. For the frittata to be good, the cooks must follow a specific receipe and pay attention to the dependencies between various tasks. For example, vegetables cannot be sauteed before they are washed, and the eggs cannot be fisked before they are broken!

Coding Parallel Algorithms. Another important challenge concerns the implementation and use of parallel algorithms in the real world. The many forms of parallelism, ranging from small to large scale, and from general to special purpose, have led to many different programming languages and systems for coding parallel algorithms. These different programming languages and systems often target a particular kind of hardware, and even a particular kind of problem domain. As it turns out, one can easily spend weeks or even months optimizing a parallel sorting algorithm on specific parallel hardware, such as a multicore chip, a GPU, or a large-scale massively parallel distributed system.

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.

Example 2.5. 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 possible applications. Indeed, it can require a significant effort to implement even a simple algorithm and optimize it to run well on a particular parallel system.

3 Work, Span, Parallel Time

This section describes the two measures—work and span—that we use to analyze algorithms. 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.

3.1 Work and Span

Work. 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 divided among multiple processors and thus does not necessarily correspond to time.

The interesting question is to what extent can the work be divided and performed in parallel. Ideally we would like to divide the work evenly. If we had W work and P processors to work on it in parallel, then even division would give each processor $\frac{W}{P}$ fraction of the work, and hence the total time would be $\frac{W}{P}$. An algorithm that achieves such ideal division is said to have ***perfect speedup***. Perfect speedup, however, is not always possible.

Example 2.6. A fully sequential algorithm, where each operation depends on prior operations leaves no room for parallelism. We can only take advantage of one processor and the time would not be improved at all by adding more.

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.

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

Definition 2.1 (Work and Span)**.** 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. For span, we differentiate between sequential and parallel composition: we calculate span as the sum of the span of sequential subcomputations or maximum of the span of the parallel subcomputations. More concretely, given two subcomputations with work W_1 and W_2 and span S_1 and S_2, we can calculate the work and the span of their sequential and parallel composition as follows. In calculating the overall work and span, the unit cost 1 accounts for the cost of (parallel or sequential) composition.

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

Note. The intuition behind the definition of work and span 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 2.7. 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? It should be clear that each cook can cook 10 eggs, for a total time of 50 minutes. Later we will discuss the "the greedy scheduling principle" which tells us that given a task with W work and S span, and using a greedy schedule, the time is upper bounded by $W/P + S$. In our case this would be $150/3 + 5 = 55$.

Example 2.8 (Parallel Merge Sort)**.** 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

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

We will see that the span for `mergeSort` is

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

Thus, when sorting a million keys and ignoring constant factors, work is $10^6 \lg(10^6) > 10^7$, and span is $\lg^2(10^6) < 500$.

Parallel Time. Even though work and span, are abstract measures of real costs, 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.

Exercise 2.1. How would you expect the parallel mergesort algorithm, `mergeSort`, mentioned in the example above to perform as we increase the number of processors dedicated to running it?

Solution. Recall that the work of parallel merge sort is $O(n \lg n)$, whereas the span is $O(\lg^2 n)$. Since span is much smaller than the work, 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 would expect for example the running time to halve when we double the number of processors. We should note that in practice, speedups tend to be more conservative due to natural overheads of parallel execution and due to other factors such as the memory subsystem that can limit parallelism.

3.2 Work Efficiency

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 more 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 2.2 (Work Efficiency)**.** We say that a parallel algorithm is ***(asymptotically) work efficient***, if the work is asymptotically the same as the time for an optimal sequential algorithm that solves the same problem.

Example 2.9. The parallel `mergeSort` function described in 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.