More on parallel computation models, Brent’s theorem for runtime upper bound

Lower bound for Tp

Lower bound for Tp

- In the best case, the total work required by the algorithm is evenly divided between the p processors.
- Hence Tp is lower bounded by $T1/p \leq Tp$, where T1 is the time taken by a single processor.
- The equality gives the best case scenario. This is the lower bound we are trying to achieve as parallel algorithm designers.

Useful upper bound for Tp

Useful upper bound for Tp

- What happens when we have infinite number of processors?
- One might suspect the compute time for an algorithm would then be zero.

Useful upper bound for Tp

- What happens when we have infinite number of processors?
- One might suspect the compute time for an algorithm would then be zero.
- But this is often not the case, because algorithms usually have an inherently sequential component to them.
- For example, suppose we represent our algorithm as a collection of computations, then if the output of one computation is used as the input to another, the first must complete before the second can begin.
- Remember Amdahl’s Law, which considered sequential computations?

Useful upper bound for Tp

- What happens when we have infinite number of processors?
- One might suspect the compute time for an algorithm would then be zero.
- But this is often not the case, because algorithms usually have an inherently sequential component to them.
- For example, suppose we represent our algorithm as a collection of computations, then if the output of one computation is used as the input to another, the first must complete before the second can begin.
- Remember Amdahl's Law, which considered sequential computations?
- The above intuition can be made rigorous by representing the dependencies between operations in an algorithm using a directed acyclic graph (DAG).

Constructing a DAG from an algorithm

Constructing a DAG from an algorithm

- Each fundamental unit of computation is represented by a node.

Constructing a DAG from an algorithm

- Each fundamental unit of computation is represented by a node.
- We draw a directed arc from node u to node v if computation u is required as an input to computation v.

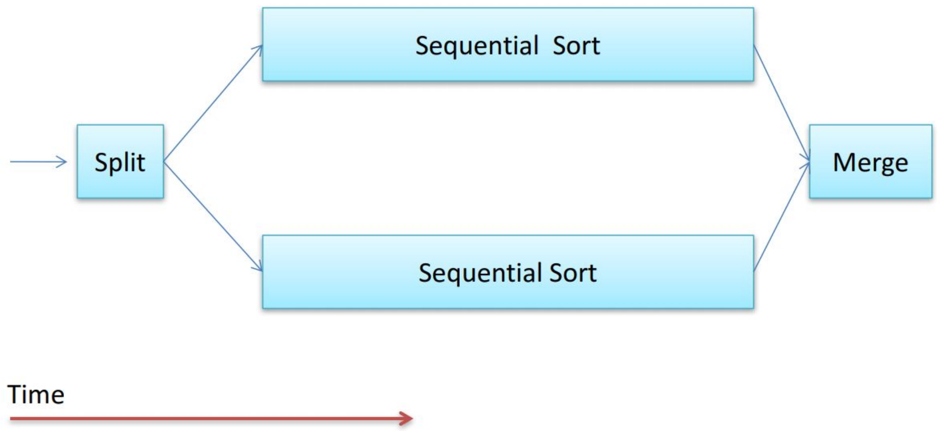
Constructing a DAG from an algorithm

- Each fundamental unit of computation is represented by a node.
- We draw a directed arc from node u to node v if computation u is required as an input to computation v.
- The resulting graph is not guaranteed to be connected— it is possible to have calculations that are completely independent of each other.

