Lesson 1-1 Introduction

The Multithreaded DAG Model

DAG = Directed Acyclic Graph : a collection of vertices and directed edges (lines with arrows). Each edge connects two vertices. The final result of these connections is that there is no way to start at some vertex 'A', follow a sequence of vertices along directed paths, and end up back at 'A'.

DAGs can be used for a variety of tasks, including modeling processes in which data flows in a consistent direction through a network of processors.

Each vertex is an operation - like a function call, addition, branch, etc.
Directed edges show how operations depend on one another.

The 'sink' depends on the output of the 'source'
Assume there is always one starting and one exit vertex.

Begin analysis by looking for a starting vertex - this is a vertex where all inputs are satisfied. This vertex can be assigned to any open processor.

Scheduling - taking units of work and assigning it to processors.

How long will it take to run the DAG? A cost model is needed.
Cost Model Assumptions: all processors run at the same speed
1 operation = 1 unit of time
Edges do not have any cost associated with them

Example Sequential Reduction

Reduction - reduce an array to a sum of its elements.
To find the cost of this reduction …. we will only care about the cost of array access and the cost of addition.

How long will it take to execute this DAG with processors?
$T_p(n)$ => ceiling of n/p (time is dependent upon the size of the array) and
$T_p(n)$ => n (then time for each addition)

The additions must be done sequentially.
Both time conditions must be true.

$T_p(n)$ => ceiling of n/p → p will always be at least one. This means a reduction will take n units of time on a PRAM.

$T_p(n)$ => n (then time for each addition)

QUIZ : A Reduction Tree

Assume associativity (a+b)+c = a+(b+c)
Assume n processors.
Assume addition is done in pairs.

What is the minimum time on a PRAM with P = n processors?
The DAG is executed level by level - and each level takes constant time - so all that is needed to calculate the time is to know the levels…. log n.

Work and Span

Work = number of vertices in the DAG = W(n)

Span = longest path through the DAG = D(n) = number of vertices on the longest span

Span is also known as the critical path.

$T_1(n) = W(n)$
$T_{infinity}(n) = D(n)$

QUIZ: Work and Span for Reduction

For the sequential DAG → span = O(n)
For the tree DAG → span = O(log(n))

Basic Work Span Laws

W(n)/D(n) = the amount of work per critical vertex = the average available parallelism in the DAG.

How many processors for the problem? W(n)/D(n)

Span Law → $T_p(n)$ => D(n)
Work Law → $T_p(n)$ => ceiling of W(n)/P

$T_p(n)$ => maximum of {Span Law, Work Law} = {D(n), ceiling of W(n)/P}

Brent's Theorem Part 1 (setup)

Is there an upper bound to execute the DAG? Yes, according to Brent's Theorem

Given a PRAM with P processors….
Break the execution into phases:

1. Each phase has 1 critical path vertex
2. Non-critical path vertices in each phase are independent. This means the vertices in the phase can have edges that enter or exit the phase, but they cannot depend on one another.
3. Every vertex has to be in some phase, and only one phase.

$$\sum_{k=1}^{D} W_k = W$$

How long will it take to execute phase k?

$$t_k = \left\lceil \frac{W_k}{P} \right\rceil \Rightarrow T_P = \sum_{k=1}^{D} \left\lceil \frac{W_k}{P} \right\rceil$$

(time to execute phase k)

QUIZ Brent's Theorem Aside

Use the following equivalencies …

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n+m-1}{m} \right\rfloor = \left\lfloor \frac{n-1}{m} \right\rfloor + 1,$$
$$\left\lfloor \frac{n}{m} \right\rfloor = \left\lceil \frac{n-m+1}{m} \right\rceil = \left\lceil \frac{n+1}{m} \right\rceil - 1,$$

Brent's Theorem Part 2

The upper bound of the time to execute the DAG is:

$$T_p \le \sum_{k=1}^{D} \frac{W_k - 1}{P} + 1$$

which becomes ….

$$T_p \le \frac{W - D}{P} + D$$

This is Brent's Theorem. It says …

The upper limit of time to execute the path, using P processors is <= The time to execute the critical path + the time to execute everything off the critical path using p processors.

**This sets the goal for any scheduler. **

$$\max\left\{D, \left\lceil \frac{W}{P} \right\rceil\right\} \le T_p \le \frac{W - D}{P} + D$$

These two limits are within a factor of 2 with each other.

This implies that you may be able to execute the DAG in a faster time than Brent predicts, but never faster than the lower bound.

Desiderata Speedup, Work Optimality, and Weak Scaling

How can we tell is a DAG is good or bad.

Speedup = best sequential time/ parallel time = $S_p(n) = T_*(n) / T_p(n)$

$T_*(n)$ → depends on the work done by the best sequential algorithm
$T_p(n)$ → depends on the work, the span, n, and p

Ideal Speedup : Linear in P (you want the speedup to be linear with the number of processors).

$S_p(n)$ = Theta(p) = Best Sequential Work/ Parallel Time = $W_*(n)/T_p(n)$

Use Brent's Theorem to get an Upper bound on time.
In the equation shown below… there is still a dependence on n, it is just not shown on the right side.

$$S_p(n) = \frac{W_*(n)}{T_p(n)} \geq \frac{p}{\frac{W}{W_*} + \frac{p-1}{W_*/D}}$$

P = number of processors
The penalty (the denominator) - to get linear scaling, the denominator needs to be a constant.

To get a constant in the denominator:
$W = W_*$ → <u>Work Optimality</u>

<u>Weak Scalability</u>
$P = O(W_* / D)$ → W_*/P = Omega(D) → work per processor has to grow proportional to the span. Span depends on problem size n.

