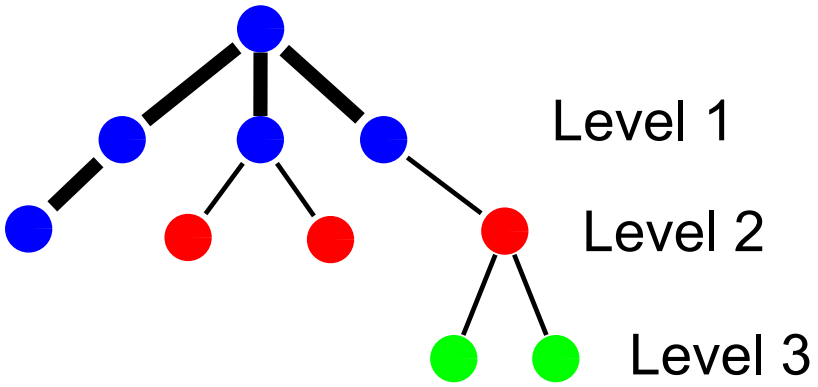
15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: General Techniques **Date:** September 6, 2007
Scribe: Charlie Won

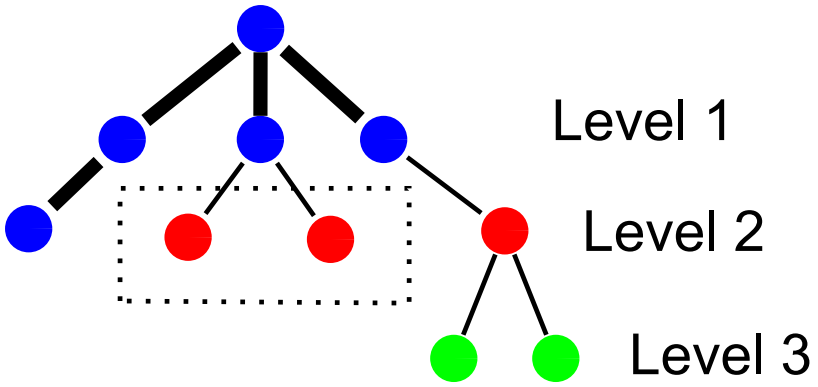
4.1 Greedy Schedule Theorem

In a nutshell, a *greedy scheduler* is a scheduler in which no processor is idle if there is more work it can do. A breadth first schedule can be shown to be bounded by the constraints of $\max(\frac{W}{P}, D) \leq T < \frac{W}{P} + D$, where W is the total work, P is the number of processors, and D is the depth.

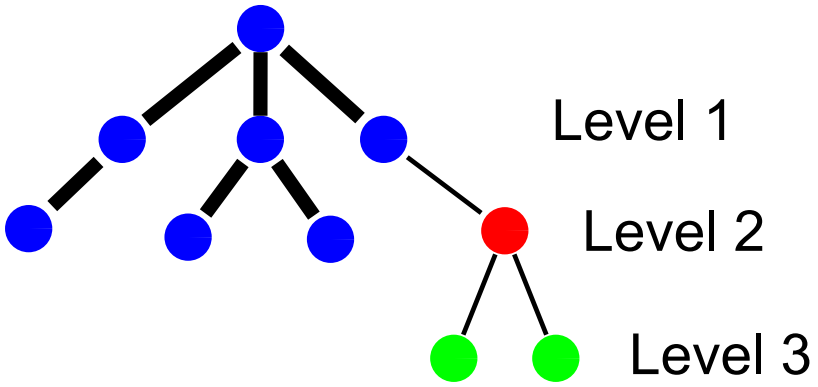
Let the following diagram illustrate a breadth-first tree of the work dependence graph. Blue vertices symbolize completed computations. Red vertices have their dependencies satisfied, and are ready to be completed. Green nodes need their dependencies to be satisfied before being computed. This diagram requires $W = 10$ computations. Suppose that there are $P = 2$ processors.



The next step would involve the following two computations:



Since there are more available computations than processors, no work is wasted. After the computation is completed, there is only 1 computation left with dependencies satisfied. Time will be wasted on the following step since not every processor would be busy.



A BFS clearly must have a T lower bound of $\frac{W}{P}$ time since this is the time it would take if steps are never 'wasted'. D must also be a lower bound since this is the minimum amount of time to reach the furthest node on the longest path.

For the upper bound - the diagram illustrates how time can only be wasted on the completion of a level. Since there are D levels, and $\frac{W}{P}$ time is required if no time is wasted, the upper bound may be established at $\frac{W}{P} + D$

Moreover, this bound can be extended to all greedy schedules.

Theorem 4.1.1 *Any greedy schedule of a DAG with W nodes and longest path (i.e. depth) D would take T steps, where* $\max(\frac{W}{P}, D) \leq T < \frac{W}{P} + D$

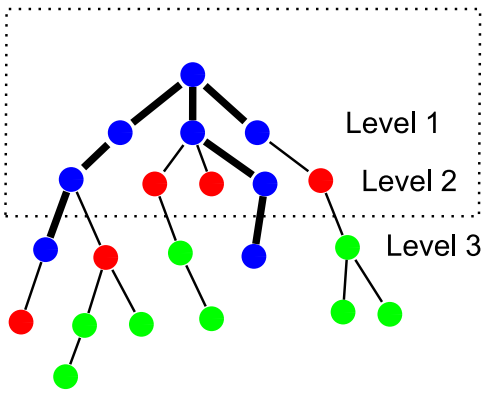
Significance: This theorem proves a pretty tight bound with a maximum range of a factor of 2. However, it's **NP**-complete to find the optimum schedule.

