Lecture 2 Notes (Sep 15, 2025)

Sam Westrick

s.westrick@nyu.edu

The Work-Span Cost Model: A Formal Notion of Parallelism

We previously defined the DAG model, which represents the history of a computation in terms of a directed acyclic graph. Vertices in this graph represent instructions that were executed, and (directed) edges indicate sequential dependencies. Scheduling a program, then, corresponds to traversing the graph under the constraint that, for every edge $u \to v$, the vertex v cannot be visited until after u.

We have also seen—both in theory, and in practice—that not all computations achieve perfect speedup. In theory, even the best possible schedule of a computation might not be able to achieve perfect speedup, due to insufficient parallelism. In practice, if we plot $\mathit{speedup}(P) = T(1)/T(P)$, we typically see a curve with a horizontal asymptote. This horizontal asymptote is an upper bound on the speedup on the computation. The question is: **where is this asymptote**? And, how can we predict where it will be?

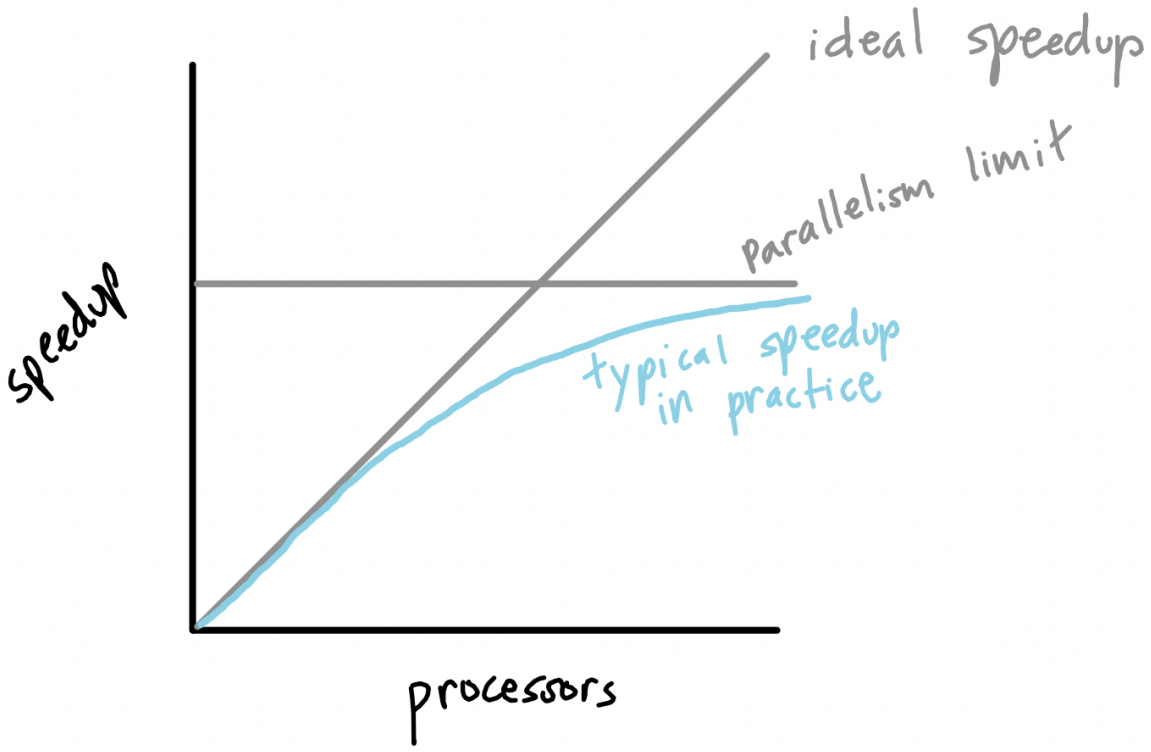

Figure 1: As the number of processors increases, the speedup approaches a limit.

To answer these questions, we can define two useful quantities.

Define **Work**: the number of vertices in the dag.

Define **Span**: the length of the longest path in the dag.

You can equivalently think about the work as the amount of time it takes to execute the computation on a single processor. The span is, in theory, the minimum possible execution time, assuming an unbounded number of processors. We often refer to the longest path in the dag as the **critical path**. Note that there can be multiple critical paths, i.e., multiple longest paths through the dag that all have the same length.

From these two quantities, we can immediately determine where a horizontal asymptote will be in our speedup plots. Let W be the work and S be the span. We have $T(1) = W$. We know $T(P) \geq S$, regardless of P, because no schedule can possibly do better than walk the critical path sequentially.