Recap:

Speedup → linear scaling is the goal.
To achieve linear scaling → the work of the parallel algorithm should match the best sequential algorithm and the work per processor should grow as a function of n.

Basic Concurrency Primitives

<u>The Divide and Conquer Scheme</u>

```
reduce(A[0:n-1])
  if n ≥ 2 then
    a ← reduce(A[0:n/2-1])
    b ← reduce(A[n/2:n-1])
    return a+b
  // else n=1
  return A[0]
```

This is the sequential version of the divide and conquer scheme.

Note that the two recursive calls are independent, and will now be called <u>SPAWN</u>

Spawn is a signal to either the compiler or the runtime system that the target is an independent unit of work. The target may be executed asynchronously from the caller.

SYNC - the dependence between a and b and the return statement. These have to be combined. Sync is used to combine the dependent statements.

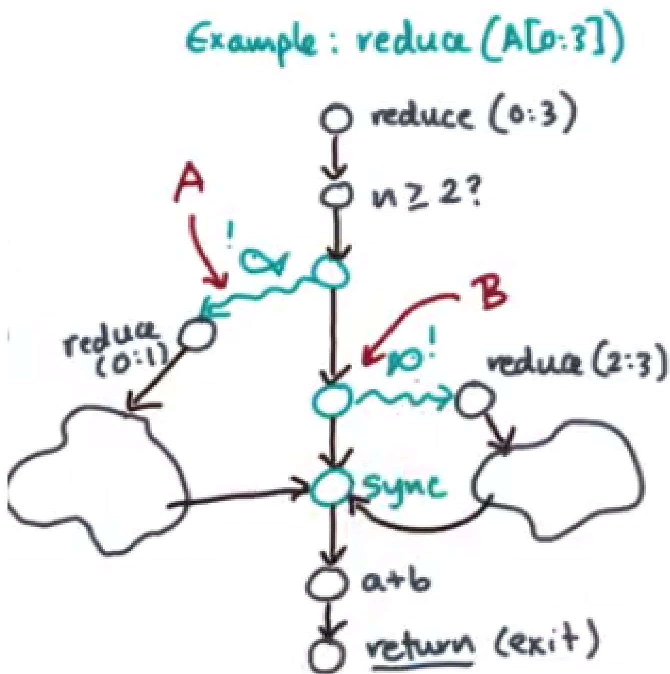
To which Spawn does a given Sync apply? The sync matches any spawn in the same frame.

Nested Parallelism = There is always an implicit sync before returning to the caller.

The spawn creates two independent paths - one path carries the new work, and one path continues carrying on after the spawn.

QUIZ: A Subtle Point About Spawns

The above cursive reduction uses two spawns - are they both necessary? You can eliminate B but not A.



If you eliminate the A path -- you eliminate concurrency -- this is bad.

If you eliminate the B path -- the two sub graphs can be executed concurrently.

Basic Analysis of Work and Span

Many of the analysis tools used on sequential algorithms can be used on parallel algorithms.

Want to analyze work and span.

Analysis : $W(n) = ?$ $D(n) = ?$

$$W(n) = \begin{cases} 2 \cdot W(\frac{n}{2}) + O(1) & n \geq 2 \\ O(1) & n \leq 1 \end{cases}$$

$$= O(n)$$

$$D(n) = \begin{cases} D(\frac{n}{2}) + O(1) & n \geq 2 \\ O(1) & n \leq 1 \end{cases}$$

Assume each spawn and sync is a constant time operation. And can be ignored for analysis.

Analyzing work is counting total operations, end up with linear work O(n).

Analyzing Span - a spawn creates two paths, the critical path is the longer of the two paths.

Desiderata For Work and Span

The goals of a parallel algorithm designer:

1. Work optimality - Achieve a degree of work that matches the best sequential algorithm.
2. Find algorithms with polylogarithmic span. $D(n) = O(\log^k n)$ this is low span
   This insures the average available parallelism grows with n.

Concurrency Primitive Parallel For

All iterations are independent of one another.
A parfor creates 'n' independent sub paths.
The end of a parfor loop will include an implicit syncpoint.
The Work of a parfor is $W_{parfor}(n) = O(n)$
The Span of a parfor is $D_{parfor}(n) = O(1)$ in theory, but in practice it will grow with n, especially if n is really large.

QUIZ Implementing ParFor

The DAG executes the spawns sequentially, one after another. This leads to a bottleneck. The Span grows with n. This is bad.

Implementing Par For Part 2

Implement par for as a procedure call (ParForT). This is a better way to implement a parallel for loop. The span will now grow logarithmically with n.

For the rest of this course, assume the ParForT implementation.

D(n) = O(log n)

QUIZ Matrix Vector Multiply

If a loop carries a dependence, then it cannot be parallelized with a par for.

Data Races and Race Conditions

```
// Computes: y ← y + A·x
for i ← 1 to n do        // Loop 1
    for j ← 1 to n do    // Loop 2
        y[i] ← y[i] + A[i,j]·x[j]
```

If we look at the nested loops, we see that the innermost loop there are iterations of ‘j’ that write to the same ‘i’.

Data Race = at least one read and one write can happen at the same data location at the same time.

Race Condition = a data race that causes an error.

**A data race does not always lead to a race condition. **

Vector Notation

t[1:n] ← A[i, 1:n] * x[1:n]   This is a more compact form of the parfor loop.
t[:] ← A[i, :] * x[:]

This can be further reduced to : y[i] ← y[i] + reduce(t)