Outline of this lecture:

1. Overview of Cilk
2. The dag computation model
3. Performance measures
4. A simple greedy scheduler

1 Overview of Cilk

A brief history of Cilk technology

In this class, we will overview various parallel programming technology developed. One of the technology we will examine closely is Cilk, a C/C++ based concurrency platform. Before we overview the development and implementation of Cilk, we shall first overview a brief history of Cilk technology to account for where the major concepts originate. Cilk technology has developed and evolved over more than 15 years since its origin at MIT. The text under this subheading is partially abstracted from the “Cilk” entry in Encyclopedia of Distributed Computing [14] with the author’s consent. I invite interested readers to go through the original entry for a more complete review of the history.

Cilk (pronounced “silk”) is a linguistic and runtime technology for algorithmic multithreaded programming originally developed at MIT. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk’s runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details like load balancing, synchronization, and communication protocols. Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. Important milestones in Cilk technology include the original Cilk-1 [2, 1, 11], Cilk-5 [7, 18, 5, 17], and the commercial Cilk++ [15] by Cilk Arts.

The first implementation of Cilk, Cilk-1, arose from three separate projects at MIT in 1993. The first project was theoretical work [3, 4] on scheduling multithreaded applications. The second was StarTech [10, 12, 13], a parallel chess program built to run on the Thinking Machines Corporation’s Connection Machine Model CM-5 Supercomputer [16]. The third project was PCM/Threaded-C [8], a C-based package for scheduling continuation-passing-style threads on the CM-5. In April 1994 the three projects were combined and christened Cilk. Cilk-1 is a general-purpose runtime system that incorporated a provably efficient work-stealing scheduler. While it provided a provably efficient runtime support, it offered little linguistic support.

1Called “Cilk” in [2, 1, 11], but renamed “Cilk-1” in [7] and other MIT documentation.
Cilk-5 introduced Cilk’s linguistic model, which provided simple linguistic extensions such as `spawn` and `sync` for multithreading to ANSI C. The extension is *faithful*, which means that parallel code retains its serial semantics when run on one processor. Furthermore, the program would be an ordinary C program if the keywords for parallel controls were elided, referred to as the *serial elision*. Cilk-5 was first released in March 1997 [7], which included a provably efficient runtime scheduler like its predecessor, and a source-to-source compiler, compiling Cilk code to processed C code with calls to the runtime library.

In September 2006, responding to the multicore trend, MIT spun out the Cilk technology to Cilk Arts, Inc., a venture-funded start-up founded by technical leaders Charles E. Leiserson and Matteo Frigo, together with Stephen Lewin-Berlin and Duncan C. McCallum. Although Cilk Arts licensed the historical Cilk codebase from MIT, it developed an entirely new codebase for a C++ product aptly named Cilk++ [15], which was released in December 2008 for the Windows Visual Studio and Linux/gcc compilers.

Cilk++ improved upon the MIT Cilk-5 in several ways. The linguistic distinction between Cilk functions and C/C++ functions was lessened, allowing C++ “call-backs” to Cilk code, as long as the C++ code was compiled with the Cilk++ compiler.² The `spawn` and `sync` keywords were renamed `cilk_spawn` and `cilk_sync` to avoid naming conflicts. Loops were parallelized by simply replacing the `for` keyword with the `cilk_for` keyword, which allows all iterations of the loop to operate in parallel. Cilk++ provided full support for C++ exceptions. It also introduced reducer hyperobjects [6]. A Cilk++ program, like a Cilk program, retains its serial semantics when run on one processor. Moreover, one may obtain the serialization of a Cilk++ program, which is the same concept as serial elision, by eliding `cilk_spawn` and `cilk_sync` and replacing `cilk_for` with `for`.

Cilk Arts was sold to Intel Corporation in July 2009, which continued developing the technology. In September 2010, Intel released its ICC compiler with Intel Cilk Plus [9]. The product included Cilk support for C and C++, and the runtime system provided transparent integration with legacy binary executables.

In this class, we will mainly focus on technology developed since Cilk-5, including some implementation design found in Cilk-5, Cilk++, and Cilk Plus.

