Timing Analysis for Parallel Applications

Time Decompositon

- Parallel execution time can be divided into:
  - Actual computation time +
  - Communication time

$$t_{parallel} = t_{comp} + t_{comm}$$

  - If there are m non-parallel message steps overall, then

$$t_{comm} = m * t_{message}$$

Message Time Decomposition

- Message time can be divided into:
  - **Latency** (or start-up time) +
  - (number of data communicated)*(delay per datum)

$$t_{message} = t_{startup} + n * t_{datum}$$

$1/ t_{datum}$ is often called "**bandwidth**", the amount of data per unit time.

Latency Hiding

- In order to prevent $t_{message}$ from destroying any speedup due to parallelism, we can try the following:
  - While a processing element is awaiting a message, perform some computation that doesn't require the message.
- Note that we are really trying to hide the entire communication cost, not just the "latency" component of it.

Latency Hiding (2)

- One technique for hiding latency is "multiprogramming":
  - On a single processor, run more than one process.
  - While one process is awaiting a message, another could be doing useful computational work.
  - This requires that process-switching be relatively efficient (e.g. using threads rather than processes).
- The ratio of processes to processors is sometimes called the "parallel slackness".

Parallel Time Complexity

- We assume familiarity with O, Ω, and Θ notation.
- Their use is to bound the time complexity as a function of the problem size “n”.

Complexity Example

- Matrix-vector multiplication:
  - n x n matrix
  - n element vector
- Assume n processors
  - Every processor has a row of the matrix
  - Each row is multiplied by the vector **simultaneously**
  - It takes O(n) to multiply one row, so

$$t_{comp}(n) \in O(n)$$

Matrix-Vector Multiplication

- If the matrix first had to be *distributed* in order for the multiplication to take place, then the cost of **distributing** the rows from one processing element is $O(n^2)$, while the cost of **collecting** the result is $O(n)$.

- Therefore, the asymptotic parallel time is the same as the obvious sequential time.

- But as we are tying up p processors during this time, this is **not** very efficient.

Matrix-Vector Multiplication

- If the **same matrix is to be used many times**, then the overhead of distribution gets less and less significant as the number of uses increases.

- In this case, the parallel time approaches O(n), which is an improvement over the $O(n^2)$ serial method.

Effort or “Cost”

- Let T_p be the time a particular algorithm takes on p processors.

- Assuming the processors don’t have other work to do, the **effort** expended is thus

  $p*T_p$

Cost Optimality

- A **cost-optimal parallel algorithm** is defined to be one in which the **effort**, as a function of problem size, is bounded by a **constant** times the **sequential effort**:

