A. V. Gerbessiotis
CIS 485/Spring 1999
Handout 2
Week 2

# The PRAM model

**Introduction**

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.

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.

(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 $0, 1, \ldots, n-1$.

# PRAM algorithms

**Parallel sum**

Value $x_i$ is initially stored in shared memory cell $i$. The sum $x_0 + x_1 + \ldots + x_{n-1}$ is to be computed in $\lg n$ parallel steps. Without loss of generality, let $n$ be a power of two. If a combining 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 running time of the algorithm is $O(1)$, and processor utilization is $O(n)$.

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 $2j$ and $2j+1$ combines (sums) these values and stores the result into memory cell $j$. After $\lg n$ steps the sum resides in cell 0.

```
//    pid() returns the id of the processor issuing the call.
begin PARALLEL_SUM (n)
    i = 1 ; j = pid();
    while (j < n/2^i)
        a = C[2j];
        b = C[2j + 1];
        C[j] = a + b;
        i = i + 1;
    end
end PARALLEL_SUM
```

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.

**Broadcasting**

A message (say, a word) is stored in cell 0 of the shared memory. We would like this message 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 message is broadcast as follows. In step $i$ each processor with index $j$ less than $2^i$ reads the contents of cell $j$ and copies it into cell $j + 2^i$. After $\lceil \lg n \rceil$ steps each processor $i$ reads the message by reading the contents of cell $i$.

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

**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. 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 $j$.

**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.

**Pointer Jumping**

We examine a technique called pointer jumping that finds many applications in designing algorithms for linked list and graph theory problems.

For an undirected graph $G = (V, E)$, $V$ denotes the set of vertices of $G$ and $E$ the set of edges. Two vertices $u$, $v$ are connected by an edge if $(u, v) \in E$. The degree of node $v$ is the number of edges incident on $v$, ie number of $u$ such that $(v, u) \in E$. A directed graph $G$ is represented by $G = (N, A)$ and is a graph whose each edge is assigned a direction. The set of vertices $N$ is sometimes called the set of nodes and the set of directed edges, the set of arcs $A$. If $u$, $v$ are connected by an arc from $u$ to $v$, then $< u, v > \in A$. For simplicity, we may write $(u, v) \in A$ as well. The out-degree of a node $u$ is the number of vertices $v$ such that $< u, v > \in A$. The in-degree of $v$ is the number of vertices $w$ such that $< w, v > \in A$. For an undirected/directed graph $G$, a path is a sequence of vertices/nodes $v_1, v_2, \ldots, v_j$ such that $(v_1, v_2), (v_2, v_3), \ldots, (v_{j-1}v_j)$ are edges of G/ $< v_1, v_2 >, < v_2, v_3 >, \ldots, < v_{j-1}v_j >$ are arcs of G. The length of the path is the number of edges/arcs on the path (eg. $j - 1$). For undirected graphs discussed in this handout we assume that they are simple i.e. they have no self-loops (edges $(v, v)$) or multiple edges. An undirected graph $G$ is connected if there is a path connecting every pair of vertices. If a simple connected undirected graph has $n - 1$ edges it is called a tree. A collection of trees forms a forest.

A rooted directed tree $T = (V, A)$ is a directed graph with a special node $r$ called the root such that (a) $\forall v \in V - \{r\}$ node $v$ has out-degree 1, and $r$ has out-degree 0, and (b) $\forall v \in V - \{r\}$ there exists a directed path from $v$ to $r$. In other words, $T$ is rooted if the undirected graph resulting from $T$ is a tree. The level of a vertex/node in a tree is the number of edges on the path to the root.

Let $F$ be a forest consisting of a set of rooted directed trees. Forest $F$ is represented by an array $P$ (P stands for "parent") of length $n$ such that $P(i) = j$ if $< i, j >$ is an arc of $F$. For a root $i$, it is $P(i) = i$.

**Problem** Given forest $F$ and array $P$ construct array $S$ where $S(j)$ is the root of the tree containing node $j$.

**Proof.** We use pointer jumping, that is, iteratively the successor of any node $i$ becomes the successor of its successor. This way the successor of a node comes closer to its root $r$. After $k$ iterations the distance between $i$ and $S(i)$ is $2^k$ unless $S(i)$ is the root (in the original forest represented by $P$). The PRAM algorithm works as follows.