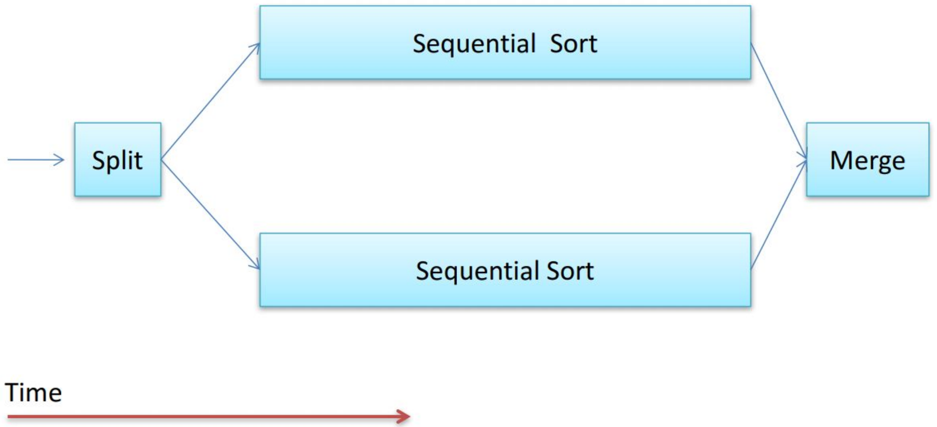
Constructing a DAG from an algorithm

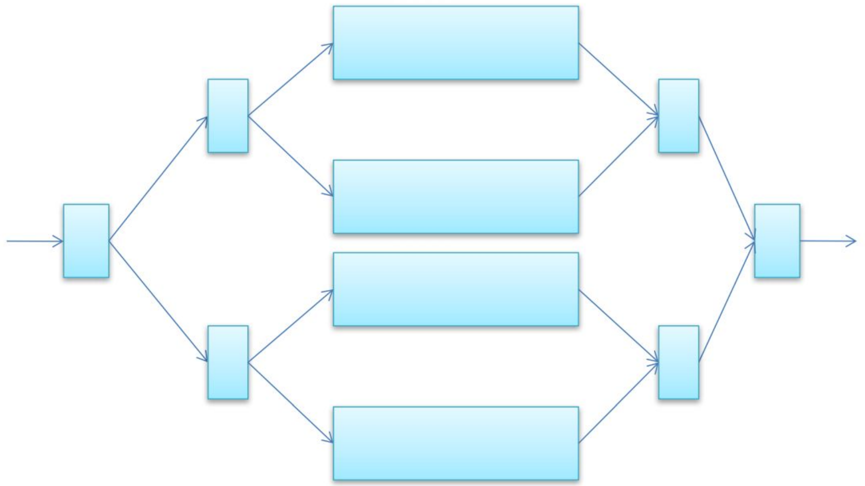
- Each fundamental unit of computation is represented by a node.
- We draw a directed arc from node u to node v if computation u is required as an input to computation v.
- The resulting graph is not guaranteed to be connected— it is possible to have calculations that are completely independent of each other.
- However, each connected component is acyclic.

Parallel DAG for mergesort (2-cores)



Parallel DAG for mergesort (2-cores, 4-cores)





How an algorithm is executed on a single machine?

How an algorithm is executed on a single machine?

- Look for the leaves of the tree, since these depend on no prior computations.

How an algorithm is executed on a single machine?

- Look for the leaves of the tree, since these depend on no prior computations.
- Evaluate all of the leaf nodes and continue through each layer of the DAG until we reach the root node (i.e. return the output).

How an algorithm is executed on a single machine?

- Look for the leaves of the tree, since these depend on no prior computations.
- Evaluate all of the leaf nodes and continue through each layer of the DAG until we reach the root node (i.e. return the output).
- It takes time proportional to the number of nodes in the graph (assuming each node represents a unit of computation which takes constant time).
- So, we define work to be

$$T1 = \text{number of nodes in DAG}$$

Different DAG layers cannot be computed in parallel

Different DAG layers cannot be computed in parallel

- Without loss of generality, assume our DAG is a tree, so the levels are well defined.

Different DAG layers cannot be computed in parallel

- Without loss of generality, assume our DAG is a tree, so the levels are well defined.
- Let the root of the tree (i.e. the output of the algorithm) have depth 0, its children depth 1, and so on.

