Chapter 1

PRAM Model

A Turing machine is a simple abstract computational device intended to investigate the extent of what can be computed on a sequential computer. This model enables one to define the cost of an algorithm (e.g., in number of operations) precisely and is the basis for complexity results (e.g., polynomial vs. NP-complete, etc.).

In view of the wide variety of parallel architectures, defining a precise yet general model for parallel computers seems hopeless. Perhaps most daunting is the modeling of the costs associated to data communication within the parallel computer. It turns out that a reasonable option is to simply ignore communication costs altogether! One then obtains an imperfect model in which the cost of an algorithm only bears a vague relationship to the algorithm’s execution times on real-world parallel computers. However, this model makes it possible to determine a precise classification of problems and algorithms and to obtain complexity results (e.g., to show that a given algorithm is optimal, to establish the minimal complexity of a problem).

The PRAM model, which stands for Parallel RAM, comprises a shared central memory that can be accessed by the various processing units (see Figure 1.1), or PUs. All PUs execute synchronously the same algorithm and work on distinct (or not) memory areas. In this model neither the number of PUs nor the size of the memory is bounded, which clearly does not hold for real-world parallel computers. Another simplifying assumption is that any PU can access any memory location in one unit of time. This assumption would also never hold in the real world. It is physically impossible to ensure that a large number of PUs can simultaneously access an arbitrarily large shared memory without incurring prohibitive overhead. In this model, the same memory location, or cell, can be accessed simultaneously by multiple PUs. For this reason, one typically considers three different PRAM models that specify which concurrent accesses to a memory cell are allowed:

- **CREW (Concurrent Read Exclusive Write):** This is the most common model. It is assumed that at a given time, i.e., during a given step of an algorithm, arbitrarily many PUs can read the value of a cell simultaneously while at most one PU can write a value to a cell.

- **CRCW (Concurrent Read Concurrent Write):** This is the most powerful model, with an unbounded number of simultaneous writes to a memory cell. Several possible “modes” are used to define the semantics of concurrent writes to a single memory cell:
  - Consistent mode: All PUs must write the same value.
  - Arbitrary mode: PUs can write different values but only one of them will be written. This mode does not specify which value is written, so the algorithm designer must be prepared for a non-deterministic execution.
  - Priority mode: The value written by the PU with the lowest index is chosen.
  - Fusion mode: A commutative and associative operation (e.g., a logical OR or AND, a maximum, a sum) is applied on-the-fly to the different written values.

- **EREW (Exclusive Read Exclusive Write):** This is the more restrictive model but also the more realistic one. Only one PU can have read/write access to a memory cell at a given time.

The above ER and EW options sometimes assume that a bounded number of PUs can have simultaneous read/write access to a memory cell, as opposed to just a single processor. In the EW case one must then specify the mode of concurrent writes, as in the CW case.

1.1 Pointer Jumping

In this section, we present three simple PRAM algorithms that all rely on a fundamental technique called pointer jumping. As we will see, this is a natural extension of the divide and conquer technique to a parallel setting.
1.1.1 List Ranking

Let us assume that we have a linked list \( L \) of \( n \) objects arbitrarily distributed throughout our PRAM’s memory. We wish to determine the distance \( d[i] \) to the end of the list for every element \( i \):

\[
d[i] = \begin{cases} 
0 & \text{if } \text{next}[i] = \text{Nil} \\
\text{next}[i] + 1 & \text{if } \text{next}[i] \neq \text{Nil}
\end{cases}
\]

A straightforward sequential algorithm consists in propagating and incrementing the \( d \) value from the end of the list toward the beginning of the list in \( n \) steps (assuming that the list is doubly-linked). The question is whether there is a PRAM algorithm with a complexity lower than \( O(n) \). Note that we assume that each algorithm step is executed in one time unit, so we use the number of steps, the execution time, and the complexity interchangeably. It turns out that one can design an algorithm with \( O(\log n) \) complexity, by associating each element of the list, \( i \), to a different processor, \( P_i \). This is possible because the PRAM model assumes an unbounded number of PUs.

