6.895 Theory of Parallel Systems	Lecture 1

Introduction & Dynamic Multithreading

Lecturer: Charles Leiserson *Scribe: Abhi Shelat and Ben Adida*

Lecture Summary

1. *Course Overview*
   We present an overview of the course topics.
2. *Introduction to Parallel Programs*
   We define basic terminology and present our first parallel program.
3. *Modeling and Measuring Parallel Programs*
   We define the graph model for parallel programming and introduce a methodology for measuring the performance of parallel progams.
4. *Greedy Scheduler*
   We prove that a simple greedy scheduler executes in no worse than twice the theoretically optimal schedule.
5. *An Anecode*
   We review the results of the ⋆Socrates to show how empirical performance analysis does not suffice in understanding parallel computation.

1 Course Overview

This course **6.895** will focus on theoretical underpinnings of parallel computation, though it is clear that theory doesn't explain everything. This year, the course will address topics in a top-down fashion so as to help students pick final projects early on:

1. High-level models
   (a) multithreading
   (b) scheduling and load-balancing
   (c) data-race detection
   (d) synchronization (locking and lock-free algorithms)
2. Network architecture
   (a) memory consistency
   (b) routing networks (cf. RAW project, configurable networks)
3. VLSI theory
   (a) graph layouts
   (b) area/volume
   (c) AT^2 complexity
   (d) arithmetic circuits
   (e) universality (routing networks and VLSI's version of the Universal Turing Machine)

2 Introduction to Parallel Programs

Consider the following program to compute the nth Fibonacci number. This program is particularly bad: it runs in exponential time where a straight-forward iterative method yields linear time and a clever approach based on repeated squarings of a particular matrix yields $O(\log(n))$ time [see Cormen, Leiserson, Rivest page 902]. Nonetheless, this program is a good example for introducing our model of parallel computation.

```
fib(n)
    if n < 2
    then return n

    x <- spawn fib(n-1)
    y <- spawn fib(n-2)

    sync
    return (x+y)
```

A `spawn` instruction allows the next instruction of the parent program to be executed at the same time as the child, which in Unix programming, is equivalent to a `fork`. A `sync` instruction requires that all locally spawned children finish their work before proceeding with the computation. In Unix programming, a `sync` is equivalent to a `join`.

Our syntax for the presented program describes ***logical parallelism***. We do not explicitly state which tasks are assigned to which processors. Instead, we focus on the computations which can be safely performed in parallel. It is the ***scheduler***'s job to determine how to map dynamically unfolding execution onto a set of processors.

3 Modeling and Measuring Parallel Programs

3.1 Instructions Streams as a DAG

We model the instruction stream of a program as a **computation directed acyclic graph (DAG)**. Figure 1 shows the ***computation DAG*** for `fib(4)`. Vertices are ***threads*** which is a maximal sequence of instructions not containing parallel control (`spawn`, `return` from a spawned procedure, or `sync`).

We have three threads:

- Thread A: The `if` statement up to the first `spawn`.
- Thread B: The computation of `n-2` before the second `spawn`.
- Thread C: The `x+y` operation before the `return`.

3.2 Performance Measures

We define several performance measures to evaluate various scheduler algorithms. We assume an ideal situation: no cache issues, no interprocessor communication costs. These results may not hold under different assumptions, but we are after the essence of understanding.

- T_P is the minimum running time on P processors
- T_1, also called ***work***, is the minimum running time on 1 processor
- T_∞ is the minimum running time on an infinite number of processors. This is equivalent to the longest path in the DAG and is referred to as the ***critical path length***.