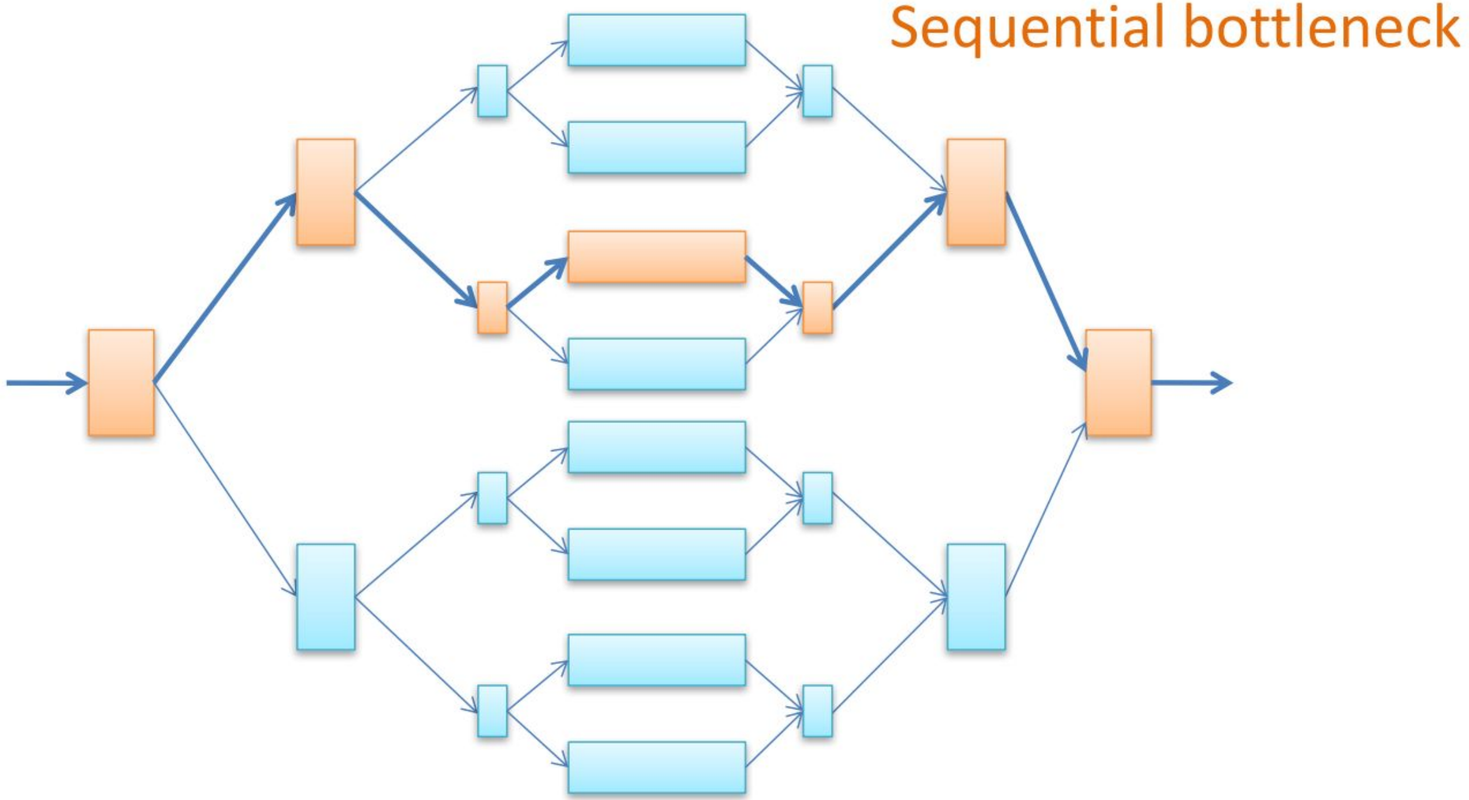
Different DAG layers cannot be computed in parallel

- Without loss of generality, assume our DAG is a tree, so the levels are well defined.
- Let the root of the tree (i.e. the output of the algorithm) have depth 0, its children depth 1, and so on.
- Suppose m_i denotes the number of operations (or nodes) performed in level i of the DAG.
- Each of the m_i operations may be computed concurrently, i.e. no computation depends on another in the same layer.

