1 Parallel and Distributed Computation

Parallel computation: Tightly coupled processors that can communicate almost as quickly as perform a computation

Distributed computation: Loosely couple processor for which communication is much slower than computation

2 PRAM Model

A *PRAM* machine consists of m synchronous processors with shared memory. This model ignores synchronization problems and communication issues, and concentrates on the task of parallelization of the problem. One gets various variations of this model depending on how various processors are permitted to access the same memory location at the same time.

- ER= Exclusive Read, only one processor can read a location in any 1 step

- CR= Concurrent Read, any number of processors can read a location in a step

- EW= Exclusive write, only one processor can write a location in any 1 step

- CW= Concurrent Write, any number of processors can write a location in a step. What it 2 processors try to write different values?

  - Common: All processors must be trying to write the same value
  - Arbitrary: An arbitrary processor succeeds in the case of a write conflict
  - Priority: The lowest number processor succeeds

The "right" model is probably an EREW PRAM, but we will study other models as academic exercises. We will sometimes refer to algorithms by the type of model that these algorithms are designed for, e.g. an EREW PRAM algorithm.

We define $T(n, p)$ to be parallel time for the algorithm under consideration with p processors on input of size n Let $S(n)$ be the sequential time complexity. Then the *efficiency* of a parallel algorithm is defined by

$$E(n, p) = \frac{S(n)}{mT(n, p)}$$

Efficiencies can range from 0 to 1. The best possible efficiency you can have is 1. Generally we prefer algorithm whose efficiency is $\Omega(1)$.

The *folding principle* can be stated in two equivalent ways:

Time Based Definition: For $k \geq 1$ it must be the case that $T(n, p) \leq kT(n, kp)$. That is, increasing the number of processors by a factor of k reduces the time by at most a factor of k. Or equivalently, reducing the number of processors by a factor of k increases time by at most a factor of k.

Efficiency Based Definition: For $k \geq 1$, $E(n, p) \geq T(n, kp)$. That is, more processors can not increase efficiency, and there is no loss of efficiency if you decrease the number of processors.

3   The OR Problem

INPUT: bits $b_1, \ldots b_n$
OUTPUT: The logical OR of the bits

One can obtain an EREW Algorithm with $T(n, p) = n/p + \log p$ using a divide and conquer algorithm that is perhaps best understood as a binary tree. The p leaves of the binary tree are n/p bits. Each internal node is a processor that OR's the output of its children. The efficiency of the EREW Algorithm is

$$E(n, p) = \frac{S(n)}{pT(n, p)} = \frac{n}{p(n/p + \log p)} = \frac{n}{p + p \log p}$$

which is $\Omega(1)$ if $p = O(n/\log n)$.

One can also obtain a CRCW Common Algorithm with $T(n, n) = \Theta(1)$. In this algorithm each processor P_i sets a variable answer to 0, then if $b_i = 1$, P_i sets answer to 1. The efficiency of this algorithm is $E(n, n) = \Theta(1)$.

4   MIN Problem

See section 10.2.1, and section 10.2.2.
INPUT: Integers $x_1, \ldots, x_n$
OUTPUT: The smallest x_i

The results are essentially the same as for the OR problem. There is an EREW divide and conquer algorithm with $E(n, n/\log n) = \Theta(1)$. Note that this technique works for any associative operator (both OR and MIN are associative).

There is an CRCW Common algorithm with $T(n, p = n^2) = 1$ and $E(n, p = n^2) = 1/n$. Here's code for processor $P_{i,j}$, $1 \leq i, j \leq j$ for the CRCW Common algorithm to compute the location of the minimum number:

```
If x[i] <= x[j] then T[i,j] = 1 else T[i, j]=0;
And[i]=1;
If T[i, j] = 0 then And[i]=0;
If And[i]=1 then Answer = i
```

What happens when the above code is run in the various CW models if there are two smallest numbers?

What happens in the various CW models if there are two smallest numbers and you just want to compute the value of the smallest number, that is if the last line is changed to

```
If And[i]=1 then Answer = x[i]
```

5   Parallel Prefix Problem

INPUT: Integers $x_1, \ldots, x_n$ OUTPUT: $S_1, \ldots, S_n$

Where $S_i = \sum_{j=1}^{j} x_j$. We give a divide and conquer algorithm. Solve the problem for the even x_i's and odd x_i's separately Then

$$S_{2i} = (x_1 + x_3 + \ldots + x_{2i-1}) + (x_2 + x_4 + \ldots + x_{2i})$$

and

$$S_{2i-1} = (x_1 + x_3 + \ldots + x_{2i-1}) + (x_2 + x_4 + \ldots + x_{2i-2})$$

This gives an algorithm with time $T(n,n) = \log n$ on EREW PRAM. This can be improved to $T(n, n/\log n) = \log n$, thus giving $\Theta(1)$ efficiency.

