Due to physical and economical constraints, a typical machine we can buy now has 4 to 8 computing cores, and soon this number will be 16, 32, and 64. While the number of cores grows at a rapid pace, the per-core speed hasn’t increased much over the past several years. Additionally, graphics processing units (GPUs) are highly parallel platforms with hundreds of cores readily available in commodity machines today. This gives a compelling reason to study parallel algorithms. In this course, we will learn algorithmic techniques that allow you to design and analyze parallel algorithms.

For this lecture, we will learn how to write the pseudocode for parallel algorithms in this class, do some examples and look at how one analyze a parallel program written using this pseudocode. When we analyze the cost of an algorithm formally, we need to be reasonably precise in what model we are performing the analysis. As in CSE241/CSE247, we will use asymptotic costs. These costs can then be used to compare algorithms in terms of how they scale to large inputs. For example, as you know, some sorting algorithms use $O(n \log n)$ work and others $O(n^2)$. Clearly the $O(n \log n)$ algorithm scales better, but perhaps the $O(n^2)$ is actually faster on small inputs. In this class we are concerned with how algorithms scale, and therefore asymptotic analysis is indeed what we want.

1 Pseudocode for Parallel Programs

We are going to use pseudocode to write algorithms. This pseudocode is based on the programming language called Cilk Plus. Cilk Plus is an extension of C++ programming language and simply adds a few extra keywords so that you can specify the parallelism in your program. There are many aspects of this language, but we are mostly interested in just three keywords, spawn, sync and parallel_for.¹ Let us look at a code example.

¹In the language itself, they are cilk_spawn, cilk_sync, and cilk_for but we will just use spawn and sync in the pseudocode.
Fib(n)
1 if n ≤ 1
2 then return 1
3 x ← spawn Fib(n − 1)
4 y ← Fib(n − 2)
5 sync
6 return x + y

Adding the spawn keyword before the function means that the runtime is allowed to execute that function in parallel with the continuation, the remainder of the parent function. The sync keyword means that all functions that have been spawned so far have to return before control can go beyond this point. Therefore, in the above code, the values of Fib(n − 1) and Fib(n − 2) can be calculated in “parallel.”

Another important concept is the parallel for loop. This construct gives the runtime system permission to execute all the iterations of the for-loop in parallel with each other. For example, if you have an array A[1..n] and you have to create an array B such that, for all i, B[i] = A[i] + 1, then you can simply write:

1 parallel_for i ← 1 to n
2 do B[i] ← A[i] + 1

In general, two things can be done in parallel if they don’t depend on each other. Let’s say I want to calculate the sum of all the elements of an array. Can I use the following program?

1 parallel_for i ← 1 to n
2 do s ← s + A[i]

These iterations are not independent since the variable s is changed in all the iterations. The program can lead to several different values of s depending on how the instructions are interleaved (scheduled) when run in parallel. These are called race conditions. A race condition occurs when more than one (logically) parallel operations access the same memory location and one of the accesses is a write.

In this course, we are concerned with deterministic parallel programs, that is, programs that give the same result regardless of the schedule. While there are many definitions of determinism, we will concentrate on programs that have no race conditions. You must be careful to make sure that when you use the spawn keyword, you are not introducing race conditions in your program.

Exercise 1 If you want to do the sum computation in parallel without determinacy races, how would you do it?
2 Let’s look at some examples

We can now look at some examples of sequential programs you have written in the past and see how we can parallelize them simply:

Matrix Multiplication Say we have two square \((n \times n)\) matrices \(A\) and \(B\). To compute their product and store it into another \(n \times n\) matrix \(C\), simply do:

\[
C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}
\]

which can be easily down by writing a triple-nested for loops. That means, it’s also easily parallelizable using parallel_for loops:

```plaintext
1 let C be a new n x n matrix
2 parallel_for i ← 1 to n
3 do parallel_for j ← 1 to n
4 do C_{ij} ← 0
5 for k ← 1 to n
6 do C_{ij} ← C_{ij} + A_{ik} \cdot B_{kj}
```

Merge Sort Recall that given an array of numbers, a sorting algorithm permutes this array so that they are arranged in an increasing order. You are already familiar with Merge Sort. Lets try to make it parallel. The most obvious thing to do to make the merge sort algorithm parallel is to make the recursive calls in parallel.

```plaintext
MergeSort(A, n)
1 if n = 1
2 then return A
3 Divide A into two \(A_{left}\) and \(A_{right}\) each of size \(n/2\)
4 \(A'_{left} ← \text{spawn} \text{ MERGE} \text{SORT}(A_{left}, n/2)\)
5 \(A'_{right} ← \text{MERGE} \text{SORT}(A_{right}, n/2)\)
6 sync
7 Merge the two halves into \(A'\)
8 return \(A'\)
```

So how can we tell if these are “good” parallel algorithms?
3 The RAM model for sequential computation:

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM) model. This model assumes a single processor executing one instruction after another, with no parallel operations, and that the processor has access to unbounded memory. The model contains instructions commonly found in real computer, including basic arithmetic and logical operations (e.g. +, -, *, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. Each instruction is assumed to take a constant amount of time, and the cost of a computation is measured in terms of the number of instructions execute by the machine.