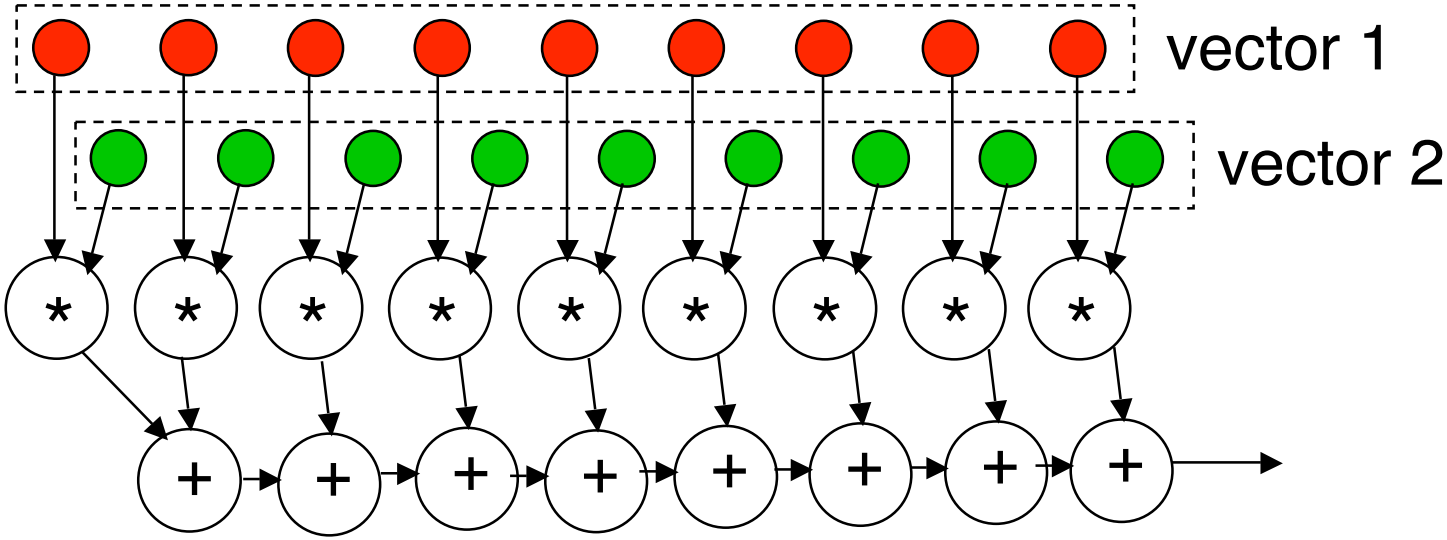
$$E_{parallel} \in O(E_{sequential})$$

- The one-shot matrix-vector multiplication is **not** cost-optimal for distributed memory using the technique described, whereas multiplication repeated at least n times is (since the cost of distributing the matrix is incurred only once).

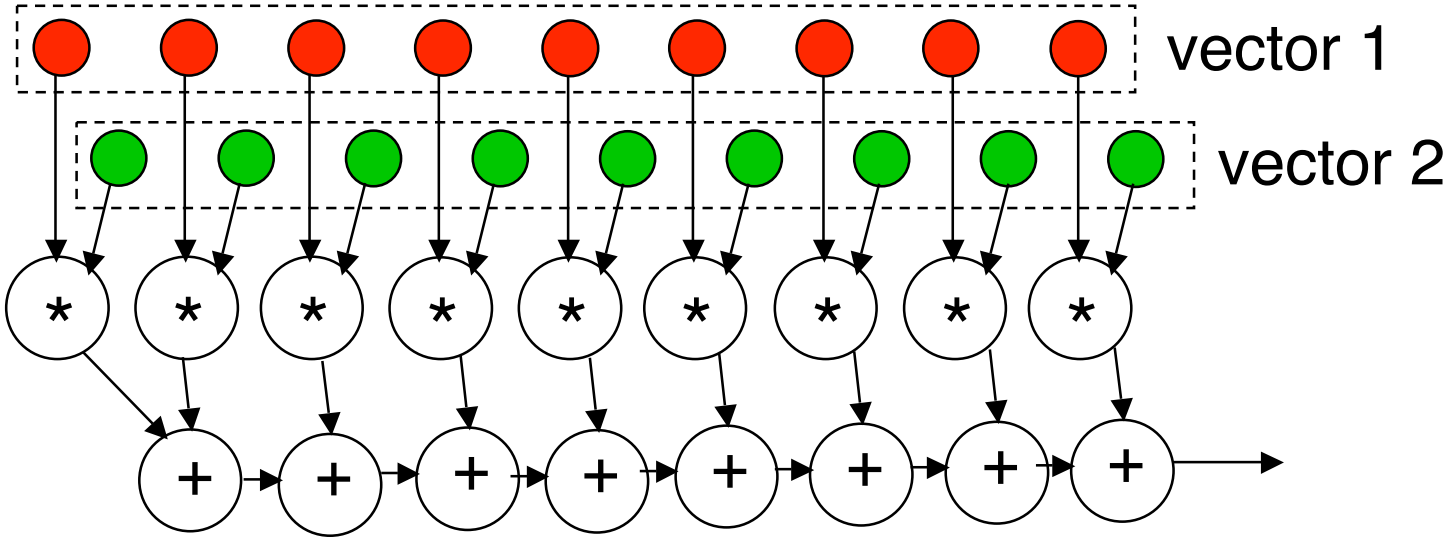
Dependency Graphs

Using **Dependency Graphs** to Illustrate Algorithms

- A vector inner product can be shown thus (ignoring distribution for now):