![Figure 1.2: Typical execution of the list ranking algorithm. Gray cells indicate active values, i.e., values that are in the process of being computed.](image)

An example execution of the list ranking algorithm is depicted in Figure 1.2 for \( n = 6 \). The principle is straightforward: At each step one divides each list into two sublists. For instance, in the first step, all elements with even indices are placed in a sublist, and all elements with odd indices are placed in another sublist. The size of each sublist is divided by two at each step, leading to a number of steps equal to \( \lceil \log n \rceil \).

We show the pseudo-code for our list ranking algorithm in Algorithm 1.1. The \textbf{for all i in parallel do} loop is executed in parallel across all the PUs that are responsible for the memory cells referenced in the body of the loop. Care must be taken, however, to ensure that all iterations of the loop can be executed safely in parallel. For instance, consider the following loop (assuming that indices for array \( A \) start at 1 and that PU \( P_i \) is responsible for updating array element \( A[i] \)):

```
for all i in parallel do
    if i > 1 then
        A[i−1] ← A[i]
```

The problem here is that processor \( P_2 \) may update \( A[2] \) before \( P_1 \) can read it. The same, if not as glaring, problem occurs in statements 7 and 8 of Algorithm 1.1. To ensure that the loop executes correctly, it suffices to ensure that all read operations happen before all write operations. Therefore, the semantic of a \textbf{for all i in parallel do} loop is assumed to be as follows:

```
for all i in parallel do
    if i > 1 then
        A[i−1] ← A[i]
```

Another problem with the algorithm as it is written in Algorithm 1.1 is that the test in Statement 4 can be done in constant time only on a CRCW PRAM. On a CRCW PRAM, the test could be implemented by having all PUs write their boolean value of \( \text{next}[i] = \text{Nil} \) to a single memory cell, say \( \text{done} \). Using a CRCW with a fusion mode for concurrent writes based on a logical \( \text{AND} \), \text{done} \) is true only once the algorithm has completed. Unfortunately, there is no such constant time solution on a CREW PRAM. On a CREW PRAM, the best approach is to perform pair-wise logical \( \text{AND} \) operations in a binary tree pattern, leading to \( O(\log n) \) steps. We will further discuss this distinction between a CRCW and a CREW PRAM in Section 1.3.1.

In the particular case of our list ranking algorithm, we know that the algorithm proceeds in \( \lceil \log n \rceil \) steps. Therefore, we can rewrite the main loop:

```
while there exists a node i such that \( \text{next}[i] \neq \text{Nil} \) do
```

as

```
for step = 1 to \( \lceil \log n \rceil \) do
```
Likewise, concurrent read access can easily be removed to obtain a for the list ranking problem, there are exactly uses the same pointer jumping technique as the list ranking algorithm in the linked list such that .

1.1.2 Prefix Computation

Given a sequence \( (x_1, x_2, \ldots, x_n) \) and a binary associative operation \( \oplus \), we wish to compute the sequence \( (y_1, y_2, \ldots, y_n) \) defined as 

\[
\begin{align*}
  y_1 &= x_1 \\
  y_k &= y_{k-1} \oplus x_k = x_1 \oplus x_2 \oplus \ldots \oplus x_k, \quad \text{for } 1 < k \leq n.
\end{align*}
\]

Let us use a PRAM with \( n \) PUs. The sequence \( (x_1, x_2, \ldots, x_n) \) is stored as a linked list such that \( x[i] = x_i \) for \( i \leq n \), \( \text{next}[i] \) points to the cell that contains \( x[i+1] \) for \( i < n \), and \( \text{next}[n] = \text{Nil} \). We can devise a simple algorithm that uses the same pointer jumping technique as the list ranking algorithm in the previous section. The corresponding pseudo-code is shown in Algorithm 1.2.

An example execution of this algorithm is depicted in Figure 1.3. Just as for the list ranking problem, there are exactly \( \log n \) steps and the global condition of the while do loop can thus easily be replaced by a for do loop. Likewise, concurrent read access can easily be removed to obtain a \( O(\log n) \) EREW algorithm.