If $h$ is the maximum height of any tree in $F$, the number of parallel time steps is $T = O(\lg h)$ on a CREW PRAM and the work $W = O(n \lg h)$. From this point on, we use the alternative definition of work $W$ (in most cases it doesn't make a difference which definition is used) to mean the actual number of operations performed by all the processors (which is not necessarily $P \cdot T$).

```
begin FIND_ROOT (P,S)
1.   in parallel: S(i) = P(i) ;
2.   while (S(i) ≠ S(S(i)))
3.       S(i) = S(S(i)).
end FIND_ROOT
```

**Problem** Assume that associated to node $i$ of $F$ is a value $V(i)$. Compute $W(i)$, for all $i$, where $W(i)$ is the sum of the $V(j)$ over all nodes $j$ in the path from $i$ to its root (a parallel prefix-like operation in a list/tree).
**Proof.** The PRAM algorithm works as follows.

```
begin PJ (P,S,V)
1.   in parallel: S(i) = P(i), W(i) = V(i) ;
2.   while (S(i) ≠ S(S(i)))
3.       W(i) = W(i) + W(S(i)).
4.       S(i) = S(S(i)).
end PJ
```

**Maximum Finding on a CRCW PRAM**

The maximum of $n$ numbers can be found on a EREW PRAM by utilizing the algorithm for parallel sum, if the + operator is replaced by a max operator. On a CRCW PRAM maximum finding can be computed faster in parallel time $T = O(1)$ and work $W = O(n^2)$.