Therefore $\mathit{speedup}(P) = T(1)/T(P) \leq W/S$.

This quantity, W/S, is *defined* as the **parallelism** of a computation. Intuitively, in theory, it is the maximum possible speedup.

> Define **Parallelism** as the ratio W/S, where W is the work and S is the span.

Greedy Scheduling is Nearly Optimal

Recall the *greedy scheduling principle*:

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

Greedy schedulers minimize idleness at every moment. It turns out that greedy scheduling alone is enough to guarantee nearly optimal performance in practice.

The key theorem (with multiple interesting proofs by a variety of authors) is as follows.

> **Theorem [Brent] [Arora, Blumofe, Plaxton]**. For a computation with work W and span S, greedy scheduling on P processors guarantees $T(P) \leq W/P + S$.

This upper-bound on $T(P)$ is nearly optimal, and the proof is actually quite short. Let $\mathit{opt}(P)$ be the optimal (best possible) P-procesor schedule of a computation that has work W and span S. We know that $\mathit{opt}(P) \geq W/P$, because no schedule can possibly do better than perfectly dividing up the work amongst the available processors. We also know $\mathit{opt}(P) \geq S$, because no schedule can do better than executing the critical path sequentially. Therefore $\mathit{opt}(P) \geq \max(W/P, S)$. Combining this with the greedy scheduling theorem above, we have

$$\max(W/P, S) \leq \mathit{opt}(P) \leq T(P) \leq W/P + S$$

Finally, note that $W/P + S \leq 2\max(W/P, S)$, and therefore both $\mathit{opt}(P)$ and $T(P)$ lie within the range of $\max(W/P, S)$ and $2\max(W/P, S)$. Hence $T(P)$ is at most a factor of two larger than $\mathit{opt}(P)$.

This result is quite powerful: it shows us that we can efficiently execute a parallel program by greedily assigning tasks to processors as they become available, and we will be *guaranteed* that the overall performance is nearly optimal.

As a final note, it is helpful to plot the idealized curve $W/(W/P + S)$ for different values of P. This curve is a decent estimate of the speedup you could hope to achieve in practice using a greedy scheduler. As $P \to \infty$, indeed, we see that this curve approaches a limit at W/S, i.e., the parallelism. And, for small values of P, assuming $W \gg S$, the curve is mostly linear; this is because (for sufficiently small P) we have $W/P \gg S$ and therefore the speedup $W/(W/P + S)$ simplifies to approximately P.

A Language-Based Work-Span Cost Model

The above approach relies on constructing (in your head) a computation graph. Alternatively, we can define the work and span of a computation directly on the program itself.

Here, we define two functions $W(\texttt{e})$ and $S(\texttt{e})$ for the work and span of executing `e`, where `e` is an expression in our programming language.

For example,

$$W(\texttt{let val () = e1 in e2 end}) = W(\texttt{e1}) + W(\texttt{e2})$$

$$S(\texttt{let val () = e1 in e2 end}) = S(\texttt{e1}) + S(\texttt{e2})$$

This says that if we execute the expression `e1` and then execute the expression `e2`, then both the work and span overall are additive. This is known as **sequential composition**.

All expressions in the language are sequential by default. The only parallel form of expression is the function `par`, which executes two functions (logically) in parallel.

$$W(\texttt{par(f,g)}) = 1 + W(\texttt{f()}) + W(\texttt{g()})$$

$$S(\texttt{par(f,g)}) = 1 + \max(S(\texttt{f()}), S(\texttt{g()}))$$

Here, the work is additive, but the span has to consider which of the two branches (`f()` or `g()`) has a longer span. The overall span is the max of the two. The $+1$ is important for accounting for the cost of `par` itself.

To fully define the work and span this way, we still need to evaluate the expressions. For example, the work (and span) of a conditional expression would need to know the value of the resulting boolean:

$W(\texttt{if eb then e1 else e2}) = W(\texttt{eb}) + W(\texttt{e1})$, if $\texttt{eb} \rightsquigarrow \texttt{true}$.

$W(\texttt{if eb then e1 else e2}) = W(\texttt{eb}) + W(\texttt{e2})$, if $\texttt{eb} \rightsquigarrow \texttt{false}$.

In the above definitions, the syntax $\texttt{e} \rightsquigarrow \texttt{v}$ means that the expression `e`, when evaluated fully, results in a value `v`.

