Lecture 1 Notes (Sep 8, 2025)

Sam Westrick

s.westrick@nyu.edu

Parallel Hardware

Almost every computing device today is parallel. My iPhone has 12 cores (Apple A18 Pro: 2 perf CPU cores, 4 efficiency cores, 6 GPU cores). You can buy a state-of-the-art desktop multicore processor for just a few hundred dollars, e.g., https://www.newegg.com/amd-ryzen-9-7900x-ryzen-9-7000-series-raphael-zen-4-socket-am5/p/N82E16819113769. For ~$15K, you can get a single chip with 192 cores (384 threads), such as the AMD EPYC 9965: https://www.techpowerup.com/cpu-specs/epyc-9965.c3904.

These devices offer the potential of serious speedups, but can be difficult to program.

Common Parallel+Concurrent Software Bugs

Parallel and concurrent software is especially prone to bugs from **race conditions** and **data races** which can be difficult to identify, let alone debug. The consequences can be severe, for example, the infamous Northeast Blackout of 2003 was due to an unintended race condition.

It is easy to write a program that exhibits race conditions in practice. For example, see `Ex.data_race_ex` in the accompanying lecture code. This example simply attempts to increment a shared counter 10 million times, in parallel.

We can run this example as follows. Note that on a single core, we always get the same result for the final value of the counter (10000000). However, when using multiple cores, the results are unpredictable.

```
$ make main

# Running with a single core always returns the same result:
$ ./main @mpl procs 1 -- data_race -n 10000000
10000000
$ ./main @mpl procs 1 -- data_race -n 10000000
10000000

# Running with multiple cores results in unpredictable results:
$ ./main @mpl procs 8 -- data_race -n 10000000
2879180
$ ./main @mpl procs 8 -- data_race -n 10000000
2806651
$ ./main @mpl procs 8 -- data_race -n 10000000
2805723
```

The problem is that each update to the mutable cell is non-atomic. In the example, we write `counter := !counter + 1`. This is equivalent to:

```
let
  val old_x = !counter
  val new_x = old_x + 1
in
  counter := new_x
end
```

Here it is clear that the update does not occur as a single instruction, but rather multiple separate instructions. If two processors execute this code simultaneously (assuming an initial counter value of 0), it is possible that both processors read same the value `old_x = 0`, and then both write the value `counter := 1`. We have **lost an update** due to an unfortunate timing of events.

To prevent this problem, in this course, we will be **avoiding mutable data**. Instead, most of the code we will write is "purely functional". This term is generally used to refer to code which computes entirely using immutable data.

The purely functional approach has numerous benefits, as we will develop throughout this course. Mostly importantly, it allows us to easily and quickly write parallel code which is efficient, scalable, and (most importantly) **correct**, often with a simple proof of correctness.

Parallelism vs Concurrency

These terms are often used interchangeably, but it is useful to distinguish them.

> Define **parallelism** as the case where the *performance* of a piece of software is affected by using multiple processors simultaneously (usually, for speedups).
>
> Define **concurrency** as the case where the *correctness* or *behavior* of the program is affected by the timing of events.

With these definitions, it is clear that **parallelism** and **concurrency** are orthogonal concerns.

It may be helpful to identify examples for each combination:

	not concurrent	concurrent
not parallel (sequential)	classic algorithms 101 (e.g. sequential mergesort, binary search, etc.)	single-processor web server (dynamically responding to queries that arrive unpredictably)
parallel	DETERMINISTIC PARALLELISM (main topic of this course!)	multi-processor web server (dynamically responding to many unpredictable queries, simultaneously, in parallel)

An Example of Parallel-But-Not-Concurrent

A class "hello world" of parallel programming are the Fibonacci numbers:

```
fun fib(n) = if n <= 1 then n else fib(n-1) + fib(n-2)
```

These can be easily computed in parallel. (See `fib` in `Ex.sml` in the accompanying code.)

```
fun fib n =
  if n <= 1 then
    n
  else
    let val (a, b) =
      ForkJoin.par (fn () => fib (n - 1),
                    fn () => fib (n - 2))
    in a + b
    end
```

The function `ForkJoin.par` is the main ingredient: this function takes two (first-class) functions as argument, executes both, and returns the two results as a tuple.