### Nested Parallelism in Cilk

Cilk supports three main keywords for parallel control: `cilk_spawn`, `cilk_sync`, and `cilk_for`. Parallelism is created using the keyword `cilk_spawn`. When a function invocation is preceded by the keyword `cilk_spawn`, the function is *spawned* and the scheduler may continue to execute the continuation of the caller in parallel with the spawned subroutine without waiting for it to return. The complement of `cilk_spawn` is the keyword `cilk_sync`, which acts as a local barrier and joins together the parallelism forked by `cilk_spawn`. The Cilk runtime system ensures that statements after a `cilk_sync` are not executed until all functions spawned before the `cilk_sync` statement have completed and returned.

---

²This distinction was later removed altogether by Intel Cilk Plus, though at the expense of sacrificing the performance and space guarantees provided by a working-stealing scheduler. We will explore this issue in more depth later in class.
The keyword \texttt{cilk\_for} is the parallel counterpart of the looping construct \texttt{for} in C and C++ that permits loop iterations to run in parallel. The Cilk Plus compiler converts the \texttt{cilk\_for} into an efficient divide-and-conquer recursive traversal over the iteration space. From the runtime system’s perspective, the \texttt{cilk\_for} construct can be desugared into code containing \texttt{cilk\_spawn} and \texttt{cilk\_sync}. Certain restrictions apply to the loop initializer, condition, and increment, for which I omit the details here and refer interested readers to [9].

Note that these keywords denote the \textit{logical parallelism} of the computation. They grant permission for parallel execution, but they do not command parallel execution. That is, not ever spawn realizes to parallel execution nor does it create a new thread. The underlying runtime system load-balances computations according to the processor resource available during execution, in a way that respect the dependencies denoted by these keywords. The underlying scheduler also provide provable guarantees for space and completion time bound, which we will see in more details later in the class.

2 The dag model for parallel 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. +, -, *, \textsc{and}, or, \textsc{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 (\textit{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.

In this lecture, we will look at how one analyze a parallel program written using Cilk Plus. 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 analyzing serial algorithms, 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.

To analyze a parallel algorithm, it’s helpful to think of the execution of the program as a \textit{directed acyclic graph (dag)}, where nodes represent instructions, and edges represent dependencies...
between instructions. If there is an edge from node \( u \) to node \( v \), then node \( u \) must be executed before node \( v \). A serial computation is just a chain of instructions. A parallel computation is a dag. If there is no path from node \( u \) to node \( v \) and vice versa, then there is no prescribed ordering between \( u \) and \( v \), and they can be executed in any order, or in parallel with each other.

For convenience, we won’t represent each instruction as a node. Instead, we will group together a sequence of instructions that does not contain any parallel keywords (i.e., spawn, sync, or returning from a spawned function), referred to as a strand, into a single node, and the dag models dependencies between strands.

To make this model concrete, let’s take a look at how we model the dag for a parallel computation. Recall the Fibonacci code from last lecture:

\[
\text{Fib}(n) \\
1 \quad \text{if } n \leq 1 \\
2 \quad \text{then return } 1 \\
3 \quad x \leftarrow \text{spawn Fib}(n - 1) \\
4 \quad y \leftarrow \text{Fib}(n - 2) \\
5 \quad \text{sync} \\
6 \quad \text{return } x + y
\]

The DAG representation of \( \text{Fib}(4) \) is shown below.

In this model, just like the RAM model, we will assume a basic set of instructions, and each instruction takes a constant amount of time, including access to memory. Unlike the RAM model, however, notice this Dag model is not really tied to the underlying machine but rather tied to the control structure of the program. As it turns out, there is a way to map the costs we derive using this model onto costs for running on parallel machines (machines with multiple processing units).

3 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.

In terms of the dag representation of a parallel computation, let’s assume that we chop up the strands so that each node consists of only one instruction (i.e., each node has cost one). Then, the work is the total number of nodes in the graph, and the span is the number of nodes along the longest path in the dag.\(^3\)

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.\(^4\) 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):

```
Foo
1  e_1
2  spawn BAR
3  spawn BAZ
4  e_2
5  sync
6  e_3
```

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:

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

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.

\(^3\)If each node may cost different amount, then the span is the sum of the cost along the path in the dag that has the largest such sum.

\(^4\)Here we assume that the scheduler is perfect and has no overheads.
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 \( T_{\infty}(n) = \theta(n \log n) \approx 10^8 \) then \( P(n) \approx 10^7 \), which is a lot of parallelism. But, if \( T_1(n) = \theta(n^2) \approx 10^8 \) then \( P(n) \approx 10^3 \), 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.