**Algorithm 1.2:** Prefix computation of a list.

One remaining question is whether this algorithm can be executed in \( O(\log n) \) time by any type of PRAM. In terms of writes, each PU \( P_i \) writes the values of \( d[i] \) and of \( \text{next}[i] \) in exclusive mode. In terms of reads, PUs \( P_i \) and \( P_j \) such that \( \text{next}[i] = j \) may read the value of \( d[i] \) concurrently, thus seemingly precluding the use of an EREW PRAM. A simple solution to overcome this problem is to introduce a temporary array and to replace 

\[
d[i] \leftarrow d[i] + \text{next}[i]
\]

by 

\[
\text{temp}[i] \leftarrow d[\text{next}[i]] \\
d[i] \leftarrow d[i] + \text{temp}[i]
\]

With this last simple transformation, we now obtain a \( O(\log n) \) algorithm on an EREW PRAM, the most restrictive PRAM model.

1.1.3 Euler Tour

In this section, we present a clever application of the prefix computation in the previous section. Consider an arbitrary binary tree with \( n \) vertices. We want to compute the depth of each vertex, i.e., its distance to the root vertex. We use a simple data structure to store the tree in memory. Each vertex \( i \) is represented by three fields: \( \text{father}[i], \text{left}[i] \), and \( \text{right}[i] \), which point to its father, its left child, and its right child, respectively, or \( \text{Nil} \) in case such parents and children do not exist.

A naive PRAM algorithm would consist of going through the tree in breadth-first fashion. The complexity is then \( O(d) \), where \( d \) is the depth of the tree. This algorithm is satisfactory when the tree is balanced because in this case \( d = O(\log n) \). However, when the tree is comb-shaped, \( d = O(n) \). The Euler tour technique leads to a \( O(\log n) \) algorithm regardless of the shape of the tree.

We associate three PUs to each vertex. We call these the A, B, and C PUs (one can think of them as three types of PUs, each with a different role in the algorithm). The principle of the algorithm is to transform the problem of computing the depth of all vertices into a single prefix computation. This is accomplished by having each PU store a value and a pointer to the value stored by another PU, thus creating a list (called the “Eulerian path”). We simply say that a PU points to another PU. We then do a prefix computation of this list. The trick consists in constructing the list so that the prefix computation leads to the desired depth values.

We depict the list for an example tree in Figure 1.4(a). The first element of the list is the value held by the A PU corresponding to the root vertex. The last element of the list is the value held by the C PU at the root also. We establish the links between the list elements as follows:

- The A PU of each vertex points to the A PU of the vertex’s left child if any, or to the vertex’s own B PU otherwise.
- The B PU of each vertex points to the A PU of the vertex’s right child if any, or to the vertex’s own C PU otherwise.
1.1 Pointer Jumping

Once the above list is established and the values of its elements are initialized, which can be done in constant time on an EREW PRAM, we can perform a prefix computation for the addition operator. The reader can easily check that this computation leads to the values shown in Figure 1.4(b). The depth-first path through the tree. We denote by $x[i]$ the value for which PU $P_i$ is responsible. These values are initialized as follows:

- If a vertex has a left child, then its $C$ PU points to the $B$ PU of the vertex’s father; otherwise, its $C$ PU points to the $C$ PU of the vertex’s father (the $C$ PU of the root vertex points to $Nil$).

The above creates a depth-first path through the tree. We denote by $x[i]$ the value for which PU $P_i$ is responsible. These values are initialized as follows:

$$x[i] = \begin{cases} 
1 & \text{if } P_i \text{ is a PU of type } A, \\
0 & \text{if } P_i \text{ is a PU of type } B, \\
-1 & \text{if } P_i \text{ is a PU of type } C.
\end{cases}$$

The depth of a vertex is stored in the value store by the $C$ PU associated to that vertex. In hindsight we now see the rationale for the initial values of the list elements:

- The $A$ PU of vertex $i$ adds 1 to the prefix sum, accounting for going down a level in the tree. More precisely, $d(\text{left}[i]) = d(i) + 1$ (where $d(i)$ is the depth of vertex $i$).
- The $B$ PU’s contribution to the sum is equal to 0 because the depth of the left child of a vertex is equal to the depth of the vertex’s right child.
- The $C$ PU’s contribution to the sum is equal to $-1$, accounting for going up a level in the tree.

We have reduced our problem to a constant-time initialization phase and a prefix computation with $O(n)$ PUs. Therefore, using the algorithm in the previous section, we obtain an $O(\log n)$ EREW algorithm.

