The Parallel Random Access Machine

*Part*1

The PRAM Model

The Parallel Random Access Machine (PRAM) is one of the simplest ways to model a parallel computer. A PRAM consists of a collection of (sequential) processors that can *synchronously* access a global *shared* memory in unit time. Each processor can thus access its shared memory as fast (and efficiently) as it can access its own local memory. The main advantages of the PRAM is its simplicity in capturing parallelism and abstracting away communication and synchronization issues related to parallel computing. Processors are considered to be in abundance and unlimited in number. The resulting PRAM algorithms thus exhibit *unlimited parallelism* (number of processors used is a function of problem size). The abstraction thus offered by the PRAM is a fully synchronous collection of processors and a shared memory which makes it popular for parallel algorithm design. It is, however, this abstraction that also makes the PRAM unrealistic from a practical point of view. Full synchronization offered by the PRAM is too expensive and time demanding in parallel machines currently in use. Remote memory (i.e. shared memory) access is considerably more expensive in real machines than local memory access as well and UMA machines with unlimited parallelism are difficult to build.

Depending on how concurrent access to a single memory cell (of the shared memory) is resolved, there are various PRAM variants. ER (Exclusive Read) or EW (Exclusive Write) PRAMs do not allow concurrent access of the shared memory. It is allowed, however, for CR (Concurrent Read) or CW (Concurrent Write) PRAMs. Combining the rules for read and write access there are four PRAM variants: EREW, ERCW, CREW and CRCW PRAMs. Moreover, for CW PRAMs there are various rules that arbitrate how concurrent writes are handled.

Convention: In this subject we name processors arbitrarily either $0, 1, \ldots, p-1$ or $1, 2, \ldots, p$.

Types of PRAMs

(1) in the *arbitrary* PRAM, if multiple processors write into a single shared memory cell, then an arbitrary processor succeeds in writing into this cell,

(2) in the *common* PRAM, processors must write the same value into the shared memory cell,

(3) in the *priority* PRAM the processor with the highest priority (smallest or largest indexed processor) succeeds in writing,

(4) in the *combining* PRAM if more than one processors write into the same memory cell, the result written into it depends on the combining operator. If it is the *sum* operator, the sum of the values is written, if it is the *maximum* operator the maximum is written.

The EREW PRAM is the weakest among the four basic variants. A CREW PRAM can simulate an EREW one. Both can be simulated by the more powerful CRCW PRAM. An algorithm designed for the common PRAM can be executed on a priority or arbitrary PRAM and exhibit similar complexity. The same holds for an arbitrary PRAM algorithm when run on a priority PRAM.

Assumptions

In this handout we examine parallel algorithms on the PRAM. In the course of the presentation of the various algorithms some common assumptions will be made. The input to a particular problem would reside in the cells of the shared memory. We assume, in order to simplify the exposition of our algorithms, that a cell is wide enough (in bits or bytes) to accommodate a single instance of the input (eg. a key or a floating point number). If the input is of size n, the first n cells numbered $0, \ldots, n-1$ store the input. In the discussion below, we assume that the number of processors of the PRAM is n or a polynomial function of the size n of the input. Processor indices are $0, 1, \ldots, n-1$.

Parallel Sum

Problem: Parallel Sum.

Input. $x_0, \ldots, x_{n-1}$

Output. Evaluate $x_0 + \ldots + x_{n-1}$.

A sequential algorithm that solves this problem requires $n-1$ additions. For a PRAM implementation, value x_i is initially stored in shared memory cell $M[i]$. The sum $x_0 + x_1 + \ldots + x_{n-1}$ is to be computed in $T = \lg n$ parallel steps. Without loss of generality, let n be a power of two. If a combining CRCW PRAM with arbitration rule *sum* is used to solve this problem, the resulting algorithm is quite simple. In the first step processor i reads memory cell i storing x_i. In the following step processor i writes the read value into an agreed cell say 0. The time is $T = O(1)$, and processor utilization is $P = O(n)$.

A more interesting algorithm is the one presented below for the EREW PRAM. The algorithm consists of $\lg n$ steps. In step i, processor $j < n/2^i$ reads shared-memory cells $M[2j]$ and $M[2j+1]$ combines (sums) these values and stores the result into memory cell $M[j]$. After $\lg n$ steps the sum resides in cell 0. Algorithm Parallel_Sum has $T = O(\lg n)$, $P = n$ and $W = O(n \lg n)$, $W_2 = O(n)$.

```
//    pid() returns the id of the processor issuing the call.
begin PARALLEL_SUM  ($n$)
1.    $i = 1$ ; $j = pid()$;
2.    while ($j < n/2^i$)
3.        $a = M[2j]$;
4.        $b = M[2j+1]$;
5.        $M[j] = a + b$;
6.        $i = i + 1$;
7.    end
end PARALLEL_SUM
```

An Example

Algorithm Parallel Sum.

```
M[0]     M[1]     M[2]      M[3]     M[4]     M[5]     M[6]     M[7]

 x0       x1       x2       x3       x4       x5       x6       x7     t=0

x0+x1     x2+x3    x4+x5    x6+x7                                     t=1

x0+...+x3 x4+...+x7                                                   t=2

x0+...+x7                                                             t=3
```

Parallel Sum continued

Algorithm PARALLEL_SUM can be easily extended to include the case where n is not a power of two. PARALLEL_SUM is the first instance of a sequential problem that has a trivial sequential but more complex parallel solution. Instead of operator *Sum* other operators like *Multiply*, *Maximum*, *Minimum*, or in general, any associative operator could have been used. As associative operator $\otimes$ is one such that $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.