## 4 Greedy Scheduling

In order to get good performance, not only do we need a work-efficient algorithm with ample parallelism, we also need to ensure that the strands are mapped onto the executing processors efficiently. In the model that we just discussed, a program can generate lots of strands on the fly, which can lead to a huge amount of parallelism, typically much more than the number of processors available when running. The model itself does not say how strands are mapped onto the processors, however. It is the job of a scheduler to decide how strands are mapped onto executing processors (which and when).

Today, we are going to look at a simple scheduler, called greedy scheduler. We say that a scheduler is greedy if whenever there is a processor available and a strand ready to execute, then the strand will be scheduled on the processor and start running immediately.

Greedy schedulers have a very nice property that is summarized by the following:

**Theorem 1** On an ideal parallel computer with \( p \) processors, a greedy scheduler executes a multithreaded computation with work \( T_1 \) and span \( T_{\infty} \) in time

\[
T_p \leq \frac{T_1}{p} + T_{\infty}
\]

This is actually a very nice bound. For any scheduler, the time to execute the computation cannot be any better than \( \frac{T_1}{p} \), since we have a total of \( T_1 \) work to do, and the best we can possibly do is divide all the work evenly among the processors. Also note that the time to execute the computation cannot be any better than \( T_{\infty} \), since \( T_{\infty} \) represents the longest chain of sequential dependences. Therefore, the best we could do is:
We therefore see that a greedy scheduler does reasonably close to the best possible (within a factor of 2 of optimal). In particular, when \( \frac{W}{p} \gg S \) the difference between the two is very small. Indeed we can rewrite equation 1 above in terms of the parallelism \( \mathbb{P} = \frac{W}{S} \) as follows:

\[
T_p \leq \max \left( \frac{T_1}{p}, T_\infty \right)
\]

\[
T_p < \frac{W}{p} + S
= \frac{W}{p} + \frac{W}{\mathbb{P}}
= \frac{W}{p} \left( 1 + \frac{p}{\mathbb{P}} \right)
\]

Therefore as long as \( \mathbb{P} \gg p \) (the parallelism is much greater than the number of processors) then we get near perfect speedup (perfect speedup would be \( W/p \)).

**Proof of Theorem 1.** To prove this bound, let’s consider the execution dag \( G \) representing the computation with work \( T_1 \) and span \( T_\infty \). For convenience and without loss of generality, let’s assume each node in \( G \) represents one instruction (i.e., unit time). At each time step, either there are \( p \) or more nodes ready (call it a complete step), or there are less than \( p \) nodes ready (call it an incomplete step). We show this time bound by bounding the number of complete and incomplete steps a greedy scheduler takes before completing the execution.

Let’s first consider complete steps. In a complete step, a greedy scheduler executes \( p \) nodes (picked arbitrarily among all the ready nodes), using all \( p \) processors. The scheduler can’t take more than \( \lfloor \frac{T_1}{p} \rfloor \) number of complete steps, because otherwise the total work performed in those complete steps would be greater than \( T_1 \).

Now we try to bound the number of incomplete steps. We assert that after each incomplete step, the longest path length decreases by one (we will come back to show this assertion later). Since \( G \) starts out having the longest path length of \( T_\infty \), and the longest path length decreases by one at each incomplete step, there can be at most \( T_\infty \) number of incomplete steps.

What remains to be shown is the assertion that the longest path length decreases by one after each incomplete step. Let’s consider an complete step, and call the subdag that has yet to be executed at the beginning of this step as \( G' \). Say \( G' \) has longest path length \( l \). The path(s) (there can be multiple longest paths having the same length) with length \( l \) in \( G' \) must start with some node with in-degree 0 (i.e., a ready node), because otherwise \( G' \) would have the longest path length \( l + 1 \). Since it’s an incomplete step, the greedy scheduler executes all ready nodes with in-degree 0, thereby decreasing the longest path length by one. 

\[\square\]
In the real world: No real schedulers are fully greedy. This is because there is overhead in scheduling the job. Therefore there will surely be some delay from when a job becomes ready until when it starts up. In practice, therefore, the efficiency of a scheduler is quite important to achieving good running time. Also the bounds we give do not account for memory affects. By moving a job we might have to move data along with it. Because of these affects the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.

References


