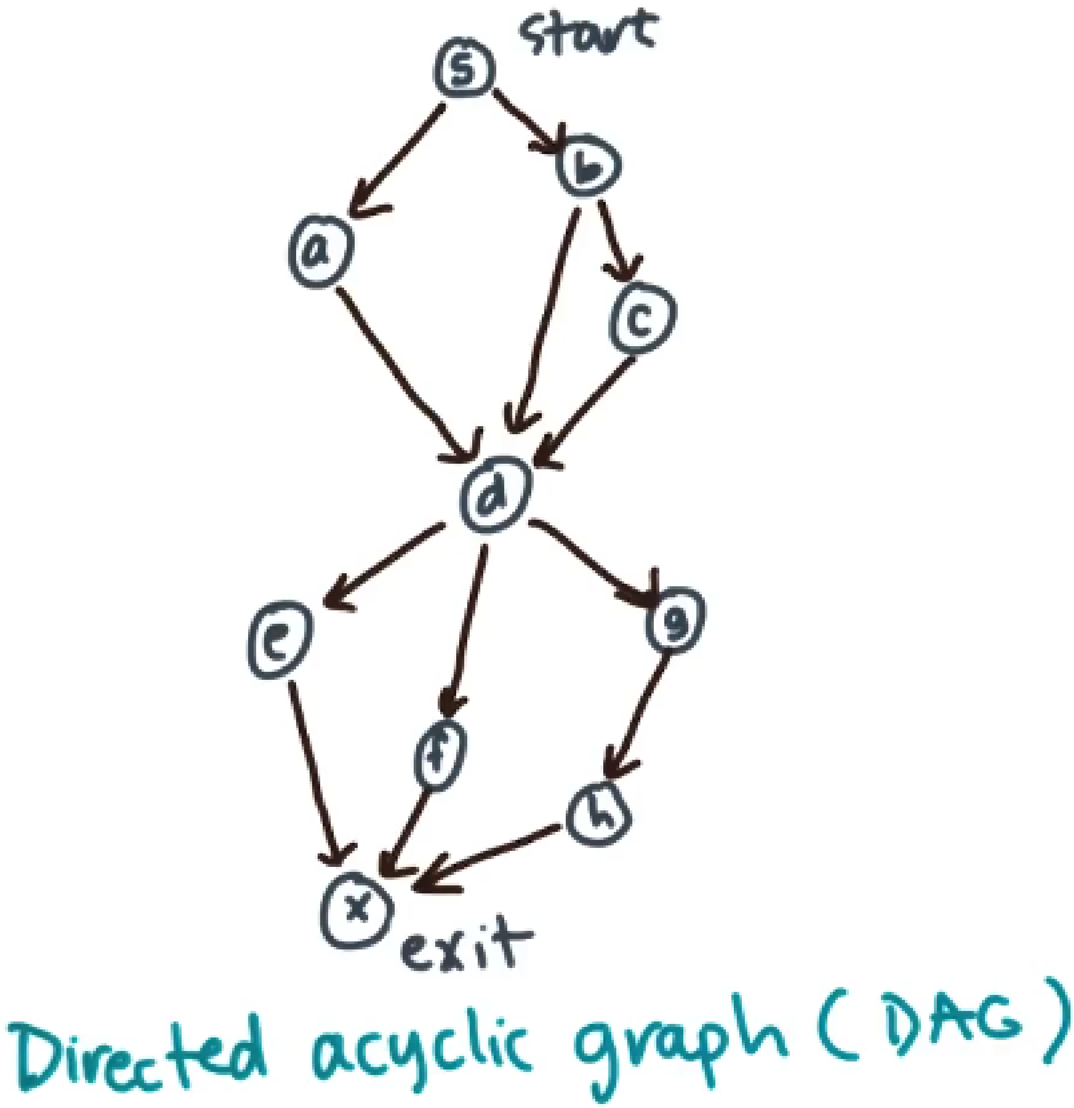
Introduction to the Work-Span Model

Introduction

1. Dynamic Multithreading Model
   - Computation can be represented by a directed acyclic graph
     - Node is a task
     - Edge is a dependency between tasks
   - Good DAG has relatively few dependencies compared to the number of tasks
2. Psuedocode notation or programming model for writing down the algorithm
   - Notation is defined to generate a computational DAG
   - Give a DAG, some runtime system figures out how to map it to cores and execute it