Exercise 1 *Can you improve* PARALLEL_SUM *so that T remains the same, $P = O(n/\lg n)$, and $W = O(n)$? Explain.*

Exercise 2 *What if i have p processors where $p < n$? (You may assume that n is a multiple of p).*

Exercise 3 *Generalize the Parallel Sum algorithm to any associative operator.*

Parallel Copy: Broadcasting

A word is stored in memory location M[0] of the shared memory. We would like this word to be read by all n processors of a PRAM. On a CREW PRAM this requires one parallel step (processor i concurrently reads cell 0). On an EREW PRAM broadcasting can be performed in $O(\lg n)$ steps. The structure of the algorithm is the reverse of the previous one. In $\lg n$ steps the word is broadcast as follows. In step i each processor with index j less than 2^i reads the contents of cell $M[j]$ and copies it into cell $M[j + 2^i]$. After $\lg n$ steps each processor i reads the message by reading the contents of cell i.

```
begin BROADCAST  ($M$)
1.   $i = 0$ ; $j = pid()$; M[0]=M;
2.   while ($2^i < P$)
3.       if ($j < 2^i$)
5.           $M[j + 2^i] = M[j]$;
6.       $i = i + 1$;
6.   end
7.   Processor $j$ reads $M$ from $M[j]$.
end BROADCAST
```

A CR?W PRAM algorithm that solves the broadcasting problem has performance $P = O(n)$, $T = O(1)$, and $W = O(n)$.

The EREW PRAM algorithm that solves the broadcasting problem has performance $P = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$.

Exercise 4 *Broadcasting on a hypercube and a butterfly (Hint: Base your solution on the* BROADCAST *algorithm).*

Parallel Prefix

Given a set of n values $x_0, x_1, \ldots, x_{n-1}$ and an associative operator, say $+$, the *parallel prefix* problem is to compute the following n results/"sums".

0: x_0,

1: $x_0 + x_1$,

2: $x_0 + x_1 + x_2$,

$\ldots$

$n-1$: $x_0 + x_1 + \ldots + x_{n-1}$.

Parallel prefix is also called *prefix sums* or *scan*. It has many uses in parallel computing such as in load-balancing the work assigned to processors and compacting data structures such as arrays. We shall prove that computing ALL THE SUMS is no more difficult that computing the single sum $x_0 + \ldots x_{n-1}$. An algorithm for parallel prefix on an EREW PRAM would require $\lg n$ phases. In phase i, processor j reads the contents of cells j and $j - 2^i$ (if it exists) combines them and stores the result in cell j.

The EREW PRAM algorithm that solves the parallel prefix problem has performance $P = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$.

A divide and Conquer Approach

```
x0  x1 x2 x3 x4 x5 x6 x7  <<Paralel Prefix "Box" for 8 inputs
 |  |  |  |  |  |  |  |
-----------  ----------
|    1    |  |   2    | <<< 2 PP Boxes for 4 inputs each
-----------  ---------
|   |  |  |\\\  |  |  |
|   |  |  | \\\\|  |  | Take rightmost output of Box 1 and
|   |  |  |  | \\\ |  | combine it with the outputs of Box2
|   |  |  |  |  | \\  |
|   |  |  |  |  | \\ \|
          x0+...+x3   x0+..+x7
       x0+...+x2   x0+...+x6
    x0+x1       x0+...+x5
x0           x0+...+x4
```

Parallel Prefix : Another Algorithm

```
// We write below[1:2] to denote X[1]+X[2]
//                [i:j] to denote X[i]+X[i+1]+...+X[j]
//                [i:i] is X[i]  NOT X[i]+X[i]
//                [1:2][3:4]=[1:2]+[3:4]= (X[1]+X[2])+(X[3]+X[4])=X[1]+X[2]+X[3]+X[4]
// Input :  M[j]= X[j]=[j:j]             for j=1,...,n.
// Output:  M[j]= X[1]+...+X[j] = [1:j]  for j=1,...,n.
 ParallelPrefix(n) // .
1.i=1;             // At this step  M[j]= [j:j]=[j+1-2**(i-1):j]
2.while (i < n ) {
3. j=pid();
4. if (j-2**(i-1) >0 ) {
5.  a=M[j];         // Before this stepM[j]         = [j+1-2**(i-1):j]
6.  b=M[j-2**(i-1)];// Before this stepM[j-2**(i-1)]= [j-2**(i-1)+1-2**(i-1):j-2**(i-
1)]
7.  M[j]=a+b;       // After this step M[j]= M[j]+M[j-2**(i-1)]=[j-2**(i-1)+1-2**(i-1
):j-2**(i-1)]
                    //                                           [j+1-2**(i-1):j] =
                    //                                           [j-2**(i-1)+1-2**(i-1
):j]=
                    //                                           [j+1-2**i:j]
8. }
9. i=i*2;
  }
```

At step 5, memory location $j - 2^{i-1}$ is read provided that $j - 2^{i-1} \geq 1$. This is true for all times $i \leq t_j = \lg(j-1) + 1$. For $i > t_j$ the test of lin e 4 fails and lines 5-8 are not executed.