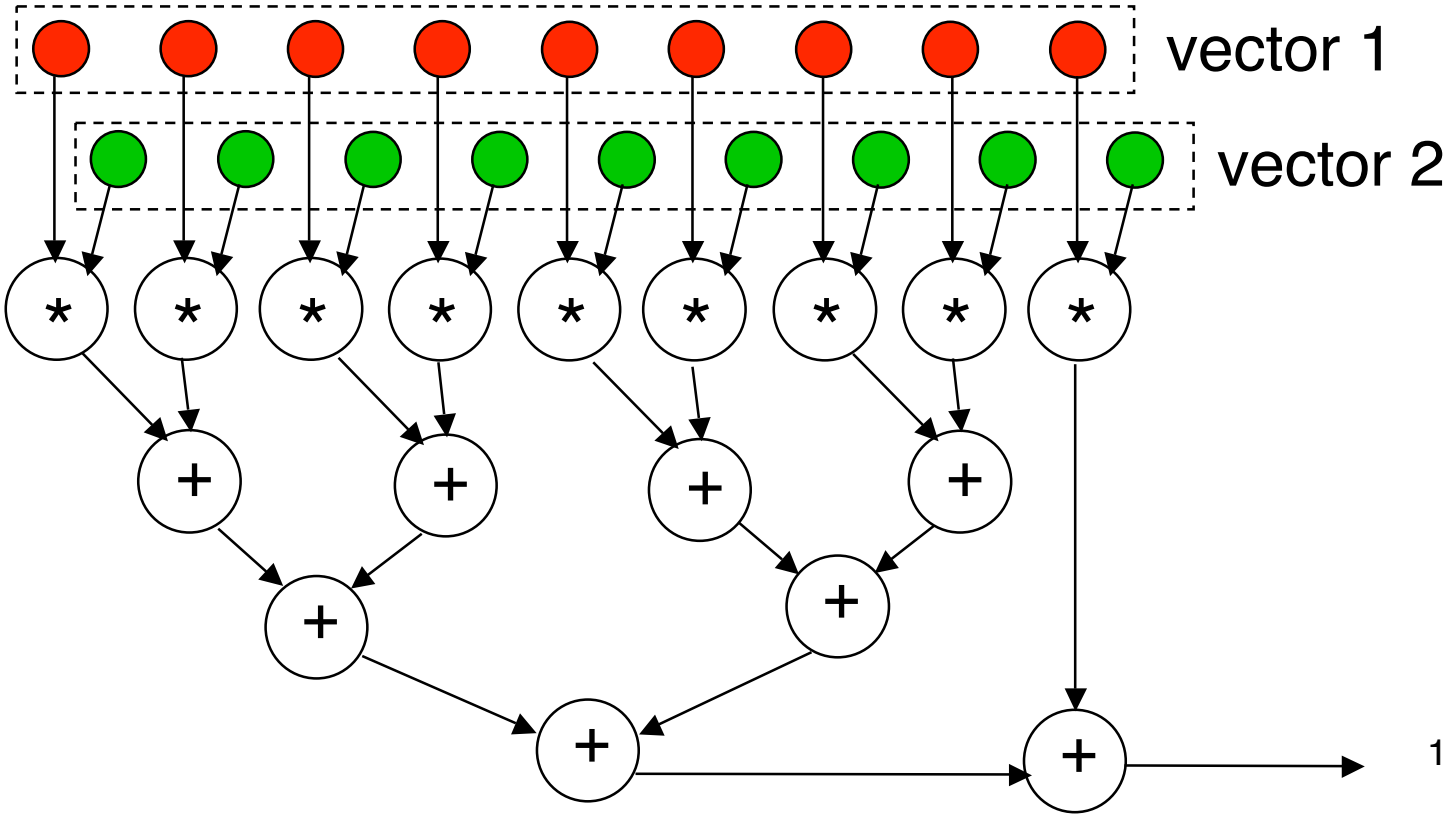
Using **Dependency Graphs** to Illustrate Algorithms



- Assume **unit time** for each operation.
- The time is proportional to path length.
- The **longest path** length for an n-element vector is O(n), similarly to serial.

Using Graphs to Illustrate Algorithms

- Restructuring the + nodes as a **tree** gives us **faster** performance on n processors.



Algorithm Analysis

- The previous tree gives us O(log n) time on n processors.
- Is it cost optimal?

Scaling Down Processors

- As the size n of the vector grows very large, we can divide the additions up among p processors, p << n,

  adding the elements within each processor sequentially and **only funnelling to a tree at the end**.

- The time for large n is then dominated by the sequential adds, which is O(n) for a given p.

- Is this cost optimal?