Different DAG layers cannot be computed in parallel

- Without loss of generality, assume our DAG is a tree, so the levels are well defined.
- Let the root of the tree (i.e. the output of the algorithm) have depth 0, its children depth 1, and so on.
- Suppose m_i denotes the number of operations (or nodes) performed in level i of the DAG.
- Each of the m_i operations may be computed concurrently, i.e. no computation depends on another in the same layer.
- But operations in different levels of the DAG may not be computed in parallel.
- For any node, the computation cannot begin until all its children have finished their computations.

Critical path



How an algorithm is executed with an unlimited number of processors?

How an algorithm is executed with an unlimited number of processors?

- At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time.
- We may then pass on the results to the next level, use as many processors as required to compute all results in parallel in constant time again, and repeat for each layer in the DAG.

How an algorithm is executed with an unlimited number of processors?

- At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time.
- We may then pass on the results to the next level, use as many processors as required to compute all results in parallel in constant time again, and repeat for each layer in the DAG.
- So with infinite processors, the compute time is given by the depth of the tree.
- We then define depth to be

$$T\infty = \text{depth of computation DAG.}$$

How an algorithm is executed with an unlimited number of processors?

- At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time.
- We may then pass on the results to the next level, use as many processors as required to compute all results in parallel in constant time again, and repeat for each layer in the DAG.
- So with infinite processors, the compute time is given by the depth of the tree.
- We then define depth to be

$$T_\infty = \text{depth of computation DAG.}$$

- Realistically, the number of processors will be limited, so what's the point of T_∞ i.e. time assuming infinite resources? It will be used in upper-bounding T_p.

Brent’s theorem

Brent’s theorem

- With T1, Tp, T∞ defined as above,

$$\frac{T_1}{p} \le T_p \le \frac{T_1}{p} + T_\infty.$$

- Since T1/p optimal, we see that T∞ allows us to assess how far off our algorithm performs relative to the best possible version of the parallel algorithm.
- T∞ can be interpreted as how parallel an algorithm is, smaller values indicate more parallelism.

Proof

Proof

On level i of our DAG, there are m_i operations. Since T1 is the total work of our algorithm, if we denote T∞ = n

$$T_1 = \sum_{i=1}^{n} m_i$$

Proof

On level i of our DAG, there are m_i operations. Since T1 is the total work of our algorithm, if we denote T∞ = n $$T_1 = \sum_{i=1}^{n} m_i$$

For each level i of the DAG, the time taken by p processors is given as $$T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1.$$

The equality follows from the fact that there are m_i constant-time operations to be performed at m_i. Once all lower levels are completed, these operations share no inter-dependencies. So, we may distribute operations uniformly to our processors.

The ceiling follows from the fact that if the number of operations is not divisible by p, we require one wall-clock cycle where some but not all processors are used in parallel.

Proof

On level i of our DAG, there are m_i operations. Since T1 is the total work of our algorithm, if we denote T∞ = n $T_1 = \sum_{i=1}^{n} m_i$

For each level i of the DAG, the time taken by p processors is given as $T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1.$

The equality follows from the fact that there are m_i constant-time operations to be performed at m_i. Once all lower levels are completed, these operations share no inter-dependencies. So, we may distribute operations uniformly to our processors.

The ceiling follows from the fact that if the number of operations is not divisible by p, we require one wall-clock cycle where some but not all processors are used in parallel.

Then,

$$T_p = \sum_{i=1}^{n} T_p^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right) = \frac{T_1}{p} + T_\infty$$

Using Brent’s theorem: summation algorithm 1

Using Brent’s theorem: summation algorithm 1

How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form

```
$s \leftarrow 0$ for $i \leftarrow 1, 2, \ldots, n$ do
    $s +=$ a[i]
end
return $s$
```

Using Brent’s theorem: summation algorithm 1

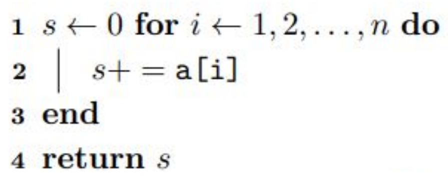
How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form

```
$s \leftarrow 0$ for $i \leftarrow 1, 2, \ldots, n$ do
    $s += $ a[i]
end
return $s$
```

For this summation algorithm T1 = n. Work of the algorithm is O(n).