Parallel Prefix : Another Algorithm

For visualization purposes, the second step is written in two different lines. When we write $x_1 + \ldots + x_5$ we mean $x_1 + x_2 + x_3 + x_4 + x_5$.

```
   x1      x2        x3        x4         x5        x6      x7       x8



1.       x1+x2      x2+x3    x3+x4      x4+x5    x5+x6    x6+x7     x7+x8

2.              x1+(x2+x3)         (x2+x3)+(x4+x5)   (x4+x5)+(x6+x7)
2.                       (x1+x2)+(x3+x4)    (x3+x4)+(x5+x6)   (x5+x6+x7+x8)

3.                                       x1+...+x5         x1+...+x7
3.                                                x1+...+x6          x1+...+x8
Finally
F. x1   x1+x2    x1+...+x3  x1+...+x4  x1+...+x5 x1+...+x6 x1+...+x7 x1+...+x8
```

Parallel Prefix Example 2

For visualization purposes, the second step is written in two different lines. When we write $[1:5]$ we mean $x_1 + x_2 + x_3 + x_4 + x_5$.

```
   We write below [1:2] to denote x1+x2
                  [i:j] to denote xi +  ... + x5
                  [i:i] is xi  NOT xi+xi!
                  [1:2][3:4]=[1:2]+[3:4]= (x1+x2) + (x3+x4) = x1+x2+x3+x4
   A   *  indicates value above remains the same in subsequent steps

0 x1         x2         x3         x4         x5         x6         x7         x8
0 [1:1]      [2:2]      [3:3]      [4:4]      [5:5]      [6:6]      [7:7]      [8:8]
1   *        [1:1][2:2] [2:2][3:3] [3:3][4:4] [4:4][5:5] [5:5][6:6] [6:6][7:7] [7:7][8:8]
1.  *        [1:2]      [2:3]      [3:4]      [4:5]      [5:6]      [6:7]      [7:8]
2.  *          *        [1:1][2:3] [1:2][3:4] [2:3][4:5] [3:4][5:6] [4:5][6:7] [5:6][7:8]
2.  *          *        [1:3]      [1:4]      [2:5]      [3:6]      [4:7]      [5:8]
3.  *          *          *          *        [1:1][2:5] [1:2][3:6] [1:3][4:7] [1:4][5:8]

3.  *          *          *          *        [1:5]      [1:6]      [1:7]      [1:8]
  [1:1]      [1:2]      [1:3]      [1:4]      [1:5]      [1:6]      [1:7]      [1:8]
  x1         x1+x2      x1+x2+x3   x1+...+x4  x1+...+x5  x1+...+x6  x1+...+x7  x1+...+x8
```

Parallel Prefix : Yet another algorithm

Consider the following variation of parallel prefix on n inputs that works on a complete binary tree with n leaves (assume n is a power of two).

```
Action by nodes
 1. Non-leaf  : If it receives l and r from left and right children, computes l + r and
                sends it up and send down to its right child the l.
 2. Root      : Step [1] except nothing is sent up.
 3. Non-leaf  : If it gets p from parent it transmits it to its left/right children.
 4. Leaf      : If it holds l and receives p from its parent it sets l = p + l (this order)
                [note p is the left argument, l is the right one, order matters]
```

```
                         x1+x2+x3+x4+x5+x6+x7+x8
                             / \
                            /   \
                           /     \
                          /       \
                         /         \
                        /           \
                       /             \ \x1+x2+x3+x4
                      /               \
                     /                 \
                    /                   \
                   /                     \
                  /                       \
                 /                         \
                /                           \
           x1+x2+x3+x4                        x5+x6+x7+X8
             / \                             /\
            /   \                           /  \
           /     \            x1+x2+x3+x4  /    \ \x1+x2+x3+x4
          /       \ \x1+x2                /      \ \x5+x6
         /         \                     /        \
        /           \                   /          \
       /             \                 /            \
   x1+x2           x3+x4           x5+x6           x7+x8
    /\              /\      x1+..+x4/\ <x1+x2+x3+x4>/\ \  x1+x2+x3+x4
   /  \      x1+x2 /  \ \x1+x2     /  \       x5+x6/  \ \ x5+x6
  /    \ \x1      /    \ \x3      /    \ \x5      /    \ \x7
 /      \        /      \        /      \        /      \
x1      x2      x3      x4      x5      x6      x7      x8        Originally
      x1+x2            x3+x4           x5+x6            x7+x8     after recving
             x1+.+x3  x1+..+x4 x1+.+x5 x1+.+x6 x5+.+x7  x5+.+x8
             x1+.+x3  x1+..+x4 x1+.+x5 x1+.+x6 x1+.+x7  x1+.+x8
```