(For those familiar with techniques for defining the semantics of programming languages, this corresponds to the notion of a "big-step" operational semantics.)

Introduction to Work and Span Recurrences

Recall the code for a parallel sum from the previous lecture.

```
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
```

We can analyze the work and span of this code, under some reasonable assumptions.

First, assume $W(\texttt{f(i)}) = S(\texttt{f(i)}) = O(1)$, i.e., constant, regardless of the index `i`. Under this assumption, the asymptotic work and span of `parallel_sum(lo,hi,f)` depends only on the size of the range `n = hi-lo`. We can then define $W(n)$ and $S(n)$ as the work and span for a range of size n.

Following the definitions we have the following. These recurrences are derived directly from the code; each recursive call corresponds to a recursive instance of $W(-)$ or $S(-)$ on the right-hand side, with an appropriate size.

$W(n) = 2W(n/2) + O(1)$, if $n > 1$.

$S(n) = S(n/2) + O(1)$, if $n > 1$.

$W(n) = S(n) = O(1)$, if $n \leq 1$.

We can then solve these recurrences to see that $W(n) = O(n)$, i.e., the work of the code is linear, and $S(n) = O(\log n)$, i.e., the span is logarithmic. The code therefore has $O(n/\log n)$ parallelism, which asymptotically increases with the size of the range, n. In other words, the amount of parallelism (and therefore the maximum possible speedup) increases with the problem size.

Generalized Divide-and-Conquer: Parallel Reduction

The above code for `parallel_sum` can easily be generalized to compute a variety of things beyond just a summation. The general notion is called `reduce` and is known as a **parallel reduction**.

The code for a reduction looks as follows:

```
fun reduce g z (lo, hi) f =
  if lo >= hi then
    z       (* user-specified: "zero" element *)
  else if lo + 1 = hi then
    f(lo)   (* user-specified: one element of the reduction *)
  else
    let
      val mid = lo + (hi - lo) div 2
      val (left, right) =
        ForkJoin.par (fn () => reduce g z (lo, mid) f,
                      fn () => reduce g z (mid, hi) f)
    in
      (* user-specified: arbitrary combination of the two results *)
      g (left, right)
    end
```

A few immediate examples:

```
(* compute the product of a sequence of integers *)
val p = reduce (fn (a, b) => a*b) 1 (0, n) (fn i => ...)
(* compute the max of a sequence of integers *)
val m = reduce Int.max (valOf Int.minInt) (0, n) (fn i => ...)
(* concatenate many strings in parallel *)
val s = reduce (fn (a, b) => a^b) "" (0, n) (fn i => Int.toString(i))
```

In general, although any function `g` could be passed to combine the elements, it is helpful to only consider functions are are **associative**.

Function g is associative if $\forall a, b, c : g(g(a, b), c) = g(a, g(b, c))$

Associativity ensures that the application order does not matter. This means that the behavior of the reduction does not depend on where exactly the midpoint is, when (in the parallel case) we split the index range in half.

Similarly, it is helpful to think of the element `z` as being an **identity** element for `g`.

An element z is an identity of function g if $\forall x : g(x, z) = g(z, x) = x$.

This gives an algebraic interpretation to the meaning of a reduce. In particular, `reduce` can be used to compute a generalized sum of elements drawn from any monoid.

Strengthened Divide-and-Conquer

Divide-and-conquer is a general strategy that can be used to compute a surprising number of interesting things. Here we will explore a few examples.

First, as we have seen, we can easily compute the total sum of a sequence of numbers: split the input in half, recursively compute the sum of the halves, and the add up the results.

Suppose we wanted to compute the **maximum prefix sum** of a sequence of numbers. For example, given the sequence $\langle 1, -2, 0, 3, -1, 0, 2, -3\rangle$, the maximum prefix sum is 3, corresponding to sum of the prefix $\langle 1, -2, 0, 3, -1, 0, 2\rangle$ which excludes the last element.