The Multithreaded DAG Model

1. Each vertex is an operation
   - Addition, function call, branch, etc.
2. Each edge shows how operations depend on each other
   - Sink depends on source
3. Parallel RAM machine used to run the application
   - PRAM is the parallel-computing analogy to the random-access machine used by sequential algorithm designers
   - Scheduling: Assigning operations to processors
4. How long will it take to execute the DAG?
   - Need a cost model to evalutate
   - Assumptions:
     - Each operation executes at the same speed
     - 1 operation = 1 unit of time
     - No edge costs



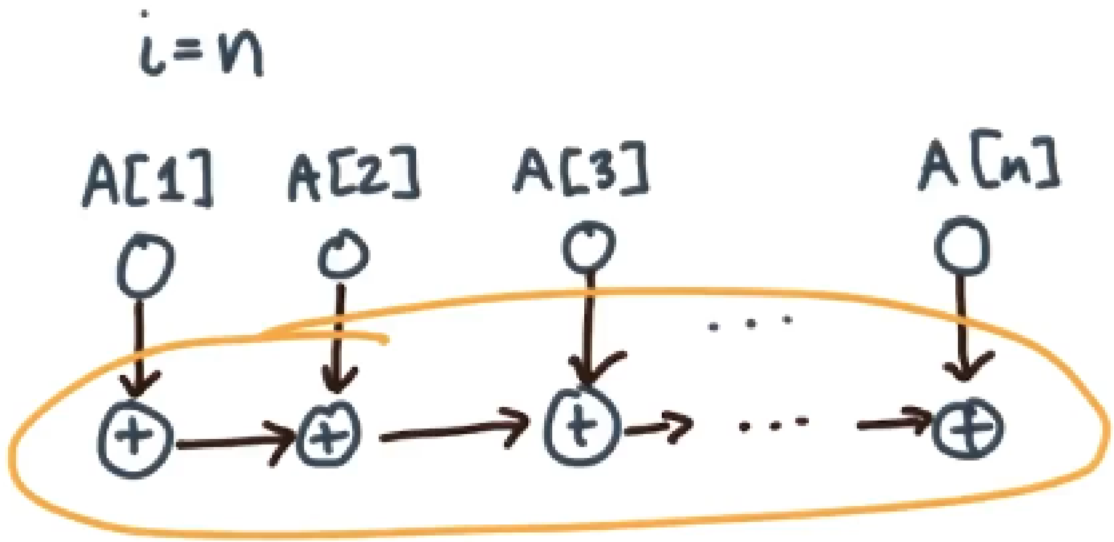
Directed Acyclic Graph

Sequential Reduction Example

1. Consider the basic reduction example:

```
int n;
int A[n];
int s = 0;
for(int i = 0; i < n; i++) {
    s = s + A[i];
}
```