Parallel Prefix : A recursive version

The parallel prefix algorithm of the previous page (tree-based) requires about $2\lg n + 1$ parallel steps, $P = n$ processors and work $W = \Theta(n \lg n)$, and $W_2 = \Theta(n)$. One could describe that version due to Ladner and Fischer as follows. By rescheduling the computation and using $P = n/\lg n$ processors, the work can be reduced to linear.

```
begin PPF_RECURSIVE  ($In[0..n-1]$,$Out[0..n-1]$,$p = 0..n-1$)
1.   Out[0] = In[0];
2.   if  $n > 1$      then
3.       $\forall\ i = 0, \ldots, n-1$     dopar
4.           X[i] = In[2i] + In[2i+1];
5.       enddo
6.       Y=PPF_recursive($X[0..n/2-1]$,$Y[0..n/2-1]$,$p = 0..n/2-1$);
7.       $\forall\ i = 0, \ldots, n/2-1$  dopar
8.           Out[2i+1]=Y[i];
9.       enddo
10.      $\forall\ i = 1, \ldots, n/2-1$  dopar
11.          Out[2i]=Y[i-1]+A[2i];
12.      enddo
13.  endif
end PPF_RECURSIVE
```

Parallel Prefix : An iterative version

An iterative version of that algorithm is depicted below.

```
begin PPF_ITERATIVE  ($In[0..n-1]$,$Out[0..n-1]$,$p = 0..n-1$)
1.  for $i = 0, \ldots, n -$ dopar
2.      T[0,i] = In[i];
3.  enddo
4.  for $j = 1, \ldots, \lg n$ do
5.      for $i = 0, \ldots, n/2^j - 1$   dopar
6.          T[j,i] = T[j-1,2i] + T[j-1,2i+1];
7.  enddo
8.  for $j = \lg n, \ldots, 0$ do
9.      for $i = 0$ dopar
10.         V[j,0] = T[j,0]; //Processor 0 executes only
11.     for $odd(i), 0 \leq i \leq n/2^j - 1$ dopar
12.         V[j,i] = V[j+1,i/2]; //Processor odd(i) executes only
11.     for $even(i), 2 \leq i \leq n/2^j - 1$ dopar
12.         V[j,i] = V[j+1,(i-1)/2]+T[j,i]; //Processor even(i) executes only
13. enddo
14. Out[i]=V[0,i];
end PPF_ITERATIVE
```

An application of parallel prefix

Binary number addition

- Add two n-bit binary numbers in $2 \lg n + 1$ steps using an n-leaf c.b.t.

Sequential algorithm requires n steps.

	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
a	0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	0	
b	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	
	s	g	p	p	g	p	s	s	p	p	s	g	p	p	p	s	s
c																	
$a+b$																	

Table 1: Binary addition example

$(a+b)_i = a_i \oplus b_i \oplus c_{i-1}$, where $\oplus = XOR$.

- s: stops a carry bit $(0+0)$
- g: generates a carry bit $(1+1)$
- p: propagates a carry bit $(0+1$ or $1+0)$.

Problem: In order to compute the k-th bit the $k-1$-st carry needs to be computed as well. There exists a non-trivial non-obvious parallel solution.

We shall try for each bit position to find the carry bit required for addition so that all bit positions can be added in parallel. We shall show that carry computation takes $\Theta(\lg n)$ time on a binary tree with a computation that is known to us: parallel prefix.

Observation. The i-th carry bit is one if the leftmost non-p to the right of the i-th bit is a g.

Question. How can we find i-th carry bit?

The previous observation takes the following algorithmic form.

```
Scan for j=i, ... , 0
  if p ignore else
    if g
       carry=1
       exit;
    else
       carry=0
       exit;
```

Such a computation requires $O(n)$ time for $j = n$ (n-th bit). Let the i-th bit position symbol (p, s, g) be denoted by x_i. Then

$c_0 = x_0 = s$

$c_1 = x_0 \otimes x_1$

$c_2 = x_0 \otimes x_1 \otimes x_2$

$c_{16} = x_0 \otimes \ldots \otimes x_{16}$, where

$\otimes$	s	p	g
s	s	s	g
p	s	p	g
g	s	g	g

Algorithm for parallel addition

Step 1. Compute symbol ($\{s, p, g\}$) for i bit in parallel for all i.

Step 2. Perform a parallel prefix computation on the n symbols plus 0-th symbol s in parallel where operator is defined as in previous table.

Step 3. Combine (exclusive OR) the carry bit from bit position $i - 1$ (interpret g as an 1 and an s as a 0) with the exclusive OR of bits in position i to find the i-th bit of the sum.

Steps 1 and 3 require constant time. Step 2, on a complete binary tree on n leaves would require $2 \lg n$ steps.

$T = 1 + 1 + 2 \lg n$. $P = 2n - 1 = O(n)$.