Note that divide and conquer into the first half and last half is more difficult because of the sum for the first half becomes a bottleneck that all of the last half of the processors want to access. Also note that this technique works for any associative operator.

6 All Pairs Shortest Path Problem

INPUT: A directed edge weighted graph G, with no negative cycles.
OUTPUT: The length of the shortest path $D[i,j]$ between any pair of points i and j.

First consider the following sequential code:

```
For i = 1 to n do
   For j=1 to n do
     D[i,j]=weight of edge (i, j)

Repeat log n times
   For i = 1 to n do
      For j=1 to n do
         For m=1 to n do
            D[i,j]=min{D[i,j], D[i,m]+D[m,j]}
```

The correctness of this procedure can be seen using the following loop invariant: After t times through the repeat loop, for all i and j, if the length of the shortest path between vertices i and j that has 2^t or less edges is equal to $D[i,j]$.

A "parallel for loop" is a loop where all operations can be executed in parallel, for example:

```
For i = 1 to n do C[i]=A[i] + B[i]
```

Question: So which loops can be replaced by parallel for loops? Answer: The second and third. This gives $T(n, p = n^2) = n \log n$ on a CREW PRAM. The fourth loop could be replaced by a parallel for loop on a machine with concurrent write machine that always writes the smallest value. But note

that the fourth loop is just computing a minimum, which is an associative operator. Thus using the standard algorithm the compute the value of an associative operator in time $\log n$ with $n/\log n$ processors, we get time $T(n, n^3/\log n) = \log n$ on an CREW PRAM.

Question: What is the efficiency? Answer: It depends what you use for $S(n)$:

- $S(n) = T(n, 1)$. This measures speed-up of the parallel algorithm, but doesn't give speed up over standard sequential algorithm.

- $S(n) = $ best achievable sequential time. But for almost all problems, the best achievable sequential time is not known.

- $S(n) = $ sequential time of "standard" or "best known" sequential algorithm. But this has the odd property that the efficiency of a parallel algorithm can change when a new sequential algorithm is discovered.

Note that there are simple sequential shortest path algorithms that run in time $O(n^3)$, and complicated ones that run in time something like $O(n^{2.4})$.

7 Odd-Even Merging

See section 10.2.1.
INPUT: Sorted lists $x_1, \ldots, x_n$, and $y_1, \ldots, y_n$.
OUTPUT: The two lists merged into one sorted list $z_1, \ldots, z_{2n}$

We give the following divide and conquer algorithm


```
Merge(x_1, ..., x_n, y_1 ... y_n)

Merge(x_1, x_3, x_5, y_2, y_4, y_6 ...) to get a_1 ... a_n
Merge(x_2, x_4, x_6, y_1, y_3, y_5 ...) to get b_1 ... b_n
for i=1 to n do z_2i-1=min(a_i, b_i)
                z_2i  =max(a_i, b_i)
```


This can be implemented on an EREW PRAM to run in time $T(n, n) = \log n$ thus giving efficiency $\Theta(1/\log n)$. The following argument establishes

5

the correctness of the algorithm. Each a_i is greater than or equal $a_1, \ldots, a_i$. Each a_i, $i > 1$ is larger than b_{i-1}. Hence, $a_i \geq z_{2i-1}$. Each b_i is greater than or equal $a_1, \ldots, b_i$. Each b_i, $i > 1$, is larger than a_{i-1}. Hence, $b_i \geq z_{2i-1}$. This same argument shows $a_{i+1} \geq z_{2i+1}$ and $b_{i+1} \geq z_{2i+1}$. So z_{2i-1} and z_{2i} must be a_i and b_i.

8   Odd-Even Merge Sorting

See section 10.2.1. We give the following divide and conquer algorithm

```
Sort(x_1, ... x_n,)

Merge(Sort(x_1, ... x_n/2), Sort(x_n/2, ... x_n))
```

This can be implemented on an EREW PRAM to run in time $T(n,n) = \log^2 n$ thus giving efficiency $\frac{n \log n}{n \log^2 n} = \Theta(1/\log n)$.

There is a EREW sorting algorithm with constant efficiency, but it is a bit complicated.