No matter how many processors we use, this code always returns the same result. There is parallelism here (it gets faster with more processors), but no concurrency.

```
# compute the 40th fibonacci number (102334155)
$ ./main @mpl procs 1 -- fib -n 40
n 40
warmup 0.0000
repeat 1
time 1.9978s

average 1.9978s
minimum 1.9978s
maximum 1.9978s
std dev 0.0000s
102334155

# on two processors: same result, but approximately twice as fast
$ ./main @mpl procs 2 -- fib -n 40
n 40
warmup 0.0000
repeat 1
time 1.0271s

average 1.0271s
minimum 1.0271s
maximum 1.0271s
std dev 0.0000s
102334155
```

Computing a Sum in Parallel

See the code for `parallel_sum` in `Ex.sml` in the accompanying code. The function `parallel_sum(lo, hi, f)` computes the summation `f(lo) + f(lo+1) + ... + f(hi-1)` in parallel. How can we do this, using only `ForkJoin.par` for parallelism?

A simple and effective solution is to process the range by splitting the range in half and recursively processing the two halves in parallel. The base cases handle ranges of indices that are either empty or a singleton.

```
fun parallel_sum(lo, hi, f) =
  if lo >= hi then
    0
  else if lo + 1 = hi then
    f(lo)
  else
    let
      val mid = lo + (hi - lo) div 2
      val (left, right) =
        ForkJoin.par (fn () => parallel_sum(lo, mid, f),
                      fn () => parallel_sum(mid, hi, f))
    in
```

```
    left + right
  end
```

Programming Exercise: Parallel Word Count

Challenge: use `parallel_sum` to compute the number of words in a file in parallel. The input is a sequence of characters, and the output should be the number of words. For example, on the input string `__hello___world_` (where `_` indicates a space), the output should be 2.

```
fun word_count(chars: char Seq.t) : int =
  (* TODO... *)
```

As an exercise, try to solve this problem. A hint is that you can solve it using a single call to `parallel_sum`. Some useful functions are:

```
Seq.length(s)      (* length of a sequence s *)
Seq.nth s i        (* the ith element of the sequence s *)
Char.isSpace(c)    (* returns true if the character c is a whitespace character *)
```

A solution is shown in the file `Ex.sml` in the accompanying code. See the function `wc` in this file.

Measuring Performance

Define $T(P)$ as the running time of a program on P processors. Next, define the (self-)speedup of a program on P processors as $selfspeedup(P) = T(1)/T(P)$. Note that we call this "self-speedup" because, as a baseline, it compares against the sequential performance of the **same program**.

Ideally, in a perfect world, $selfspeedup(P) \approx P$. However, in reality, this is almost never the case.

There are numerous reasons for this. Three important reasons in particular:

- **The memory wall**. On modern chips, the maximum throughput of memory loads does not match the cumulative computational power of all the cores together. That is, if all processors are loading from memory simultaneously, then (in the hardware) there will be significant delays, causing the peek performance to plateau.
- **Scheduling is not free**. The system has to perform work to make decisions about which processor should be assigned to execute each task. Tasks also have to be migrated between processors, which requires additional instructions to be executed. All of this is additional overhead. It is unavoidable for parallel computation, but sequential code has no such overhead.
- **Insufficient parallelism**. At the extreme, consider that an entirely sequential program obviously has a speedup plateau: $selfspeedup(P) \approx 1$ regardless of P. More generally, many problems are *difficult to parallelize* and may not be able to expose a sufficient number of parallel tasks to keep all processors happy. We will study this issue in particular in great detail throughout the course.

To measure speedups, we often use speedup plots. An example speedup plot is shown in Figure 1.

Scheduling

We have thrown around some intuitive terms (tasks, processors, assigning tasks to processors, etc.), but it is time to make this more precise.

A useful model is the **DAG model**, where we model the execution of a parallel program in terms of a **d**irected **a**cyclic **g**raph (DAG). We often refer to this graph as a **computation graph**. Each vertex in the graph represents a group of instructions that were executed sequentially, without any synchronization. Edges in the graph represent dependencies.