Parallel Prefix : Segmented Parallel Prefix

A segmented prefix (scan) computation consists of a sequence of disjoint prefix computations. Let the x_{ij} below take values from a set X and let $\oplus$ be an associative operator defined on the elements of set X. Then the segmented prefix computation for

$$x_{11}x_{12}\ldots x_{1k_1} \mid x_{21}x_{22}\ldots x_{2k_2} \mid \ldots \mid x_{m1}x_{m2}\ldots x_{mk_m} \mid$$

requires the computation of all

$$p_{ij} = x_{i1} \oplus x_{i2} \oplus \ldots \oplus x_{ij} \;\; \forall\, 1 \le i \le m,\; 1 \le j \le k_i$$

In bried the segment separator $\mid$ terminates one prefix operation and starts another one.
One way to deal with a segmented prefix computation in parallel is to extend $(X, \oplus)$ into $(X', \otimes)$ so that

$$X' = X \cup \{|\} \cup \{|x : x \in X\}$$

i.e. X' has more than twice the elements of X: it has all the elements of X, the segment separator $\mid$ and a new element $|x$ which consists of the segment separator and x. The new operator $\otimes$ is associative if we define it as follows.

$\mid \otimes \mid = \mid$,	$\mid \otimes x = \mid x$,	$\mid \otimes \mid x = \mid x$,
$x \otimes \mid = \mid$,	$\mid x \otimes \mid = \mid$,	$x \otimes y = x \oplus y$
$\mid x \otimes y = \mid (x \oplus y)$	$x \otimes \mid y = \mid y$	$\mid x \otimes \mid y = \mid y$

Now, if the length of the segmented prefix formula is n we can assign n processors to solve the problem with parallel prefix in asymptotically the same time. Note that an element in X' requires for its representation no more than 2 extra bits of the storage size of an element of X. If an $\oplus$ computation takes $O(1)$ time so does an $\otimes$ computation.

Segmented Parallel Prefix: Example and Refinements

Example
$\{2,3\}$ $\{1,7,2\}$ $\{1,3,6\}$.

Create
2 3 |1 7 2|1 3 6.

And result is:
2 5|1 8 10|1 4 10.

We can refine the previous algorithm as follows.

$\oplus$	b
a	$(a \oplus b)$

$\otimes$	b	$\vert b$
a	$(a \oplus b)$	b
$\vert a$	$\vert (a \oplus b)$	b

Matrix Multiplication

A simple algorithm for multiplying two $n \times n$ matrices on a CREW PRAM with time complexity $T = O(\lg n)$ and $P = n^3$ follows. For convenience, processors are indexed as triples (i, j, k), where $i, j, k = 1, \ldots, n$. In the first step processor (i, j, k) concurrently reads a_{ij} and b_{jk} and performs the multiplication $a_{ij}b_{jk}$. In the following steps, for all i, k the results $(i, *, k)$ are combined, using the parallel sum algorithm to form $c_{ik} = \sum_j a_{ij}b_{jk}$. After $\lg n$ steps, the result c_{ik} is thus computed.

The same algorithm also works on the EREW PRAM with the same time and processor complexity. The first step of the CREW algorithm need to be changed only. We avoid concurrency by broadcasting element a_{ij} to processors $(i, j, *)$ using the broadcasting algorithm of the EREW PRAM in $O(\lg n)$ steps. Similarly, b_{jk} is broadcast to processors $(*, j, k)$.

The above algorithm also shows how an n-processor EREW PRAM can simulate an n-processor CREW PRAM with an $O(\lg n)$ slowdown.

```
                                               CREW    EREW
1. aij to all           (i,j,*) procs    O(1)    O(lgn)
   bjk to all           (*,j,k) procs    O(1)    O(lgn)
2. aij*bjk at           (i,j,k) proc     O(1)    O(1)
3. parallel sum aij *bjk (i,*,k) procs    O(lgn)  O(lgn)   n procs participate
             j

4. cik = sum aij*bjk                     O(1)    O(1)
           j


               3        3                 3
 T=O(lgn),P=O(n ) W=O( n  lgn)    W  = O(n )
                                   2
```

Logical AND operation

Problem. Let $X_1 \ldots, X_n$ be binary/boolean values. Find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$.

The sequential problem accepts a $P = 1, T = O(n), W = O(n)$ direct solution.

An EREW PRAM algorithm solution for this problem works the same way as the PARALLEL SUM algorithm and its performance is $P = O(n), T = O(\lg n), W = O(n \lg n)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.

In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the shared memory location i. We can find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$ in constant time on a CRCW PRAM. Processor 1 first writes an 1 in shared memory cell 0. If $X_i = 0$, processor i writes a 0 in memory cell 0. The result X is then stored in this memory cell.

```
begin LOGICAL_AND  ($X_1 \ldots X_n$)
1.    Proc 1    writes an    1   in cell 0.
2.    if  $X_i = 0$ processor $i$ writes 0 into cell 0.
end LOGICAL_AND
```

The result stored in cell 0 is 1 (TRUE) unless a processor writes a 0 in cell 0; then one of the X_i is 0 (FALSE) and the result X should be FALSE, as it is.

Exercise 5 *Give an $O(1)$ CRCW algorithm for LOGICAL OR.*

Maximum finding

Problem. Let $X_1 \ldots, X_N$ be n keys. Find $X = \max\{X_1, X_2, \ldots, X_N\}$.

The sequential problem accepts a $P = 1, T = O(N), W = O(N)$ direct solution.