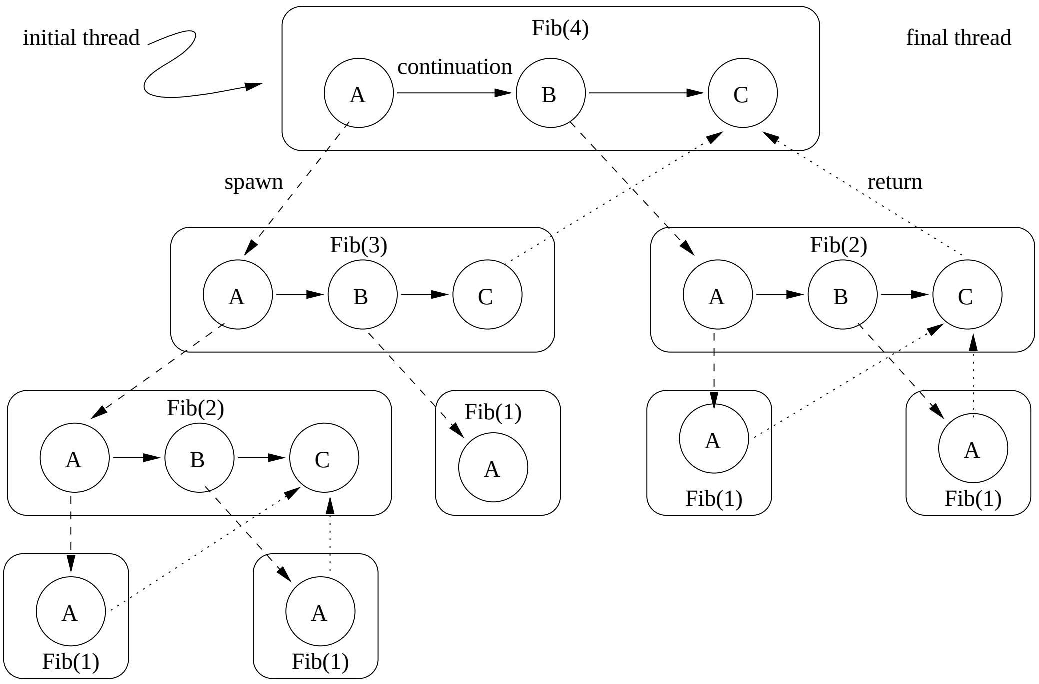

Figure 1: The instruction stream for `fib(4)`. Each circle represents a thread. Each solid arrow represents a `continuation`, each dashed arrow represents a `spawn` operation, and each dotted arrow represents a `return` from a spawned procedure.

In our example, if all threads run in unit time, we have $T_1 = 17$. In practice, the threads would be weighted by their individual runtimes. The ***critical path length*** is $T_\infty = 8$.

These measures immediately define lower bounds:

$$T_P \geq T_1/P \tag{1}$$

Proof Idea Assume by contradiction that $T_P < T_1/P$. In the best case, P processors can do at most P work in 1 step. By multiplying both sides by P, then $PT_P < T_1$, which implies that the total number of operations performed (left-hand side) is less than T_1, a contradiction. □

$$T_P \geq T_\infty \tag{2}$$

Proof Idea Once again, suppose not. If $T_P < T_\infty$, simply using P of the unlimited number of processors available reduces T_∞, which is a contradiction. □

$$T_1/T_P = \textbf{speedup} \tag{3}$$

Proof Idea With P processors, the maximum speedup is P. (Remember: the theory only applies to this simplified model.) □

A program's parallel execution plan can have:

- linear speedup: $T_1/T_P = \Theta(P)$
- superlinear speedup: $T_1/T_P = \omega(P)$ (not possible in this model, though it is possible in others).
- sublinear speedup: $T_1/T_P = o(p)$

The maximum possible speedup for a program given particular T_1 and T_∞ is T_1/T_∞, also called ***parallelism***. The ***parallelism*** is the average amount of work that can be done in parallel for every step along the critical path.

4 The Greedy Scheduler

A scheduler's job is to map a computation to particular processors. In reality, a scheduler typically must make decisions about what tasks or threads to perform on particular processors without the benefit of foresight, i.e, it does not know what it must schedule in the future. In general, these online schedulers are quite complex and will be discussed later in the term. Therefore, to begin our analysis, we focus on offline scheduling. Specifically, we consider the ***G***reedy Scheduler, a scheduler which attempts to do as much work as possible at every step.

In any schedule, there are two types of steps:

- **complete step**: There are at least P threads that are ready to run. The greedy scheduler selects any P of them and runs them.
- **incomplete step**: There are strictly less than P threads that are ready to run. The greedy scheduler runs them all.

Our first important theorem of the course is a beautiful argument, which extolls the virtue of greedy scheduling.