Using Brent’s theorem: summation algorithm 1

How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form



```
$s \leftarrow 0$ for $i \leftarrow 1, 2, \ldots, n$ do
    $s += $ a[i]
end
return $s$
```

For this summation algorithm T1 = n. Work of the algorithm is O(n).

What’s T2 on this algorithm? T2 = n as well, since we haven’t written the code in a way which is parallel.

Using Brent’s theorem: summation algorithm 1

How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form

```
$s \leftarrow 0$ for $i \leftarrow 1, 2, \ldots, n$ do
    $s += $ a[i]
end
return $s$
```

For this summation algorithm T1 = n. Work of the algorithm is O(n).

What’s T2 on this algorithm? T2 = n as well, since we haven’t written the code in a way which is parallel.

The depth of the algorithm is O(n), since we have written it in a sequential order. In fact, T∞ = n. So, if we increase the number of processors but keep the same algorithm, the time of algorithm does not change, i.e. T1 = T2 = · · · = T∞ = n.

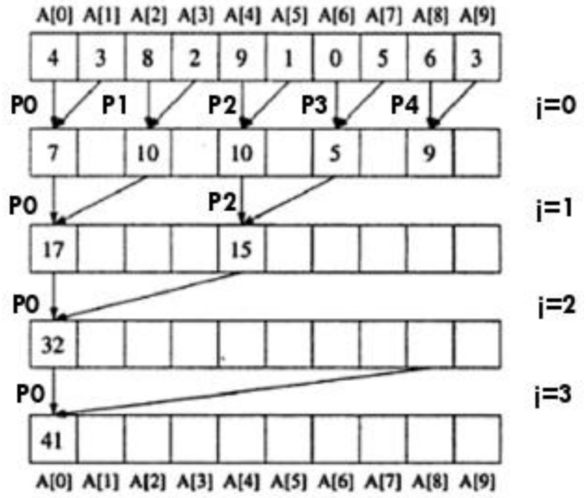
Using Brent’s theorem: summation algorithm 2

GLOBAL ARRAY BASED EXECUTION

The processors in a PRAM algorithm manipulate data stored in global registers.

For adding n numbers we spawn $\lfloor\left(\frac{n}{2}\right)\rfloor$ processors.

Consider the example to generalize the algorithm.


A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
4 3 8 2 9 1 0 5 6 3
P0 P1 P2 P3 P4
j=0
7 10 10 5 9
P0 P2
j=1
17 15
P0
j=2
32
P0
j=3
41
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]

GLOBAL ARRAY BASED EXECUTION

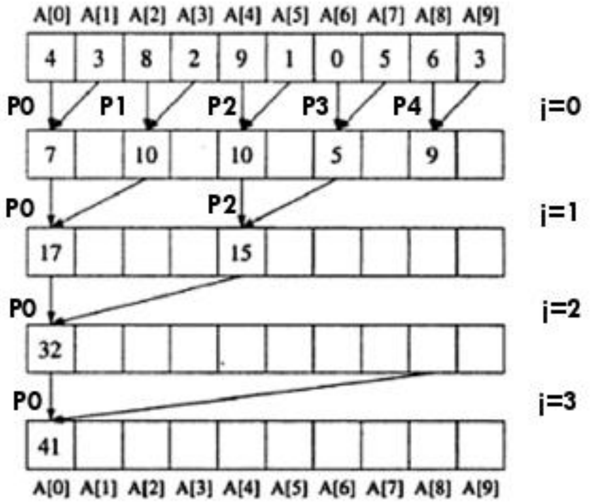
Each addition corresponds to:

A[2i]+A[2i+2^j].