An EREW PRAM algorithm solution for this problem works the same way as the PARALLEL SUM algorithm and its performance is $P = O(N), T = O(\lg N), W = O(N \lg N), W_2 = O(N)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.

In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the local memory of processor i.

The CRCW PRAM algorithm MAX1 to be presented has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

The second algorithm to be presented in the following pages utilizes what is called a doubly-logarithmic depth tree and achieves $T = O(\mathrm{lglg} N)$, $P = O(N)$ and $W = W_2 = O(N\, \mathrm{lglg} N)$.

The third algorithm is a combination of the EREW PRAM algorithm and the CRCW doubly-logarithmic depth tree-based algorithm and requires $T = O(\mathrm{lglg} N)$, $P = O(N)$ and $W_2 = O(N)$.

Algorithm MAX1

```
**begin** MAX1  $(X_1 \ldots X_N)$
1.    **in proc** $(i,j)$ **if** $X_i \geq X_j$ **then** $x_{ij} = 1$;
2.                                   **else** $x_{ij} = 0$;
3.    $Y_i = x_{i1} \wedge \ldots \wedge x_{in}$ ;
4.    Processor $i$ reads $Y_i$ ;
5.    **if** $Y_i = 1$ processor $i$ writes $i$ into cell 0.
**end** MAX1
```

In the algorithm, we rename processors so that pair (i,j) could refer to processor $j \times n + i$. Variable Y_i is equal to 1 if and only if X_i is the maximum.

The CRCW PRAM algorithm MAX1 has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

Doubly Logarithmic-Depth Tree

In preparation of algorithm MAX2 we introduce a **doubly logarithmic-depth tree.**
Let $N = 2^{2^n}$, for some integer n.
A *doubly logarithmic-depth* tree with N leaves is defined as follows.

(1) The root of the tree at level 0 has $2^{2^{n-1}} = N^{1/2}$ children in level 1.

(2) Each node at level 1 has $2^{2^{n-2}} = N^{1/2^2}$ children in level 2.

(3) Each node of level i has $2^{2^{n-(i+1)}} = N^{1/2^{i+1}}$ children in level $i+1$.

(4) Each node of level $n-1$ (the level before the last) has $2^{2^{n-n}} = N^{1/2^n} = 2$ children in level $n = \text{lglg}N$.

(5) The nodes of level n are the **leaves** of the tree.

Some properties of a *doubly logarithmic-depth* tree are listed below.

(1) The **height** of the tree is $n = \text{lglg}N$.

(2) A node of level i has $2^{2^{n-(i+1)}}$ children in level $i+1$.

(3) The TOTAL number of level i nodes is $2^{2^{n-1}} 2^{2^{n-2}} \ldots 2^{2^{n-i}} = 2^{2^n - 2^{n-i}}$.

(4) The Product $(2^{2^{n-i-1}})^2 \times 2^{2^n - 2^{n-i}}$ is $O(2^{2^n}) = O(N)$.

Algorithm MAX2

Algorithm MAX2 below achieves better work performance than MAX1 $T = O(\mathrm{lglg}N)$, $P = O(N)$, and $W = W_2 = O(N\,\mathrm{lglg}N)$.

Algorithm MAX2 works as follows.

(1) Initially, items X_i are on the N leaves of the tree.

(2) The root will hold the result at the completion of the algorithm.

(3) Processors are assigned to the nodes of the tree in some predetermined fashion.

(4) All nodes of the tree other than the leaves hold an UNDEFINED value in the beginning of the execution.

(5) If a node u at level i holds an UNDEFINED value and its M children hold some intermediate results M^2 processors are assigned to u to find the maximum of M numbers (the partial results of the children of u) using MAX1 in constant time. Node u then holds the computed maximum (and ceases to hold an UNDEFINED value).

```
begin MAX2  ($X_1 \dots X_N$)
0.    The $i$-th cell contains $X_i$; $N = 2^{2^n}$.
1.    for  ($i = n-1; i \geq 0; i--$) do
2         begin
3.            Assign to each node $u$ of level $i$, $(2^{2^{n-i-1}})^2$ processors (i.e. the square of its children in level $i+1$.) ;
4.            Use algorithm MAX1 and these processors to find the maximum
              of the values stored at the children of $u$ and store the result at $u$;
5.        Node $u$ ceases to hold an UNDEFINED.
6.        end
end MAX2
```

Algorithm MAX3

Algorithm MAX2 has $W = W_2 = O(N \operatorname{lglg} N)$. Algorithm MAX3 below has $W_2 = O(N)$. It uses MAX2 and the EREW PRAM algorithm as subroutines.

The EREW algorithm finds the SUM or the MAXIMUM of N numbers by working from the leaves to the root of a binary tree, ie $\lg N$ levels. If we stop the computation after i levels, we have $N/2^i$ partial results, each result being the MAXIMUM of 2^i numbers initially stored in the leavs of the subtree rooted at the partial result.

Max3 first runs the EREW PRAM algorithm for $i = \operatorname{lglglg} N$ levels so that a total of $N/\operatorname{lglg} N$ partial MAXIMA are computed.