Proof: The lower bounds can be trivially shown since $W = T \cdot P$ is the maximum work that can be done in T time, and it takes D steps to traverse the longest path.

For the upper bound: let there be a greedy schedule that has completed t steps, has completely finished L levels away from the start node, and has potentially completed parts of levels $> L$.

To illustrate, consider Figure 4.1. Level 1 has been completed, but there are also parts of levels 2 and 3 that have also been completed.

Since the first L levels have been completed, the dependencies for level $L + 1$ are guaranteed to be satisfied. (There are therefore only blue and red computations on level 2.)



It can be seen that the first $L+1$ levels resemble the progress of a BFS. Since the schedule is greedy, it can only run out of work in similar ways to the BFS. Work can only be wasted on completion of a level or else there will remain computations with their dependencies satisfied.

More formally, time can only be wasted on step $t+1$ if level $L+1$ is completed during that step. If $L+1$ is not completed during the step, time was not wasted since greedy algorithms do any possible available work when processors are free. Since there are at most D levels, at most D steps can be wasted. ■

However, practical issues can occur which make some greedy schedule algorithms infeasible or impractical.

Example: 'Conventional' Matrix Multiplication on an n by n matrix starts with a first level of n numbers. Each of these n branches have n more values branching off. then branching off *another* n branches off each of these second level numbers. This means that n^3 values must be stored in memory, which can be a rather large figure for large n.

4.2 Algorithmic Techniques

There are a variety of ideas used when it comes to parallel algorithms.

4.2.1 Divide and Conquer

This idea simply involves dividing a problem into subproblems and recursing on those smaller subproblems in parallel.
Examples: merge, binary search and pivot, recurse

Or as in the homework problem, D&C may be used to merge 2 sorted arrays, which we'll call A and B.

One method involves first finding the overall median using the traditional sequential D&C method in $O(\log(n))$ time.

Now recurse on the parts of A and B less than the median, and the parts of A and B greater than the median. The results of the second recursion may be appended to the end of the first recursion.

Overall, this takes $O(n)$ work and $O(\log^2(n))$ depth.

4.2.2 Partitioning

The partitioning strategy loosely refers to techniques that involve breaking up the problem into smaller, independent subproblems of almost equal sizes, and solving the little pieces simultaneously. As an example, we will sketch how to apply the partitioning technique to the problem of merging two sorted arrays.

A partitioning method to the problem may be performed by dividing an array into $\frac{n}{\log(n)}$ parts of size $\log(n)$. Let's define the first element of each partition to be the head. A binary search may then be performed on a second array to locate where the heads should be inserted. After the heads have been inserted, the rest of the elements may be added in sequentially. Merging that now needs to be done is tricky and was not gone over.

4.2.3 Pointer Jumping

This technique involves the manipulation of pointers in parallel. One may use this technique to find the tail of a linked list when given:

- pointers to each node
- the 'head' of the desired linked list

Let

$$\text{A} \longrightarrow \text{B} \longrightarrow \text{C} \longrightarrow \text{D} \longrightarrow \text{E} \longrightarrow \text{F}$$

symbolize the linked list.

These pointers are not necessarily sequential or adjacent in memory, and there may be multiple linked lists in memory. F's 'next' pointer points to itself. The idea of this algorithm is to update every node's 'next' pointer to '→next→next' after every step.

On the first step, $A \to B$ becomes $A \to C$, $B \to C$ becomes $B \to D$, and $C \to D$ becomes $C \to E$.

$D \to F$, $E \to F$, and $F \to F$ remain the same.

Therefore... the algorithm would proceed as follows:

A $\longrightarrow$ C $\longrightarrow$ E $\longrightarrow$ F

A $\longrightarrow$ E $\longrightarrow$ F

A $\longrightarrow$ F

This takes at most $\log(n)$ time where the linked list has length n. However, this is not work efficient. It has work $O(n \cdot \log(n))$. A temporary variable may be used in each node to store these pointer changes so that the actual link structure isn't modified.

4.2.4 Contraction

This technique can be used to solve the Prefix Sum Problem. For each element in an array, one wants to sum all previous elements.

Array Contents	a	b	c	d	e	f
Prefix Sum	i	a	a+b	a+b+c	a+b+c+d	a+b+c+d+e

where i represents the operator's identity.

Contraction combines adjacent elements and solves this recursively - first producing:

a+b	c+d	e+f

In the next step, $a + b + c + d$ would be computed by combining indices 0 and 1. $a + b + c$ would be computed by adding c to $a + b$.

Finally e would be added to $a + b + c + d$ for the final values.

The technique shrinks the problem to $log(n)$ depth.

The remaining (odd-indexed) computations at each step can be done in constant time with linear work.

Overall:
$W(n) = W(\frac{n}{2}) + O(n)$
$D(n) = D(\frac{n}{2}) + O(1)$

4.2.5 Symmetry breaking

With the following linked list:

$$H \longrightarrow G \longrightarrow F \longrightarrow E \longrightarrow D \longrightarrow C \longrightarrow B \longrightarrow A$$

Can partial sums be generated with pointer jumping and randomization? This technique is needed since you may not know your index in common data structures such as linked list. Looking at your predecessor and successor doesn't help.