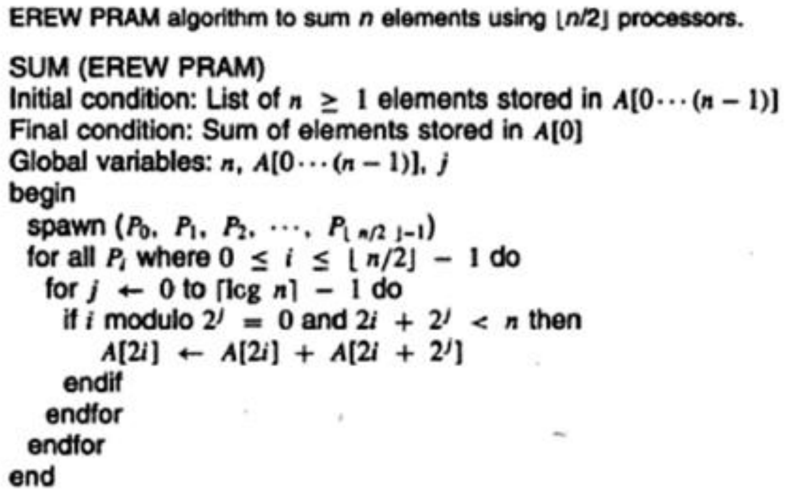
Note, the processor which is active has an i such that:
i mod 2^j=0 (ie. keep only those processors active).

Also check that the array does not go out of bound.

- ie, $2i+2^j<n$


A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
4 3 8 2 9 1 0 5 6 3
P0 P1 P2 P3 P4
j=0
7 10 10 5 9
P0 P2
j=1
17 15
P0
j=2
32
P0
j=3
41
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]

EREW PRAM PROGRAM



EREW PRAM algorithm to sum n elements using $\lfloor n/2 \rfloor$ processors.

```
SUM (EREW PRAM)
Initial condition: List of $n \geq 1$ elements stored in $A[0 \cdots (n-1)]$
Final condition: Sum of elements stored in $A[0]$
Global variables: $n$, $A[0 \cdots (n-1)]$, $j$
begin
  spawn ($P_0$, $P_1$, $P_2$, $\cdots$, $P_{\lfloor n/2 \rfloor - 1}$)
  for all $P_i$ where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do
    for $j \leftarrow 0$ to $\lceil \lg n \rceil - 1$ do
      if $i$ modulo $2^j = 0$ and $2i + 2^j < n$ then
          $A[2i] \leftarrow A[2i] + A[2i + 2^j]$
      endif
    endfor
  endfor
end
```

Using Brent’s theorem: summation algorithm 2

Instead of (a1 + a2) + a3 + a4 + . . . we assign each processor a pair of elements from our array, such that the union of the pairs is the array and there is no overlap.

At the next level of our DAG, each of the summations from the leaf-nodes will be added by assigning each pair a processor in a similar manner.

If the array length is not even, we can simply pad it with a single zero.

Using Brent’s theorem: summation algorithm 2

Instead of (a1 + a2) + a3 + a4 + . . . we assign each processor a pair of elements from our array, such that the union of the pairs is the array and there is no overlap.

At the next level of our DAG, each of the summations from the leaf-nodes will be added by assigning each pair a processor in a similar manner.

If the array length is not even, we can simply pad it with a single zero.

This results in an algorithm with depth T∞ = log2 n. Hence by Brent’s theorem,

$$T_p \leq \frac{n}{p} + \log_2 n.$$

As n → ∞, our algorithm does better since n/p dominates. Remember Gustafson?

As p → ∞, our algorithm does worse, since all that remains is log2 n

COMPLEXITY

The SPAWN routine requires $\lceil log \left\lfloor \frac{n}{2} \right\rfloor \rceil$ doubling steps.

The sequential for loop executes $\lceil \log n \rceil$ times.