Then it applies MAX2 where N in MAX2 is equal to the number of partial results ie $N/\operatorname{lglg} N$.

```
begin MAX3  $(X_1 \ldots X_N)$
0.   The $i$-th cell contains $X_i$;
1    begin
2.   Use the EREW PRAM algorithm on a complete binary tree on $N$ leaves to
     reduce the original problem to computing the maximum of $N/\operatorname{lglg} N$ values
     (ie proceed from the leaves up to the nodes of level $\lg N - \operatorname{lglglg} N$);
3.   Use MAX2 to find the maximum of the $N/\operatorname{lglg} N$ values of Step2;
4.   end
end MAX3
```

Step 2 of algorithm MAX3 requires $T = O(\operatorname{lglglg} N)$, $P = O(N)$ and $O(N)$ work (number of comparisons is at most the number of edges of the tree).

Step 3 requires $P = O(N)$, $T = O(\operatorname{lglg} N)$ and total work $W = W_2 = O(N)$ by the analysis of MAX2.

Question. Is there a $p \leq n$ processor CRCW PRAM algorithm that finds the maximum of N keys faster than MAX2 or MAX3?

A matching Lower bound and Turan's Theorem

Definition. On an undirected graph $G = (V, E)$, an **independent set** is a set of vertices such that no two vertices are connected by an edge.

Definition. On an undirected graph $G = (V, E)$, a **clique** is a set of vertices such any two vertices are connected by an edge.

A clique on n vertices is the "complement" of an independent set on the same n vertices. The complete graph on n vertices is a clique on n vertices by default. A clique of n vertices has $n(n-1)/2$ edges.

Theorem (Turan) Let $G = (V, E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. Graph G has an independent set of size at least $n^2/(2m + n)$.

Another formulation of Turan's Theorem is the following one.

Theorem (Turan): Second version Let $G = (V, E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. If graph G has no p clique then it has at most $(1 - 1/(p-1))n^2/2$ edges.

Proof. If $n \le p - 1$ then G does not have obviously a p-clique and G has at most $n(n-1)/2$ edges. It is obvious that $n(n-1)/2 \le (1 - 1/(p-1))n^2/2$ by elementary calculation.

Thus the interesting case left is $n \ge p$. If graph G has the maximum number of edges but does not have a p-clique it must have a $(p-1)$-clique. This is because otherwise we could add edges to G to create such a $(p-1)$-clique; this would contradict the maximality of edges of G. Call C a $(p-1)$-clique of G. C has $(p-1)(p-2)/2$ edges. Call G' the graph G without C, i.e. $G' = G - C$. Graph G' has m' edges. Let k be the number of edges that go from G' to C. By induction on G' we have that $m' \le (1 - 1/(p-1))(n-p+1)^2/2$. Since G does not have a p-clique every vertex of G' is connected to at most $p - 2$ vertices of C (since if it were connected to all the vertices of C a p-clique would have been formed). Thus $k \le (p-2)(n-p+1)$.

Therefore the number of edges m of G is

$$m \le (p-1)(p-2)/2 + (1 - 1/(p-1))(n-p+1)^2/2 + (p-2)(n-p+1) \le (1 - 1/(p-1))n^2/2.$$

A matching Lower bound and Turan's Theorem

If we solve this inequality for p we get

$$p - 1 \geq \frac{n^2}{n^2 - 2m}$$

Therefore an equivalent formulation is that G has a $p-1$ clique of size at least $\frac{n^2}{n^2-2m}$.

Now take the complementary graph (where an edge becomes a non-edge and a non-edge becomes an edge). The complement of G has $N = n$ vertices and $M = n(n-1)/2 - m$ edges. From the latter we get that $2m = n(n-1) - 2M = n^2 - n - 2M$. A $p-1$ clique in G becomes an independent set in its complement whose size is at least $\frac{n^2}{n^2-2m} = \frac{N^2}{N^2-2m} = \frac{N^2}{N^2-n^2+n+2M} = \frac{N^2}{N+2M}$, noting that $N = n$. This latter bound is the expression in the first version of Turan's theorem.

An easy corollary is that a graph with n/k vertices, $k \geq 1$, and n edges has an independent set of size at least $n/4k^2$.

Some preliminry results

We are going to show some lower bounds on finding the MAXIMUM of n keys. The model of computation we are going to use is the decision tree model, and in fact the **parallel decision tree model** where we allow p processors to work at any time step each one performing a single comparison between two keys. The decision tree model in the case $p = 1$ was used to prove the lower bound $\Omega(n \lg n)$ for comparison-based sorting. Note that this model deals with **information gathering** to compute the MAXIMUM. One also needs to process this information to derive the MAXIMUM. The model assume that processing is **for free**. We can do so because we plan to establish a lower-bound (minimum possible running time) not to establish an algorithm for realizing this bound.

We know from the sequential setting that finding the MAXIMUM of n keys requires at least $n - 1$ comparisons.
Lemma 1. MAX can be found in ONE parallel step with $n(n-1)/2$ processors.

Proof. n keys allow $n(n-1)/2$ pairs and thus comparisons to be realized to obtain all the information required to find the MAXIMUM (one needs to do some additional processing eg. establishing the rank but that it is for free!). If we have that many processors each one responsible for one comparison, this concludes the proof.