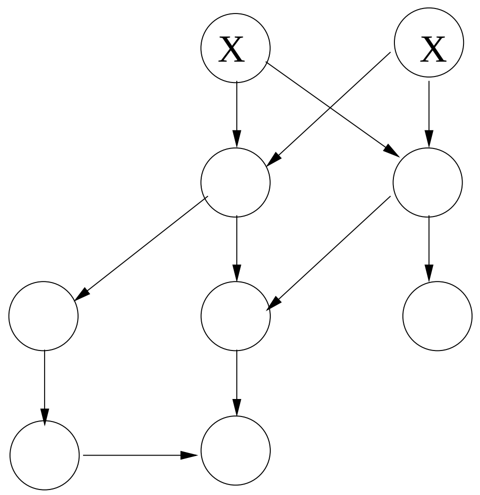

Figure 2: The leftover graph G' to execute during an incomplete step. Each nodes represents threads. The vertices represent continuations from one thread to another (due to parallel control such as `spawn` or `continue`). The nodes marked with an 'X' are threads already executed.

Theorem 1 *(Graham, Brent) Given P processors, a greedy scheduler executes any computation G with work T_1 and critical path length T_∞ in time:*

$$T_p \quad \leq \quad T_1/p + T_\infty \tag{4}$$

Proof In any schedule, there are only two types of steps: complete, and incomplete.

We first consider the complete steps. There can be no more than T_1/p complete steps; otherwise, work $> T_1$, which contradicts the definition of T_1.

We now consider the incomplete steps. Let G' be the subgraph of G that remains to be executed immediately prior to a given incomplete step, as shown in Figure 2. We note that all threads with in-degree 0 in G' are ready to be executed. Since we are dealing solely with incomplete steps, we also know that all of these threads can be immediately executed. The critical path of G' starts with a thread of in-degree 0. Thus, every incomplete step reduces the critical-path length by one unit. In other words, there can be no more than T_∞ incomplete steps.

Thus, the total time for executing incomplete and complete steps in a greedy scheduler is at most $T_1/P + T_\infty$. □

Corollary 2 *A greedy scheduler is always within a factor of 2 of optimal*

Proof Given the lower bounds 1 and 2, we can express a more concise lower bound as:

$$T_P \geq \max(T_1/P, T_\infty) \tag{5}$$

In addition, we can trivially express:

$$T_1/P \leq \max(T_1/P, T_\infty) \tag{6}$$

$$T_\infty \leq \max(T_1/P, T_\infty) \tag{7}$$

Given Theorem 1, we conclude

$$\begin{aligned} T_P &\leq T_1/P + T_\infty && (8)\\ &\leq \max(T_1/P, T_\infty) + \max(T_1/P, T_\infty) && (9)\\ &\leq 2\max(T_1/P, T_\infty) && (10) \end{aligned}$$

which shows that T_P is within twice lower bound 5. □

Every step either contributes to progress on the work (complete step), or progress on the critical path (incomplete step). A greedy scheduler is generally a good scheduler.

Corollary 3 *The greedy scheduler achieves linear speedup when* $P = O(T_1/T_\infty)$

Proof

$$\begin{aligned} T_p &\leq T_1/p + T_\infty && (11)\\ &= T_1/p + O(T_1/p) && (12)\\ &= \Theta(T_1/p) && (13) \end{aligned}$$

□

The idea is to operate in the range where T_1/P dominates T_∞. Given a particular problem with fixed T_1 and T_∞, adding more processors increases P and reduces T_1/P while T_∞ remains unaffected. Once T_∞ becomes significant in comparison to T_1/P, the critical path's dependencies reduce the ability to parallelize the computation to additional processors. As long as T_1/P dominates T_∞, all processors can be used efficiently.

5 Anecdote

$\star$Socrates, a chess program, had two versions, one including an optimization that significantly improved performance on 32 processors. The performance improvement reduced the total work, but increased the critical-path length:

- $T_{32} = 65sec$, $T_1 = 2048$, $T_\infty = 1$
- $T'_{32} = 40sec$, $T'_1 = 1024$, $T_\infty = 8$

On 512 processors, however, the optimization becomes a problem:

- $T_{512} = T_1/512 + T_\infty = 5$
- $T'_{512} = T'_1/512 + T'_\infty = 10$

The moral of the story: work and critical-path length are more important to understand thean simple empirical performance measurements.