Generalization of Scale-down

- Whenever the total number of operations (including communication as an operation) in the parallel case is proportional to the serial complexity, we can achieve cost optimality by scaling down the number of processors relative to the problem size.

- The general concept is captured by **Brent's Lemma**.

Brent's Lemma

- If an algorithm A entails m operations and can be done in parallel time t with *some* number of processors,

  then p processors can execute A in time

  $$t + (m-t)/p$$

  assuming any added scheduling time can be ignored.

Brent's Lemma Summarized

- t = time on *some* number of processors (could be arbitrarily-large)
- m = number of operations (each unit time)
- time on p processors is $\leq$ t + (m-t)/p

Application of Brent's Lemma

- To achieve cost optimality for **vector inner product**, use **n/(log n)** processors.
- We observed that product can be done in time O(log n) with arbitrarily-many processors.
- Brent's lemma says it can be done with p = n/(log n) processors in time

$$\underbrace{\log n}_{t} + \underbrace{(2n-1}_{m} \underbrace{-\log n)}_{-t} / \underbrace{(n/\log n)}_{p}$$

Application of Brent's Lemma

$\log n + (2n-1-\log n) / (n/\log n)$

$= \log n + 2 \log n - (\log n)/n - (\log n)^2/n$

which is $O(\log n)$.

Proof of Brent's Lemma (1)

- Consider the graph of the algorithm done with some number of processors in time t.
- Let s_i be the number of operations done at the i^{th} **level**, i.e. at "time" i.
- On p processors, we can **reschedule** the s_i operations in time $\lceil s_i/p \rceil$ since they are **independent**.

Proof of Brent's Lemma (2)

- The total computation can therefore be done on p processors in time

$$\text{sum}(i = 1 \text{ to } t, \lceil s_i/p \rceil)$$

Proof of Brent's Lemma (3)

$$\text{sum}(i = 1 \text{ to } t, \lceil s_i/p \rceil)$$

is bounded by

$$\text{sum}(i = 1 \text{ to } t, (s_i+p-1)/p)$$

$$= \text{sum}(i = 1 \text{ to } t, s_i/p) + \text{sum}(i = 1 \text{ to } t, p/p) - \text{sum}(i = 1 \text{ to } t, 1/p)$$