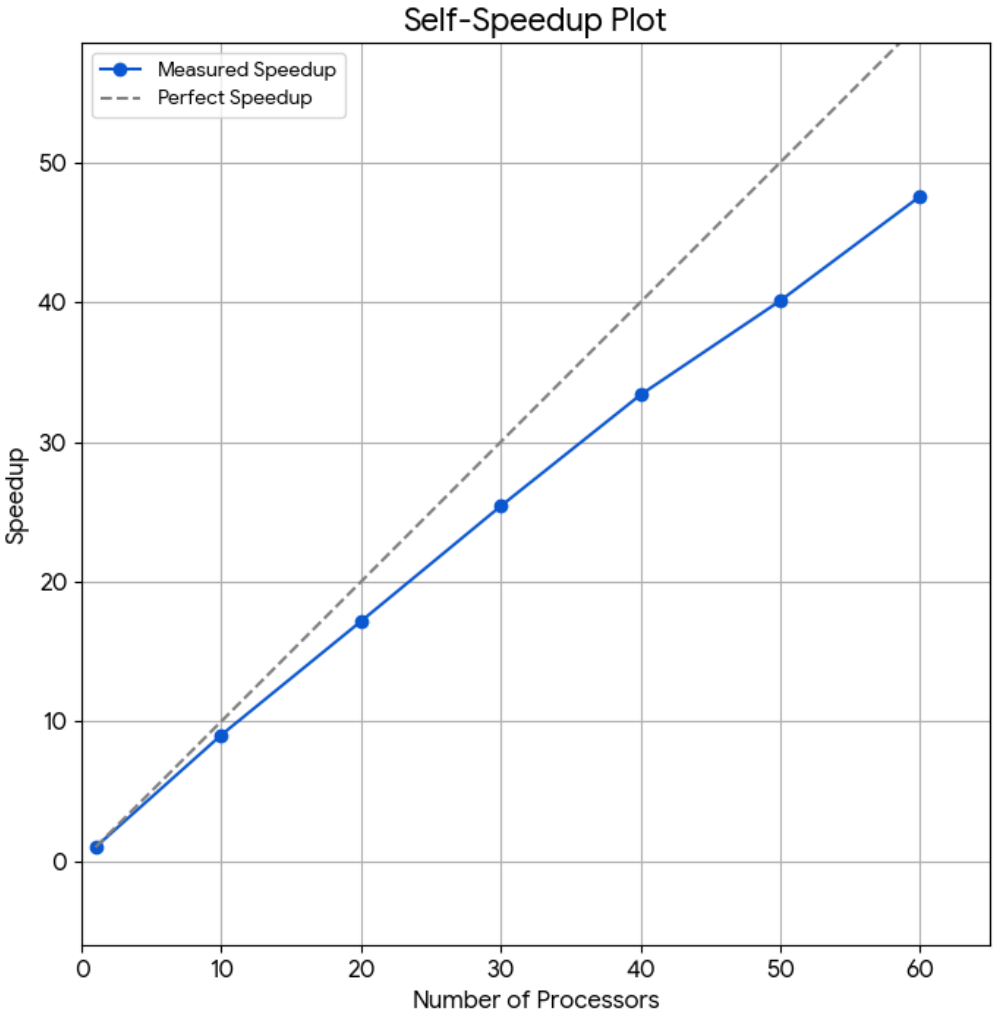
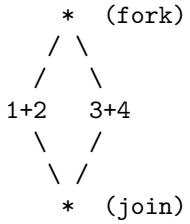


Figure 1: Example speedup plot

For example, execution of the code `ForkJoin.par(fn () => 1+2, fn () => 3+4)` can be modeled as the following DAG, with edges pointing from top to bottom. Here, I am labeling the two middle vertices by the expressions they compute, for clarity.



The vertex at the top is known as a **fork** vertex; this vertex shows that the computation splits (forks) into two computations. Every **fork** vertex has a corresponding **join** vertex. The join vertex does not execute until after both of the inner computations have finished, hence the two incoming edges.

A larger example is shown in Figure 2. In this example, we draw the computation graph corresponding to an execution of `Ex.fib(4)` from `Ex.sml` in the accompanying code from lecture.

With a computation graph in hand, we can now imagine **scheduling** the graph, specifically by assigning vertices to processors. This assignment has to respect the dependencies in the graph: for every edge `a -> b` in the graph, we cannot assign `b` to a processor until after `a` has been assigned to a processor on a previous time step.

A simple strategy which is extremely effective in practice is to assign vertices **greedily**. This is known as the **greedy scheduling principle**:

> **Greedy scheduling principle**. If a vertex is ready to be assigned, then no processor should be idle.