Figure 1.4(a) shows these values, written as ‘PU type = value’ for simplicity. The left child of a vertex is equal to the depth of the vertex’s right child. The $PU$ of the root vertex points to the left child of that vertex. The cost $C_{\delta}(n)$ of a PU of type $\delta$ is equal to $\delta(n)$ (the number of PUs). Therefore, the cost is minimal if, at each step, all PUs are used to perform useful computations, i.e., computations that are part of the sequential algorithm. In this case, the cost is equal to the work.

The speedup of a PRAM algorithm is the factor by which the program’s execution time is lower than that of the sequential program.

DEFINITION 1.1 (Cost and Work). The cost of a PRAM algorithm is defined as $C_{\delta}(n) = p \cdot T_{\delta}(p, n)$. The work $W_{\delta}(n)$ of a PRAM algorithm is the sum over all PUs of the number of performed operations. The difference between cost and work is that the work does not account for PU idle time.

Intuitively, cost is a rectangle of area $T_{\text{par}}(p, n)$ (the execution time) times $p$ (the number of PUs). Therefore, the cost is minimal if, at each step, all PUs are used to perform useful computations, i.e., computations that are part of the sequential algorithm. In this case, the cost is equal to the work.

The speedup of a PRAM algorithm is the factor by which the program’s execution time is lower than that of the sequential program.

DEFINITION 1.2 (Speedup and Efficiency). The speedup of a PRAM algorithm is defined as $S_{\text{par}}(n) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(p, n)}$.

The efficiency of the PRAM algorithm is defined as

$$e_{\text{par}}(n) = \frac{S_{\text{par}}(n)}{p} = \frac{T_{\text{seq}}(n)}{p \cdot T_{\text{par}}(p, n)}.$$  

Some authors use the following definition: $S_{\text{par}}(n) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(1, n)}$, where $T_{\text{par}}(1, n)$ is the execution time of the algorithm with a single PU. This definition quantifies how the algorithm scales: if the speedup is close to $p$ (i.e., is $\Omega(p)$), one says that the algorithm scales well. However, this definition does not reflect the quality of the parallelization (i.e., how much we can really gain by parallelization), and it is thus better to use $T_{\text{seq}}(n)$ than $T_{\text{par}}(1, n)$.

1.2.2 A Simple Simulation Result

In this section we present our first “simulation result.” With this result one can reason about the performance of an algorithm for a given PRAM given a
1.2. Performance Evaluation of PRAM Algorithms

The main idea is that a PRAM with fewer PUs can simulate a PRAM with more PUs by executing fewer operations concurrently (with some loss of performance).

PROPOSITION 1.1. Let $A$ be an algorithm whose execution time is $t$ on a PRAM with $p$ PUs. $A$ can be simulated on a RAM of the same type with $p' \leq p$ PUs in time $O\left(\frac{t}{p'}\right)$. The cost of the algorithm on the smaller PRAM is at most twice the cost on the larger PRAM.

Proof. Each step of $A$ can be simulated in at most $\frac{t}{p'}$ time units with $p' \leq p$ PUs by simply reducing concurrency and having the $p'$ PUs perform sequences of operations. Since there are at most $t$ steps, the execution time $t'$ of the simulated algorithm is at most $\frac{t}{p'} t$. Therefore, $t' = O\left(\frac{t}{p'}\right) = O\left(\frac{t}{p}\right)$.

We also have $C_{p'} = t' p' \leq \frac{t}{p'} (p' + 1) = p t (1 + \frac{1}{p}) = C_p (1 + 1/p) \leq 2C_p$.

A direct consequence of this result is that the cost of an algorithm is at least of the same order of magnitude as the sequential complexity (otherwise, the simulation of this algorithm with a single PU would lead to a lower complexity). A PRAM algorithm is said to be efficient when its cost has the same order of magnitude as the complexity of the corresponding sequential algorithm (i.e., its efficiency is $\Omega(1)$). Consequently, the work of an efficient algorithm cannot be reduced by more than a constant factor.