2. The DAG looks like this:

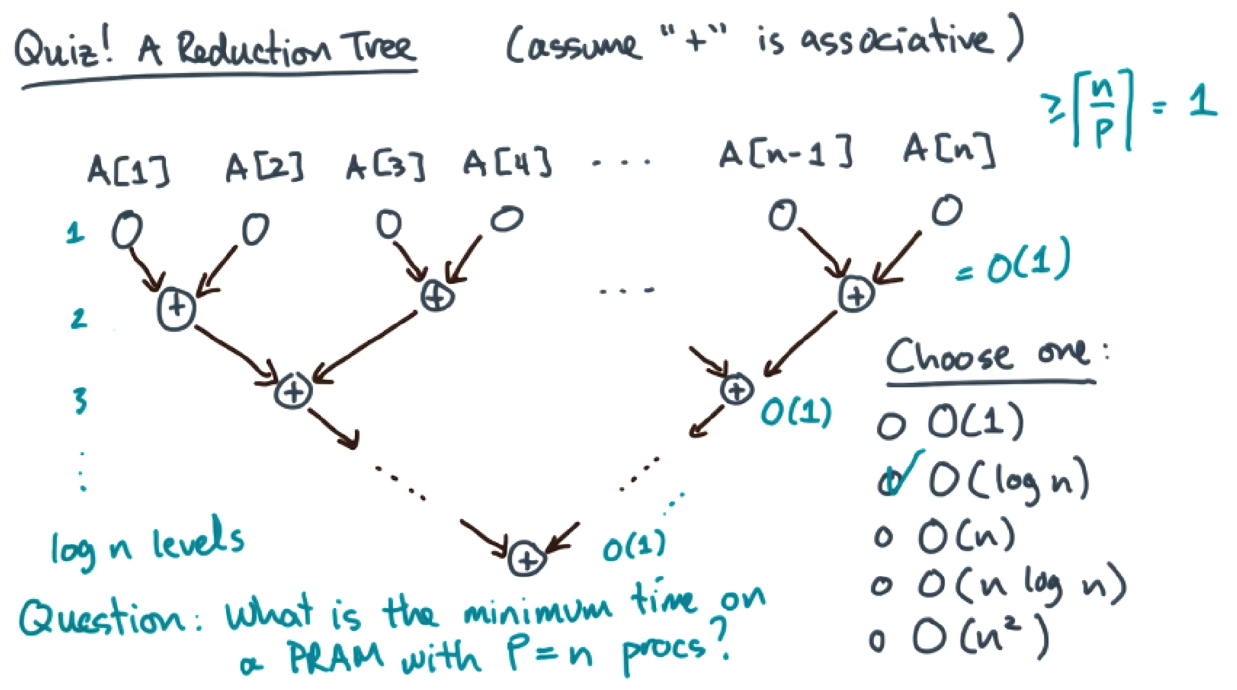


Reduction Example

3. The total cost (P is the number of processors)
   - Tp(n) >= ceil(n/P) for loads
   - Tp(n) >= n for additions
   - Because P >= 1, we can simplify to Tp(n) >= n

A Reduction Tree

1. Assume + is associative (a + (b + c) == (a + b) + c)
2. What is the minimum time on a PRAM with P = n processors?
   - O(log(n))
   - Each level takes constant time, so the question is how many levels exist?