This model has served well for analyzing the asymptotic runtime of sequential algorithms, and most work on sequential algorithms to date has used this model. One reason for its success is that there is an easy mapping from algorithmic pseudocode and sequential languages such as C and C++ to the model, and so it is reasonably easy to reason about the cost of algorithms. That said, this model should only be used to derive the asymptotic bounds (i.e., using big-O, big-Theta and big-Omega), and not for trying to predict exact running time. One reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

4 Performance measures

For a parallel algorithm, we are concerned with the amount of parallelism in the algorithm, or in other words, how it scales with the number of processors. We will measure the parallelism using two metrics: work and span. Roughly speaking, the work corresponds to the total number of instructions we perform, and span (also called depth or critical path length) to the number of instructions along the longest chain of dependences. Another way of thinking about it is: work is the running time of the program on one core, while span is the running time of the program on an infinite number of cores.\footnote{Here we assume that the scheduler is perfect and has no overheads.} Henceforth, we will use $T_p$ to denote the time it takes to execute a computation on $p$ processors, so we will use $T_1$ to denote work and $T_\infty$ to denote span.

Just as an example, let’s analyze the work and span for the following code (note that any function can be represented in this manner):
The work of a program is the running time on one core (as measured with the RAM model), so we simply ignore all the spawns and syncs in the program:

\[ T_1(\text{FOO}) = T_1(e_1) + T_1(\text{BAR}) + T_1(\text{BAZ}) + T_1(e_2) + T_1(e_3) . \]

The span calculation is more involved:

\[ T_\infty(\text{FOO}) = T_\infty(e_1) + \max \{ T_\infty(\text{BAR}), T_\infty(\text{BAZ}), T_\infty(e_2) \} + T_\infty(e_3) . \]

**Exercise 2** What’s the work for \( \text{FIB}(n) \)?

*Hint: The closed form for Fibonacci number \( \text{Fib}_i = \frac{\phi^i - \tilde{\phi}^i}{\sqrt{5}} \), where \( \phi = \frac{1 + \sqrt{5}}{2} \) and \( \tilde{\phi} = \frac{1 - \sqrt{5}}{2} \) (i.e., the two roots of the equation \( x^2 = x + 1 \)).

**Exercise 3** What’s the span for \( \text{FIB}(n) \)?

Once we know the work and span of an algorithm, its **parallelism** is simply defined as the work over span:

\[ \mathbb{P} = \frac{T_1}{T_\infty} \]

One way of thinking about parallelism is that it denotes the average amount of work that can be performed in parallel for each step along the span. This measure of parallelism also represents roughly how many processors we can use efficiently when we run the algorithm.

The parallelism of an algorithm is dictated by both the work term and the span term. For example, suppose \( n = 10,000 \) and if \( T_1(n) = \theta(n^3) \approx 10^{12} \) and \( T_\infty(n) = \theta(n \log n) \approx 10^5 \) then \( \mathbb{P}(n) \approx 10^7 \), which is a lot of parallelism. But, if \( T_1(n) = \theta(n^2) \approx 10^8 \) then \( \mathbb{P}(n) \approx 10^3 \), which is much less parallelism. The decrease in parallelism is not because of the span was large, but because the work was reduced.
**Goals:** In parallel algorithm design, we would like to keep the parallelism as high as possible but without increasing the work. In general, the goals in designing efficient algorithms are

1. First priority: to keep work as low as possible
2. Second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this course we will mostly cover parallel algorithms that is **work efficient**, meaning that its work is the same or close to the same (within a constant factor) as that of the best sequential algorithm. Now among the algorithms that are work-efficient, we will try to minimize the span to achieve the greatest parallelism.

**Exercise 4** We said that `cilk_for` is just syntactic sugar for `cilk_spawn` and `cilk_sync`. How would you go about converting a for loop to spawns and syncs? Think about minimizing span while doing so.

**Exercise 5** What is the work and span of the following code:

```
1   parallel_for i ← 1 to n
2       do B[i] ← A[i] + 1
```