9   Pointer Doubling Problem

In the pointer doubling problem each of the n processors is given a pointer to a unique element of a singly linked list of n items. The goal is for each processor to learn its location in the linked list, e.g. the the processor with the 17th element in the list should know that it is 17th in the list.

```
for i= 1 to n-1  in parallel do
      d[i]=1

d[n]=0

repeat log n times
   for each item i in parallel
```

```
    if next[i] != nil then
        d[i]=d[i] + d[next[i]]
        next[i]=next[next[i]]
```

The correctness of the code follows from the following loop invariant: The position of i equals `d[i] + d[next[i]] + d[next[next[i]]] + ...`

Note that this is essentially solving the parallel prefix problem, with the work done before the recursion instead of after. To solve the parallel prefix problem we would replace the initialization

```
for i= 1 to n-1  in parallel do
     d[i]=1
```
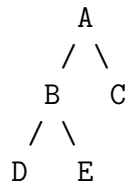
by

```
for i= 1 to n-1  in parallel do
     d[i]=x[i]
```

10   Eulerian Tour Technique to Find Tree Depths

The input is a binary tree with one processor per node. Assume that each processor knows the location of one node. The problem is to compute the depth of each node in the tree. We show by example how to reduce this to pointer doubling. From the following tree

```
        A
       / \
      B   C
     / \
    D   E
```

We create the list (the first line)

```
 1, 1, -1, 1, -1, -1, 1, -1
 A B  D  B  E  B  A  C  A
```

and call pointer doubling with d[i] initialized to either 1 or -1 appropriately. The depth of a node is then computed by looking at the sum up to the point shown in the second line.

11 Expression Evaluation

The input is an algebraic expression in the form of a binary tree, with the leaves being the elements, and the internal nodes being the algebraic operations. The goal is to compute the value of the expression. Some obvious approaches won't work are: 1) Evaluate nodes when both values of children are known, and 2) parallel prefix. The first approach won't give you a speed up if the tree is unbalanced. The second approach won't work if the operators are not be associative.
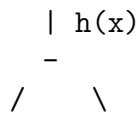
First assume that the only operation is subtraction. We label edges by functions. We now define the cut operation. If we have a subtree that looks like
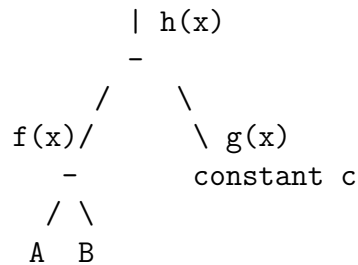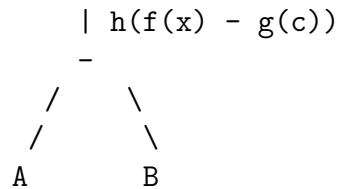
```
        | h(x)
        -
     /     \
  f(x)/      \ g(x)
    -         constant c
   / \
  A   B
```

and cut on the root of this subtree we get

```
        | h(f(x) - g(c))
        -
     /     \
    /       \
   A         B
```

If we have a subtree that looks like

```
        | h(x)
        -
     /     \
```

```
   f(x)/      \ g(x)
 constant c  -
           / \
          A  B
```

and cut on the root of this subtree we get

```
        | h(f(c) - g(x))
        -
      /   \
     /     \
   A        B
```

Thus we are left with finding a class of functions, with the base elements being constants, that are closed under composition, subtraction of constants, and subtraction from constants. This class is the functions of the form $ax + b$, a is $+1$ or -1 and b can be any number.

Note that in one step we can apply cuts to all nodes with an odd numbered left child that is a leaf. Note that in one step we can apply cuts to all nodes with an odd numbered right child that is a leaf. This leads to the following algorithm:

```
Repeat log n times
   For each internal node v in parallel
      if v has odd numbered left child that is a leaf then cut at v
      if v has odd numbered right child that is a leaf then cut at v
   renumber the leaves using pointer doubling
```

Note that in $\log n$ steps we will down to a constant sized tree since each iteration of the outer loop reduces the number of leaves by one half. So $T(n, n) = \log^2 n$ since number the left or right leaves can be done in $\log n$ time using the Eulerian tour technique.

12   A Problem that is Hard to Parallelize

No one knows a fast parallel algorithm for the following problem, known as the Circuit Value Problem.

INPUT: A Boolean circuit F consisting of AND and OR, and NOT gates, and assignment of 0/1 values to the input lines of the circuit.
OUTPUT: 1 if the circuit evaluates to be true, and 0 otherwise.

More precisely, no one knows of a parallel algorithm that runs in time $O(log^k n)$ for some k, with a polynomial number of processors. Here n is the size of the circuit. Further this problems is complete for polynomial time sequential algorithms in the sense that if this problem is parallelizable (time $O(log^k n)$ for some k, with a polynomial number of processors) then all problems that have polynomial time sequential algorithms are parallelizable.