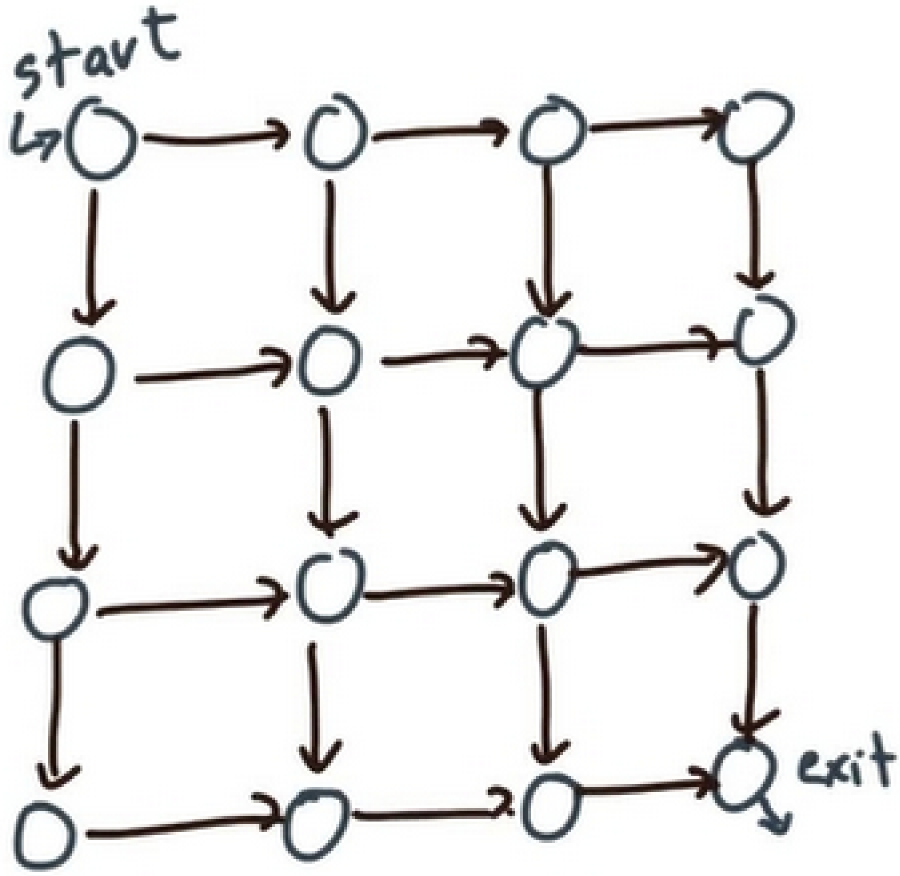
Directed Acyclic Graph Quiz 1

Work and Span

1. Which DAG is better, sequential or tree-based?
   - Typically, you would prefer the implementation that can exploit more parallelism (tree-based)
2. Given a DAG, there are two questions to answer:
   - Work W(n): How many total vertices are there?
   - Span D(n): How many vertices are on the longest path?
     - Also called the critical path
     - Historically was called depth
3. What can we say about the total cost Tp(n)?
   - If all operations have unit cost and P=1, T(n) = W(n)
   - If all operations have unit cost and P=inf, T(n) = D(n)

Work and Span Quiz

1. What are the work and span of this DAG?
   - W(n) = 16
   - D(n) = 7



Directed Acyclic Graph Quiz 2

Work and Span for Reduction

1. What is the span of the sequential vs tree-based reduction example?
   - Sequential: D(n) = O(n)
   - Tree-based: D(n) = O(log(n))

Basic Work Span Laws

1. The ratio of work to span measures the amount of work per critical path vertex
   - W(n)/D(n) tells you the average available parallelism in the DAG
2. On a PRAM system, we want at least W/D processors to keep them busy
3. Span law: Tp(n) >= D(n)
4. Work law: Tp(n) >= ceil(W(n)/P)
5. Both laws must be true simultaneously, so we can combine them
6. Work-span law: Tp(n) >= max(D(n), ceil(W(n)/P))

Brent's Theorem - Part 1

1. Is there an upper-bound on the total cost?
   - Yes; Brent's theorem proves it
2. Deriving Brent's theorem

- Break execution into phases:
  - Each phase has 1 critical path vertex
  - All non-critical path vertices in each phase are independent
  - Every vertex must appear in some phase
- Each phase k has Wk vertices
  - Sum(Wk) over all k = W
- How long will it take to execute phase k?
  - Tk = ceil(Wk/P) => Tp = sum(Tk)