As a sidenote, we first solve a simpler problem. Let binary value $X_i$ resides in the local memory of processor $i$. The problem is to find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$ in constant time on a CRCW PRAM. Processor 0 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 MAX1 (X_1 ... X_n)
1.   in proc (i,j) if X_i ≥ X_j then x_ij = 1;
2.                              else x_ij = 0;
3.   Y_i = x_i1 ∧ ... ∧ x_in ;
4.   Processor i reads Y_i ;
5.   if Y_i = 1 processor i writes i into cell 0.
end MAX1
```

Algorithm MAX1 is a time optimal algorithm for maximum finding on a CRCW PRAM. Its complexity is $T = O(1)$ and $W = O(n^2)$.

**Coloring**

**Definition 1** *A directed cycles is a directed graph $G = (V, E)$ such that the in-degree and out-degree of every node is one. Then, for every $u, v \in V$ there is a directed path from $u$ to $v$ (and from $v$ to $u$ as well). A $k$-coloring of $G$ is a mapping $c : V \to \{0, \ldots, k-1\}$ such that $c(i) \neq c(j)$, $\forall i \neq j$ and $< i, j > \in E$.*

We are interested in 3-colorings of directed cycles. In the sequential case, this problem is easy to solve. Color vertices of the cycle alternately with two colors 0 and 1 and at the end, a third color may be required for the last node of the cycle, if the first and the node before the last are colored differently. In a parallel setting this problem looks difficult to parallelize because it looks so symmetric! All vertices look alike. In order to solve this problem in parallel we represent the graph by defining $V = \{0, \ldots, n-1\}$ and an array $S$, the successor array, so that $S(i) = j$ if $< i, j > \in E$. A predecessor array can be easily derived from the identity $P(S(i)) = i$. For a number $i$ let $i = i_n \ldots i_2 i_1$ be its binary representation. Then $i_k$ is the $k$-th lsb bit of $i$.
**Claim** After a single call to COLOR1 a valid coloring is derived from a previously valid coloring.
**Proof.** Before the call to COLOR1 adjacent vertices are colored differently by a coloring say $C_1$. Let us assume for the sake of contradiction that an application of COLOR1 results in a coloring $C_2$ that fails to color properly two vertices $i, j$, i.e $c(i) = c(j)$ for $< i, j > \in E$. These colors were obtained after an application of step 3, i.e. $c(i) = 2(k-1) + c(i)|_k$

**begin** COLOR1 $(P,S,c)$
1. **in parallel** $\forall 0 \le i < n$
2. Let $k$ be the lsb position that $c(i)$ and $c(S(i))$ differ;
3. Set $c(i) = 2(k-1) +$ ($k$-th lsb of $c(i)$;
**end** COLOR1

and $c(j) = 2(l-1) + c(j)|_l$. Since $c(i) = c(j)$ (because of the $C_2$ coloring) we must have that $k = l$ and $c(i)|_k = c(j)|_k$, i.e the previous colors of $i$ and $j$ (in $C_1$) agreed in the $k$-th lsb. This contradicts the assumption that $k$ is the first lsb position where $c(i)$ and $c(j)$ differ under $C_1$.

Repeated application of COLOR1 results as it is proved below in a 6-coloring of a directed cycle as described in algorithm COLOR2A. Initially a trivial coloring of the $n$ vertices with $n$ colors is found, i.e. a mapping $c$ such that $c(i) = i$. After a call to COLOR1, an initial $n$-coloring of a directed cycle gives rise to a $(2\lg n + 1)$-coloring. If at some point a coloring uses 3 bits (up to 8 colors) then a new coloring after another application of COLOR1 would require at most six colors ($c(i) = 2(k-1) + c(i)|_k \le 5$, for $k \le 3$).

**Claim** Algorithm COLOR2A 6-colors a directed cycle.

**begin** COLOR2A $(P,S,c)$
1. **in parallel** $\forall 0 \le i < n$ set $c(i) = i$;
2. **repeat**
3. call Algorithm COLOR1 ;
4. **until** at most 6 colors are used.
**end** COLOR2A

**Proof.** Algorithm COLOR2A initially colors the vertices with $n$ colors using $c$ bits, i.e $2^{c-1} \le n < 2^c$. After the first iteration, $\lceil \lg c \rceil + 1$ bits are only used (colors $0, \ldots, 2c-1$ are used). Let us define $\lg^{(1)}(x) = \lg x$, $\lg^{(2)}(x) = \lg\lg x$ and in general $\lg^{(i)}(x) = \lg(\lg^{(i-1)}(x))$. We then define $\lg^*(x) = \min\{i : \lg^{(i)}(x) \le 1\}$. After the first iteration of Loop 2-4 an $O(\lg n)$-coloring is derived. After the second iteration an $O(\lg\lg n)$-coloring is derived. After $O(\lg^*(n))$ iterations a 6-coloring is derived. The complexity of the algorithm is thus $T = O(\lg^*(n))$ and $W = O(n \lg^*(n))$.

A question arises whether a 3-coloring is possible. As soon as a 6-coloring is obtained, a 3-coloring can be derived by perturbing the 6-coloring as in step 3 of COLOR2 below that colors the vertices of a directed cycles with 3 colors.

**begin** COLOR2 $(P,S)$
1. Call COLOR2A;
2. **do for each** $3 \le i \le 5$ ;
3. if a vertex is colored $i$ recolor it with the smallest possible color from $\{0,1,2\}$;
**end** COLOR2

Step 3 is realized in $O(1)$ parallel steps, loop 2 is repeated 3 times and COLOR2 has the same asymptotic time complexity as COLOR2A.

**List Ranking**

Consider a linked list $L$ of $n$ nodes whose order is specified by the use of a successor array ($S(i)$ is the successor of $i$ in the linked list, $0 \le i < n$). If $t$ is the tail of the list, $S(t) = 0$. The problem of *list-ranking* is to determine the distance of each node from the tail of the list. The sequential complexity of list ranking is linear in $n$ and the sequential problem is a prefix-like problem. Parallel List-ranking has many applications in parallel graph algorithms. We first present a non-optimal parallel algorithm for list-ranking (LIST1).

The time complexity of LIST1 on an EREW PRAM is $T = O(\lg n)$ and $W = O(n \lg n)$.

```
begin LIST1 (P,S)
0.   Input S(.) matrix
     Output V(i) is distance from tail of node i;
1.   ∀0 ≤ i < n do in parallel
2.      if (S(i) ≠ 0) V(i) = 1;
3.              else   V(i) = 0;
4.   ∀0 ≤ i < n do in parallel
5       B(i) = S(i);
6.      while (B(i) ≠ 0 ∧ B(B(i)) ≠ 0) do
7.          V(i) = V(i) + V(B(i));
8.          B(i) = B(B(i));
end LIST1
```