For the example computation graph we considered earlier, we can construct a **greedy schedule** by adhering to the greedy scheduling principle at every time step. Assuming each vertex takes a single unit of time to execute, a 2-processor greedy schedule is shown in Figure 3. There are 13 vertices in the graph, and this particular 2-processor greedy schedule finishes in 8 units of time. Abstractly, this would correspond to a 2-processor self-speedup of $13/8 = 1.625$.

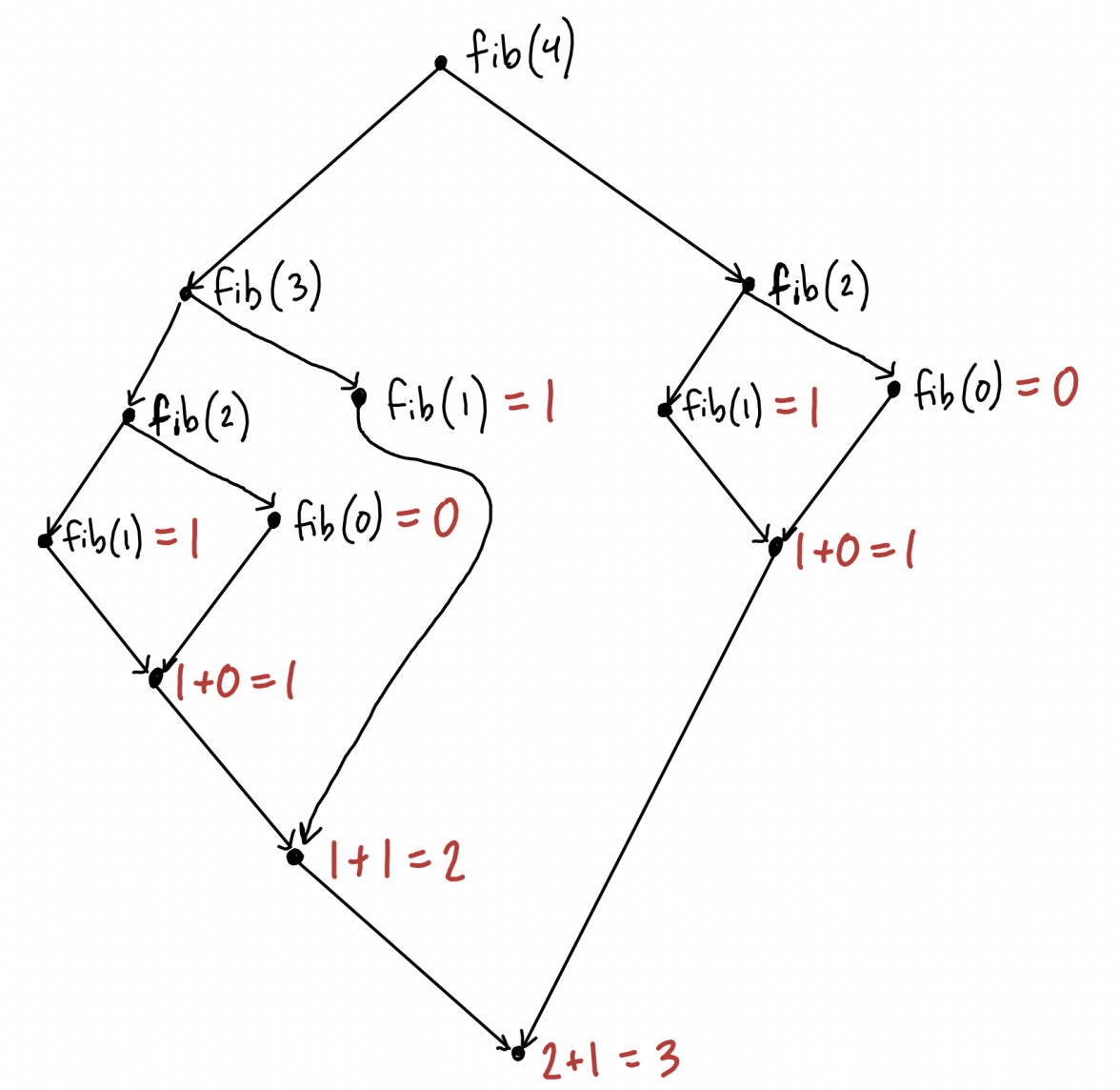


Figure 2: Computation graph for `fib(4)`. Vertices are labeled with the expressions they compute, for clarity.