Lemma 2. In order to find MAX in ONE parallel step we need at least $n(n-1)/2$ processors.

Proof. In order to prove a lower bound, we use a proof by contradiction. Suppose we can do it with one fewer processor $P = n(n-1)/2 - 1$. Since there are $n(n-1)/2$ pairs of keys to compares, one such pair is not compared. Call the keys of the pair x, y. What we are going to show, by playing the role of an adversary, is that we can set up the values of the n input keys so that the missing comparison of the x, y is the crucial one to establish the maximum. We thus play the role of an adversary whose only mission is to make the algorithm that uses P processors to fail. To do so, we set the results of the comparisons in such a way that x, y are the MAX and SECOND MAX keys. Thus the comparison between x, y (that is not being performed) is CRUCIAL in determining the MAXIMUM of the n keys. Since it is not performed we cannot find the MAX with P processors. Contradiction is established.

Question. What can you prove about SECOND MAX?

Some preliminry results

An interesting question is how many processors one needs to use to find the MAX of n keys not in one parallel step but in two parallel steps.

Problem 1. Find MAX in two steps with $O(n^{3/2})$ processors.

Hint. Split n keys into groups of $\sqrt{n}$. Compute MAX of each group in first step, and MAX of MAXes in second step.

Problem 2. Find MAX in two steps with $O(n^{4/3})$ processors.

Hint. Optimize the splitting. $\sqrt{n}$ might not be optimal.

Problem 3. Show that MAX in two steps requires $\Omega(n^{4/3})$ processors.

The Proof is a repetition of the arguments of the proof of Theorem 1.

A matching lower bound

Theorem 1 (Valiant) Computing the maximum of n keys requires at least lglg$n$ parallel steps with $p \leq n$ processors.
Proof. (by induction) It is proved by what we call **an adversary argument** through induction. An **adversary** for this problem is allowed to choose the input keys by modifying their values in such a way so as to force the algorithm to run for at least lglg$n$ steps. These modifications should not invalidate, however, the operations of the algorithm already performed.

View step i as a graph $G_i = (V_i, E_i)$. The vertices are the keys and the edges are the comparisons performed at step i.

Consider initially graph G which becomes G_1. Since any max finding algorithm can perform no mores than $p \leq n$ comparisons at any time step, $|E_1| \leq n$ and $|V_1| = n$. So by Turan's theorem G_1 has an independent set of size $n/4$. Call this set of keys I_1. Consider now the computation where all the I_1 keys are the larger than anything in $V_1 - I_1$. In this case every node in I_1 wins in the comparisons performed and is a candidate for the maximum. Since I_1 is an independent set we have no information on the relative order of the keys in I_1.

Consider now G_2 and take the intersection of G_2 and I_1. It has $\leq n$ edges (since $p \leq n$ can be performed at a time) and at least $n/4$ vertices (lower bound for I_1 size. So the graph G_2 intersected by I_1 has an independent set of size at least $n/64$ and call it I_2. Repeating the same thing for G_i and I_{i-1} we end up with an independent set of size $n/2^{2^{i+1}-2}$. So if we repeat this procedure about $\Omega(\lg\lg n)$ times the independent set will drop below 2 and the maximum will be established. This takes however $\Omega(\lg\lg n)$ parallel steps.

One step back

Let us prove Problem 3.

Proof (Problem 3). We set up a graph, just as in Theorem 1, with n vertices. Let us have p processors. In one step they can force p comparisons. By Turan's theorem the graph on n vertices and p edges must have an independent set of size at least $k = n^2/(n+2p)$. We can set the values of the keys or equivalently determine the output of the comparisons so that the keys of the named independent set are all candidates for the MAX. Since they form an independent set none has been compared to any other key of the set. Thus in the second round we can find the MAX among these k keys in $k(k-1)/2$ comparisons/processors using Lemma 1, if and only if we can afford to do so i.e. $p \leq k(k-1)/2$, which leads to $p = \Omega(n^{4/3})$. $\square$

Problem 4. Is there an algorithm that finds the MAX in $O(\lg \lg n)$ parallel steps using n processors? What is the work of the algorithm?

Long Hint. Consider n keys. Splits into $n/3$ groups of 3 keys each. For each group we can detemine the MAX in $3(3-1)/2 = 3$ comparisons using 3 procs per group. Total number of processors used is $n/3 \cdot 3 = n$. Thus we are left with determining the max of $n/3$ keys.

Take the $n/3$ Maxima, and split them into groups of 7. We have $n/(3 \cdot 7)$ groups of 7 keys. Each group requires $7(7-1)/2 = 21$ processors to find the MAX of the group. Total processors used is $n/21 \cdot 21 = n$, that we can afford to. Thus after the second second it suffices to find the max of $n/21$ keys to determine the MAX of the original n keys.

How do we split the $n/21$ keys next? What is the pattern?

Say we at some point we end up having n/s keys. We split them into groups of t so that $t(t-1)/2$ comparisons/processors per group. You can fill in the details to show that this way $\lg \lg n$ can be achieved.