$$= m/p + t - t/p$$

$$= t + (m-t)/p, \text{ as stated.}$$

Illustration of Brent

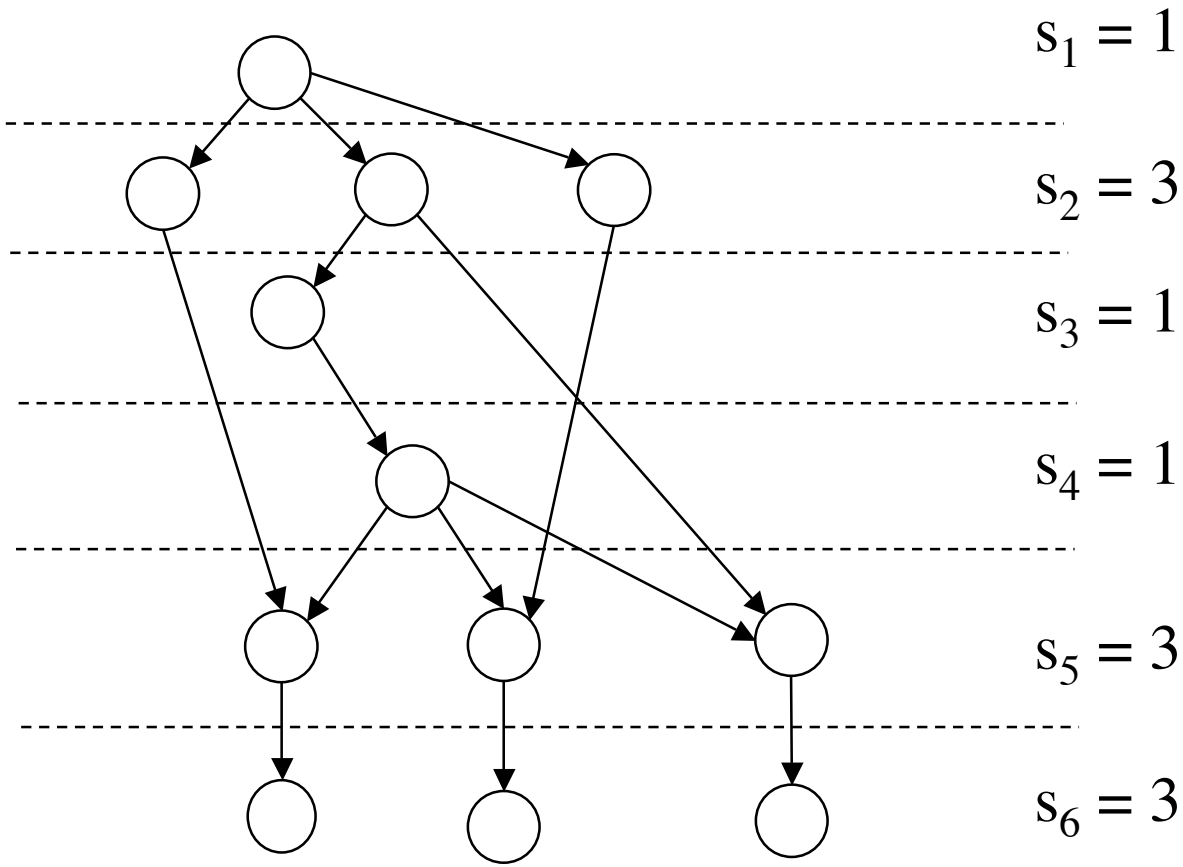
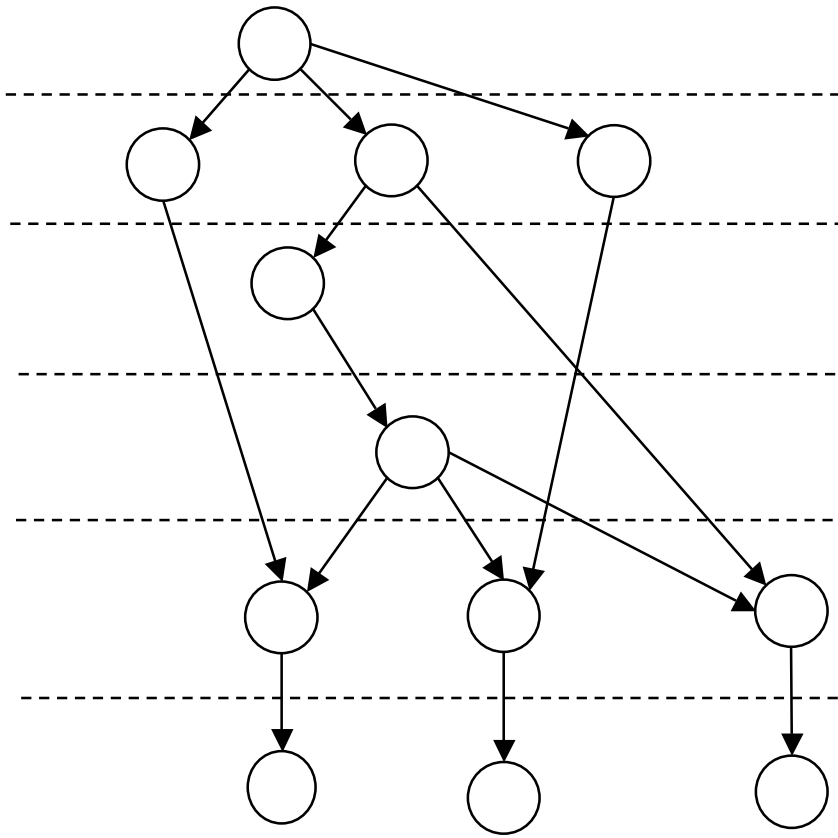


Illustration of Brent for 2 processors

Brent's bound predicts: $t + (m-t)/p = 6 + (12-6)/2 = 9$



	time on 2 processors
$s_1 = 1$	1
$s_2 = 3$	2
$s_3 = 1$	1
$s_4 = 1$	1
$s_5 = 3$	2
$s_6 = 3$	2

total time = 9

Graph Exercise

- By the **prefix sum problem**, we mean that of computing from an array

  $x_0, x_1, x_2, \ldots, x_{n-1}$

  the array

  $(x_0), (x_0+x_1), (x_0+x_1+x_2), \ldots, (x_0+x_1+x_2+\ldots+ x_{n-1})$
- Can this problem be sped up using parallelism?
- Is there a cost optimal version?