- Each iteration takes constant time.

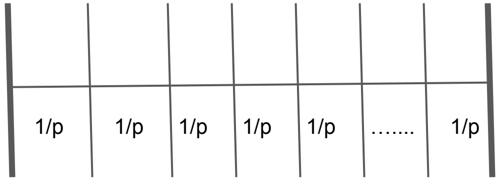
Hence overall time complexity is $\Theta(\log n)$ given n/2 processors.

$$T_p \leq \frac{n}{p} + \log_2 n.$$

Brent’s theorem tighter bound

On level i of our DAG, there are m_i operations. Since T1 is the total work of our algorithm, if we denote T∞ = n $T_1 = \sum_{i=1}^{n} m_i$

For each level i of the DAG, the time taken by p processors is given as $T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \le \frac{m_i}{p} + 1.$



So a tighter bound will be $T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \le \frac{m_i}{p} + 1.$-1/p

Brent’s theorem with tighter bound $T_p = \sum_{i=1}^{n} T_p^i \le \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right)$ -1/p $= \frac{T_1}{p} + T_\infty$ - T∞/p

Compare runtime on two machines with finite processors

Assume a parallel computer where each processor can perform an operation in unit time.

Further, assume that the computer has exactly enough processors to exploit the maximum concurrency in an algorithm with M operations, such that T time steps suffice.

Brent's Theorem say that a similar computer with fewer processes, P, can perform the algorithm in time, $T_P \leq T + (M - T)/P$

$$T_p = \sum_{i=1}^{n} T_p^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right) - 1/p \quad = \frac{T_1}{p} + T_\infty \ - T\infty/p$$

Finite processors for parallel reduction

We know of a solution with large number of processors, which takes Θ(logn) time.

Let us reduce the number of processors to $\lfloor(\frac{n}{\log n)\rceil}\rfloor$ processors.

Thus,

$$T_p \leq \lceil \log n \rceil + \frac{(n-1) - \lceil \log n \rceil}{\left\lfloor \frac{n}{logn} \right\rfloor} = \Theta\left(logn + logn - \frac{logn}{n} - \frac{\log^2 n}{n}\right)$$

$$= \Theta(\text{logn})$$

Thus reducing the number of processors from n to $\left\lfloor \frac{n}{logn} \right\rfloor$ does not change the complexity of the parallel algorithm.

Summary

- Parallel algorithms can be represented as DAG, for better runtime analysis
- Nodes are computations, edges indicate dependencies between computations

Summary

- Parallel algorithms can be represented as DAG, for better runtime analysis
- Nodes are computations, edges indicate dependencies between computations
- Number of nodes indicate computation time on single machine T1
- Runtime on parallel machine cannot be better than T1/p, where p indicates number of processors. This is the **parallel runtime lower bound**.

Summary

- Parallel algorithms can be represented as DAG, for better runtime analysis
- Nodes are computations, edges indicate dependencies between computations
- Number of nodes indicate computation time on single machine T1
- Runtime on parallel machine cannot be better than T1/p, where p indicates number of processors. This is the **parallel runtime lower bound**.
- Nodes at same depth can be computed simultaneously by parallel machines
- So total depth of DAG indicate level of parallelism in the program, smaller depth means more parallel.

Summary

- Parallel algorithms can be represented as DAG, for better runtime analysis
- Nodes are computations, edges indicate dependencies between computations
- Number of nodes indicate computation time on single machine T1
- Runtime on parallel machine cannot be better than T1/p, where p indicates number of processors. This is the **parallel runtime lower bound**.
- Nodes at same depth can be computed simultaneously by parallel machines
- So total depth of DAG indicate level of parallelism in the program, smaller depth means more parallel.
- Maximum depth path (longest chain of sequential dependency) is the critical path.
- Even with infinite processors, critical path time (called T_∞) cannot be reduced.
- Brent’s law fixes **parallel runtime upper bound** at $T1/P + T_\infty$ or $T1/P + T_\infty - T_\infty/p$