When reducing the number of processors in Proposition 1.1, we have grouped operations from a parallel algorithm and simulated them on a single processor. In practice, this is often the case that these operations could be executed faster using a sequential algorithm directly. It is, for example, easy to design a mixed version of the prefix computation algorithm whose execution time with $p$ PUs is $O(n/p + \log(p))$ instead of $O(n, \log(n)/p)$. Assuming the list is split into $p$ sub-lists of $n/p$ consecutive elements, each PU can sequentially compute the prefix for its own sub-list in $O(n/p)$. Then, one can propagate sub-prefixes of the sub-lists in $O(\log p)$.

The study of the efficiency of algorithms is extremely important in practical contexts. Indeed, in the real world, resources are not free and the efficiency can decrease sharply as $p$ increases beyond some reasonable threshold. The typical reason for this decrease in efficiency is that when $p$ becomes too large relative to $n$, communication costs become large. Further increases of $p$ can even increase overall execution time.

There are scenarios in which it is possible to solve a problem more than $p$ times faster using $p$ processors than when using a single processor of the same type. In this case efficiency is greater than 1 and one speaks of super-linear speedup. One cause of super-linear speedup is that $p$ processors typically have $p$ times as much cache and memory as a single processor. Therefore, when using sufficiently many processors, the entire data for the problem at hand may suddenly fit entirely in memory (or in cache), thus avoiding costly swapping (or cache misses). It is important to understand that, although highly desirable in practice, achieving super-linear speedup does not mean that the parallel algorithm is particularly efficient. From a strictly parallel algorithms perspective, comparing a parallel execution of an algorithm that achieves super-linear speedup to a sequential execution that runs out of memory or cache is not a fair comparison. One option would be to compute a speedup relative to the execution time when using the smallest number of processors that leads to a super-linear speedup.

1.2.3 Brent's Theorem

THEOREM 1.1 (Brent). Let $A$ be an algorithm that executes a total number of $m(i)$ operations and that runs in time $t$ on a PRAM (with some unspecified number of PUs). $A$ can be simulated in time $O\left(\frac{t}{p} + t\right)$ PRAM of the same type that contains $p$ PUs.

Proof. Say that at step $i$ $A$ performs $m(i)$ operations (implying that $\sum_{i=1}^{m(i)} m(i) = m$). Step $i$ can be simulated with $p$ PUs in time $m(i) \leq \frac{m(i)}{p} + \frac{m(i)}{p} + 1$. One can simply sum these upper bounds to prove the theorem.

Brent’s theorem makes it possible to quantify the performances of an algorithm when the number of PUs is reduced. Let us consider for example the computation of the maximum of a list of integers on an EREW PRAM. This computation can be done in $O(\log n)$ time by structuring the computation as a binary tree, and computing pair-wise maxima at each step until there is only one value left. The first step requires $O(n)$ PUs. But what if we had fewer than $O(n)$ PUs? Brent’s theorem tells us that we can simulate the algorithm on $p$ PUs in $O\left(\frac{t}{p} + \log n\right)$ time (we perform $m = n - 1 = O(n)$ comparisons).

Therefore, if we choose $p = \frac{n}{\log n}$, we achieve the same execution time but with fewer PUs!

1.3 Comparison of PRAM Models

In this section, we discuss the comparative power of the EREW, CREW, and CRCW models.
1.3. Comparison of PRAM Models

One may wonder whether a CRCW PRAM is more powerful than a CREW PRAM. The problem of model comparison can be stated as such: Is there a problem such that, using the same number of PUs, we can solve this problem with a CRCW PRAM strictly faster than with the best algorithm for solving this problem with a CREW PRAM?

The problem of finding the maximum value of an array gives us the answer to the previous question. With Algorithm 1.3, we can compute the largest value in array $A$ in $O(1)$ time on a CRCW PRAM with $O(n^2)$ PUs. In this algorithm all PUs write the same value concurrently. False, and thus the PRAM runs in consistent mode. The first and the third loops use only one PU whereas the second loop uses $n(n-1)$ PUs, with $P_{i,j}$, $i \neq j$, being responsible for the comparison $A[i] < A[j]$.

More generally, with a CRCW PRAM with $n$ PUs one can compute the fusion $\bigoplus_{i \in [n]} e_i$ of $n$ elements $(e_1, \ldots, e_n)$, where $\bigoplus$ is an associative operator, in $O(1)$ time. Such a reduction operation cannot be done in fewer than $\Omega(\log n)$ steps with a CREW PRAM. Indeed, with this model, at most two values can be merged into a single one in one step. Therefore, the number of values that need to be merged is halved at each step for a total of $\Omega(\log n)$ steps.