Brent's Theorem Aside

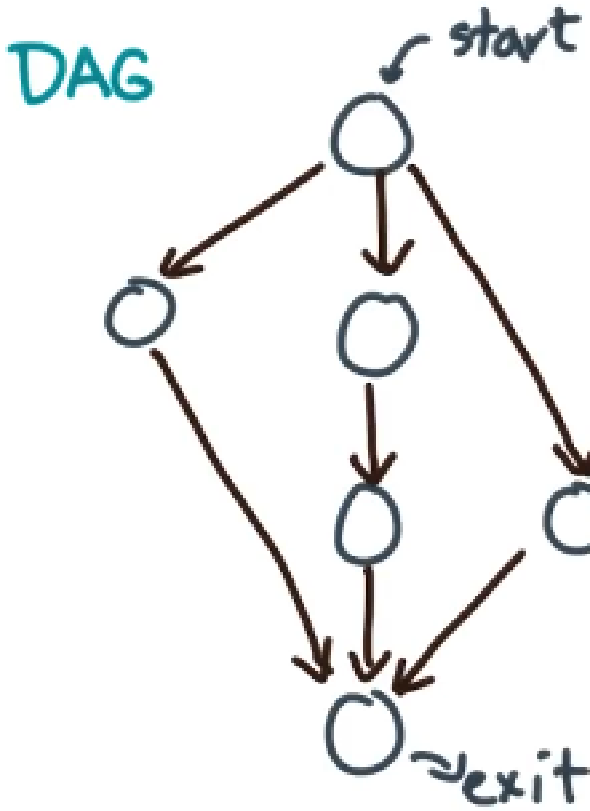
1. Let a and b be integers where a, b > 0
2. Which of these identities are true?
   - ceil(a/b) = floor((a + b - 1)/b)
   - floor(a/b) = floor((a + b + 1)/b)
   - ceil(a/b) = floor((a - 1)/b) + 1
   - floor(a/b) = floor((a + 1)/b) - 1
3. All are true

Brent's Theorem - Part 2

1. Using the floor/ceiling quotient identities:
   - Tp = sum(ceil(Wk/P))
   - Tp = sum(floor((Wk-1)/P)) + 1
   - Tp <= sum((Wk-1)/P) + 1
   - Tp <= (W-D)/P + D
2. Because we're solving for the upper bound, we can eliminate the floor
   - floor(x) <= x
3. Brent's theorem sets a goal for any scheduler
4. max(D, ceil(W/P)) <= Tp <= (W-D)/P + D
   - Upper and lower bound are within a factor of 2 of each other
   - May be able to execute faster than Brent's theorem predicts

Applying Brent's Theorem

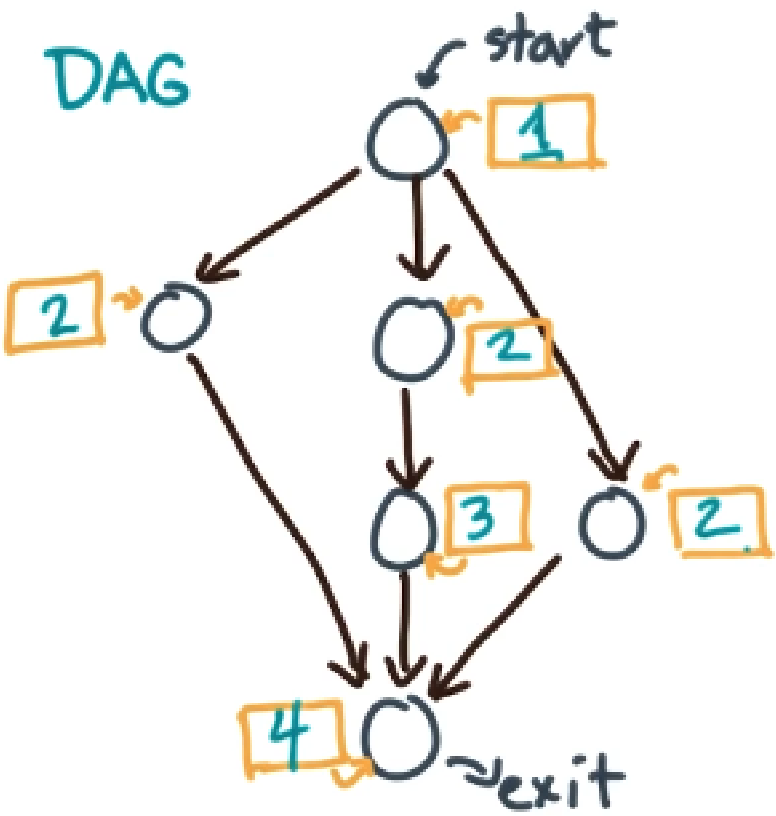
1. What is the upper bound for this DAG executed on a 2-processor PRAM computer?
   - W = 6, D = 4, P = 2
2. Tp <= (W-D)/P + D
   - Tp <= 5