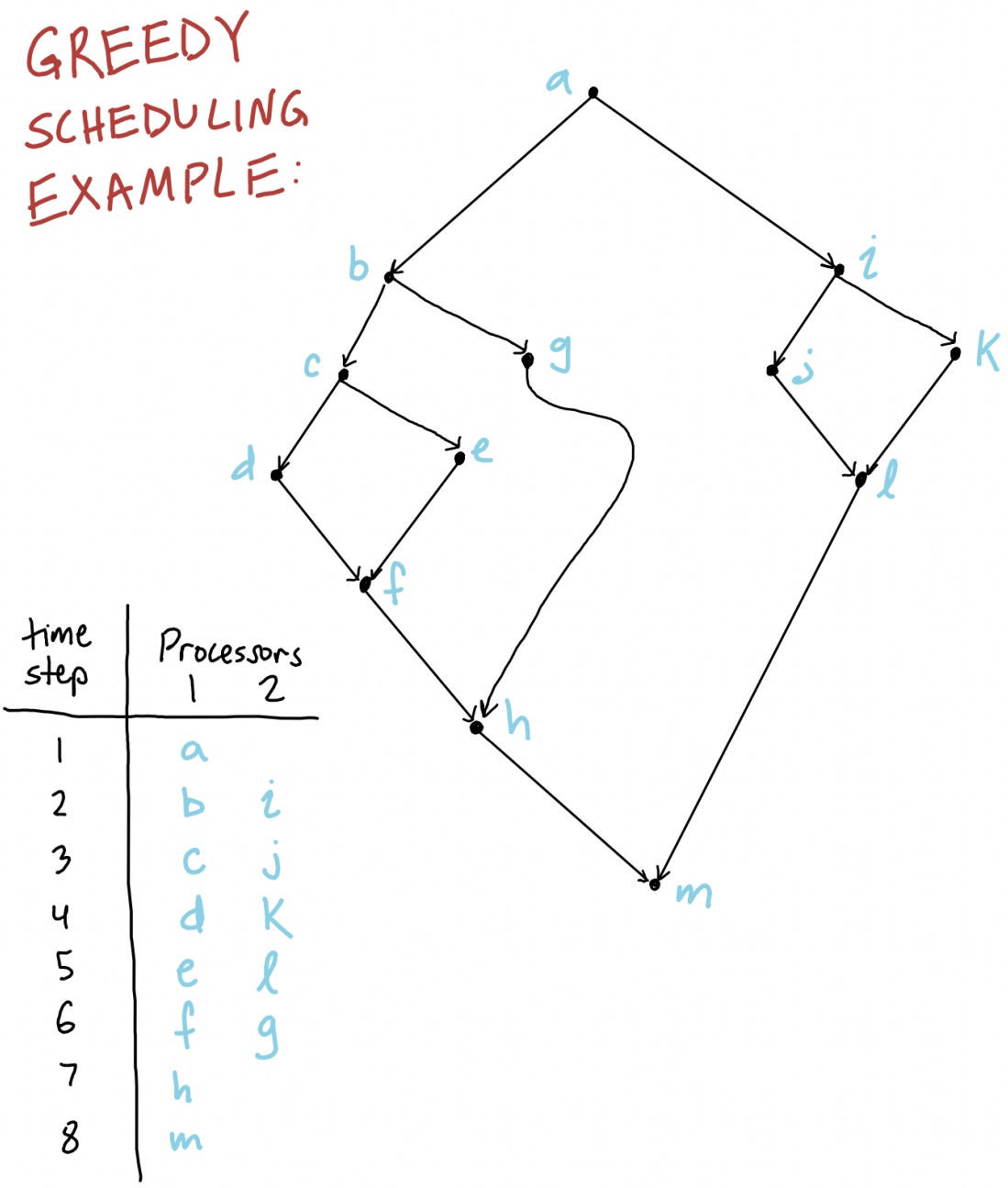


Figure 3: Greedy 2-processor schedule of an example computation graph. There are 13 vertices, which would have taken a single processor 13 time steps. With 2 processors, we complete the execution using only 8 time steps. This (abstractly) yields a 2-processor self-speedup of $13/8 = 1.625$.