We could attempt to split the sequence in half, recursively compute the maximum prefix sums of the halves, and then combine the results somehow. But how do we combine the results? The best overall prefix might cross the midpoint, and we have no way of combining the results of our two recursive calls to compute the overall best prefix. We are stuck.

```
fun max_prefix_sum(lo, hi, f) =
  if lo >= hi then
    0
  else if lo + 1 = hi then
    (* if the single element here is negative, then the best prefix sum
     * is 0, i.e., we should take the empty prefix.
     *)
    Int.max (f(lo), 0)
  else
    let
      val mid = lo + (hi - lo) div 2
      val (left_prefix, right_prefix) =
        ForkJoin.par (fn () => max_prefix_sum(lo, mid, f),
                      fn () => max_prefix_sum(mid, hi, f))
    in
      (* What do we do here?? What if the best prefix crosses the midpoint?? *)
      ...
    end
```

To combine the results, we need to know more than just the best prefix sum. The case where the best overall prefix sum crosses the midpoint can be handled by also keeping track of the *total sum of the left-hand side*.

```
fun max_prefix_sum_and_total(lo, hi, f) =
  if lo >= hi then
    (* best prefix sum is 0, total sum is 0 *)
    (0, 0)
  else if lo + 1 = hi then
    let
      val x = f(lo)
      val best_prefix_here = Int.max (x, 0)
      val total_sum_here = x
    in
```

```
      (best_prefix_here, total_sum_here)
    end
  else
    let
      val mid = lo + (hi - lo) div 2
      val ((p1, t1), (p2, t2)) =
        ForkJoin.par (fn () => max_prefix_sum_and_total(lo, mid, f),
                      fn () => max_prefix_sum_and_total(mid, hi, f))
    in
      (* Now we can handle the case that the prefix sum crosses the midpoint.
       * If it does, the best overall prefix sum would be t1+p2, i.e., the
       * total sum on the left plus the best prefix on the right.
       *)
      (Int.max (p1, t1+p2), t1+t2)
    end

(* Now we can compute the original problem by running the strengthened
 * divide-and-conquer and then throwing away the overall total sum
 *)
fun max_prefix_sum(lo, hi, f) =
  let val (p, _) = max_prefix_sum_and_total(lo, hi, f)
  in p
  end
```

This idea, of seemingly computing **more** than what you originally thought you needed to, is called **strengthening**. It is analogous to *strengthening the inductive hypothesis of an inductive proof.*

Note also that we can simplify the above code by just calling `reduce`:

```
fun max_prefix_sum(lo, hi, f) =
  let
    fun g ((p1, t1), (p2, t2)) = (Int.max (p1, t1+p2), t1+t2)
    val z = (0, 0)
    val (p, _) =
      reduce g z (lo, hi) (fn i =>
        let val x = f(i)
        in (Int.max (x, 0), x)
        end)
  in
    p
  end
```

A Larger Example: Maximum Contiguous Subsequence Sum

Using a similar idea as above, an interesting problem is to compute the *maximum contiguous subsequence sum* (MCSS), i.e., the largest sum of a contiguous slice of the input sequence. For example, on the input $\langle 1, -2, 0, 3, -1, 0, 2, -3\rangle$, the MCSS is 4; this corresponds to the contiguous subsequence $\langle 3, -1, 0, 2\rangle$ in the middle.

Our solution will use a 4-tuple:

```
( p   (* the best prefix sum *)
, t   (* the total sum *)
, s   (* the best suffix sum *)
```

```
, b   (* the best overall solution found so far *)
)
```

We have already see that both the total sum and maximum prefix sum can be computed efficiently. The maximum suffix sum is symmetric to the prefix sum, and the solution is nearly identical, but with everything symmetrically reversed (i.e., we consider the total sum on the right instead of the left).

The best overall solution then only needs to additionally consider a contiguous subsequence which crosses the midpoint. If s_1 is the best suffix on the left and p_2 is the best prefix on the right, then the best candidate which crosses the midpoint would have the sum $s_1 + p_2$.

Altogether, this gives us the following combination function:

```
fun combine ((p1, t1, s1, b1), (p2, t2, s2, b2)) =
  let
    val p = Int.max (p1, t1+p2)
    val t = t1+t2
    val s = Int.max (s1+t2, s2)
    val b = Int.max (s1+p2, Int.max (b1, b2))
  in
    (p, t, s, b)
  end
```

The complete solution (written simplified in terms of `reduce`) is as follows:

```
fun mcss(lo, hi, f) =
  let
    val z = (0, 0, 0, 0)
    val g = combine (* as defined above *)
    val (_, _, _, b) =
      reduce g z (lo, hi) (fn i =>
        let
          (* for the singleton element x, we need to know the best
           * prefix, suffix, solution, total sum, etc.
           *)
          val x = f(i)
          val v = Int.max (x, 0)
        in
          (v, x, v, v)
        end)
  in
    b
  end
```