Directed Acyclic Graph Quiz 3

The Slack in Brent's Bound - Part 1

1. The lower bound in the previous example is 4
2. Can you relabel the DAG such that it takes 5 time units?
   - By scheduling both non-critical path nodes in the same phase, we must take two units to execute them since P=2
   - This is what makes scheduling a tricky subject



Directed Acyclic Graph Quiz 4

Desiderata Speedup - Work Optimality - and Weak Scaling

1. Speedup: Best sequential time / parallel time
   - Sp(n) = Ts(n) / Tp(n)
   - Ideal speedup: Linear in P
     - Want P speedup when using P processors to parallelize an algorithm
2. Can use Brent's theorem to get an upper bound on time and therefore a lower bound on speedup
   - Sp(n) >= Ws / ((W-D)/P + D)
   - Sp(n) >= P / ((W/Ws) + (P-1)/(Ws/D))
     - This says that if we want linear scaling, the denominator must be a constant
     - Work-optimality: W = Ws (If we get a very parallel algorithm by dramatically increasing the work relative to the best sequential algorithm, it's actually bad for speedup)
     - Weak-scalability: P should be proportional to Ws/D (Work per processor must grow proportional to the span, which depends on problem size)

Which Parallel Algorithm is Better

1. Consider two parallel algorithms for solving the same problem:
   - Algorithm 1: W1(n) = n^2log(n), D1(n) = log(n)
   - Algorithm 2: W2(n) = n^2, D2(n) = n

Basic Concurrency Primitives

1. Spawn: Signal to the compiler or runtime system that the target is an independent unit of work

    - Target of a spawn is always either a function or procedure call
2. Sync: Signal to the compiler or runtime system that there are dependent units
3. Sync waits for any spawn that has occurred so far within the same stack frame
4. There's always an implicit sync at the return
    - This results in nested parallelism within the DAGs produced by this model

A Subtle Point about Spawn

1. Consider the following program:

```
reduce(A[0:n-1])
if n >= 2
{
    a <- spawn reduce(A[0:n/2-1])
    b <- spawn reduce(A[n/2-1:n-1])
    sync
    return a+b
}
else
{
    return A[0]
}
```

2. Which spawn may be eliminated without increasing the span (asymptotically)?
    - A but not B
    - B but not A (true)
        - If we eliminate A, it increases the length of the critical path because B must wait for A to finish. If we eliminate B, we're only affecting the middle path, which isn't critical
    - A or B, but not both
    - Neither - must keep both

Basic Analysis of Work and Span

1. Analyze the complexity for the recursive reduction:
    - We can write a recurrence relation and solve it:
        - T(n) = 2 * T(n/2) + O(1) if n >= 2
        - T(n) = O(1) if n <= 1
    - Solve using the Master Theorem: O(n)

```
reduce(A[0:n-1])
if n >= 2
{
    a <- spawn reduce(A[0:n/2-1])
    b <- spawn reduce(A[n/2-1:n-1])
    sync
    return a+b
}
else
{
    return A[0]
}
```

2. Assume spawn and sync are constant time (O(1))
    - Recurrence for work is therefore the same as the sequential algorithm
3. Span is slightly different; D = max(Da, Db) + O(1) because span is the length of the critical path in the DAG

- D(n) = D(n/2) + O(1) if n >= 2
- D(n) = O(1) if n <= 1
- D(n) = O(log(n))

Solve a Recurrence

1. What is the solution to this recurrence?
   - D(n) = D(n/2) + O(1) if n >= 2
   - D(n) = O(1) if n <= 1
2. O(log(n))

Desiderata for Work and Span

1. Work and span goals when developing a parallel algorithm:
   - Achieve a degree of work that matches the best sequential algorithm
     - Achieving work-optimality (O(n) if humanly possible)
   - Goal for span is polylogarithmic ("low" span)
     - D(n) = O(log^k(n))
     - Ensures that the average available parallelism grows with n
   - Motivation: W/D = O(n/log^k(n)) grows with n, "close" to linearly
2. Always have to use judgment to decide if a parallel algorithm has good work and span

Concurrency Primitive Parallel For

1. A parfor loop indicates that all iterations are independent of one another
   - Iterations can be executed in any order
   - In terms of a DAG, this creates n independent subpaths
2. By convention, the end of a parfor loop includes an implicit sync point
   - Work of parfor W(n) = O(n)
   - Span of parfor D(n) = O(1), but only in theory

Implementing Par For - Part 1

1. Consider a parfor loop implemented as follows:

```
for i <- 1 to n do
    spawn foo(i)
sync
```

2. Assuming the cost of "foo" is O(1), what is the span of this implementation?
   - O(1)
   - O(logn)
   - O(n) (true)
     - The threads are spawn in series which causes the critical path of our DAG to be n
   - O(nlogn)

Implementing Par For - Part 2

1. Instead of the above implementation, consider implementing a parfor as follows:

```
ParForT(foo, a, b)
{
    let n = b - a + 1
    if n = 1
    {
        foo(a)
```

```
    }
    else
    {
        let m = a + floor(n/2)
        spawn ParForT(foo, a, m-1)
        ParForT(foo, a, m-1)
        sync
    }
}
```

2. The span of this implementation is:
   - D(n) = O(log(n)) assuming foo = O(1)
3. Assume this implementation when considering parfor loops

Matrix Vector Multiply

1. Consider the following matrix-vector multiply implementation:

```
for i <- 1 to n do
    for j <- 1 to n do
        y[i] <- y[i] + A[i,j] * x[j]
```

2. The work is W(n) = O(n^2)
3. Which for loops may be safely converted into parallel for loops?
   - Only loop 1 (true)
   - Only loop 2
   - Both loops 1 and 2
   - Neither loop 1 nor 2

Data Races and Race Conditions

1. Data race: At least one read and one write may happen at the same memory location, at the same time
2. Race condition: A data race causes an error

Putting it all Together - Part 1

1. For the matrix-vector multiply example, the work is W(n) = O(n^2)
2. What is the span?
   - O(1)
   - O(log(n))
   - O(n) (true)
     - Technically, the span is log(n) + n (log(n) from the outer loop, n from the inner loop) but this is still O(n)
   - O(nlog(n))
   - O(n^2)

Putting it all Together - Part 2

1. Modify the matrix-vector multiply algorithm as follows:

```
parfor i <- 1 to n do
{
    let t[1:n] be a temporary array
    parfor j <- 1 to n do
    {
        t[j] = A[i,j] * x[j]
```

```
    }
    y[i] <- y[i] + reduce(t)
}
```

2. What is the span?
   - O(log(n)) (true)
   - O(log^2(n))
   - O(log^3(n))

Vector Notation

1. Can use vector notation in pseudo-code to make the notation more compact

```
parfor i <- 1 to n do
{
    let t[1:n] be a temporary array
    t[:] <- A[i,:] * x[:] // implicit parfor
    y[i] <- y[i] + reduce(t)
}
```

2. Element-wise operations have linear work and logarithmic span
3. This can further be simplified by eliminating the temporary array
   - Important to remember when considering storage costs

```
parfor i <- 1 to n do
{
    y[i] <- y[i] + reduce(A[i,:] * x[:])
}
```

Conclusion

1. Good algorithms are work-optimal and have low span
2. Divide-and-conquer is an excellent paradigm in parallel algorithm development as well
3. Separate how you express concurrency from how you execute it
   - This model ignores communication costs, which are important once the size of the computation gets big enough