What about the CREW and EREW models then? Here again, there is a simple problem that separates these models: Determine whether a given element $e$ belongs to a set $(e_1, \ldots, e_n)$ of $n$ distinct elements. This problem can be easily solved in $O(1)$ steps using a CREW PRAM with $n$ PUs. First, a boolean variable, $res$, is initialized to $False$. Then, in parallel, each PU $P_i$ compares $e$ with $e_i$ (implying concurrent read access to $e$) and set $res$ to $True$ if both elements are equal. Since the $e_i$ are pair-wise distinct, at most one PU will modify $res$ (ensuring exclusive writes).

On an EREW PRAM, PUs cannot read $e$ simultaneously. Comparisons can be done in $O(1)$ units of time only if $\Omega(n)$ copies of $e$ have been created, which cannot be done in less than $\Omega(\log n)$ time. More generally, broadcasting information to $n$ PUs on an EREW PRAM requires $\Omega(\log n)$ steps, with double the number of copies of the information at each step.

1.3.2 Simulation Theorem

In the previous section, for each model, we have exhibited algorithms for which the separation factor between the different PRAM models was $\Omega(\log p)$. One may, however, wonder whether larger factors could be found. The following theorem proves that these factors are in fact maximal.

THEOREM 1.2. Any CRCW algorithm with $p$ PUs has an execution time of at most $O(\log p)$ times lower than the best EREW algorithm with $p$ PUs for solving the same problem.

Proof. Let us assume that the CRCW PRAM uses a consistent mode (only concurrent writes of the same values are authorized). We show a method to simulate concurrent writes with only exclusive writes (concurrent reads can be handled in a similar fashion).

Let us consider a given step of the CRCW algorithm and simulate it with $O(\log p)$ steps of an EREW algorithm. Both PRAMs have the same computing power (i.e., the same number of processors). Therefore, we only need to focus on memory accesses. Our EREW algorithm requires a temporary array $A$ of size $p$. When PU $P_i$ of the CRCW algorithm writes a value $x_i$ to address

\begin{algorithm}
\begin{algorithmic}[1]
  \State \textbf{COMPUTE-MAXIMUM}(\(A, n\))
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \State \(m[i] \leftarrow True\)
  \EndFor
  \ForAll{\(i, j \in \{1, \ldots, n\}^2, i \neq j\)} \textbf{in parallel} do
    \If{\(A[i] < A[j]\)} \State \(m[i] \leftarrow False\)
  \EndIf
  \EndFor
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \If{\(m[i] = True\)} \State \(\text{max} \leftarrow A[i]\)
  \EndIf
  \EndFor
  \Return \text{max}
\end{algorithmic}
\end{algorithm}

\text{ALGORITHM 1.3: CRCW algorithm to compute the largest value of an array.}

\text{1.3.1 Model Separation}

One may wonder whether a CRCW PRAM is more powerful than a CREW PRAM? How do we know that the answer is no? One way to prove it is to construct an algorithm that can simulate the CRCW algorithm with the CREW algorithm. In this section, we present two such simulations: one for CRCW simulation with exclusive writes, and another for CRCW simulation with exclusive reads.

\text{Example 1.3.1.} Let us consider the CRCW algorithm for computing the maximum value of an array. We can simulate this algorithm with the following CREW algorithm:

\begin{algorithm}
\begin{algorithmic}[1]
  \State \textbf{COMPUTE-MAXIMUM}(\(A, n\))
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \State \(m[i] \leftarrow True\)
  \EndFor
  \ForAll{\(i, j \in \{1, \ldots, n\}^2, i \neq j\)} \textbf{in parallel} do
    \If{\(A[i] < A[j]\)} \State \(m[i] \leftarrow False\)
  \EndIf
  \EndFor
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \If{\(m[i] = True\)} \State \(\text{max} \leftarrow A[i]\)
  \EndIf
  \EndFor
  \Return \text{max}
\end{algorithmic}
\end{algorithm}

\text{ALGORITHM 1.3: CRCW algorithm to compute the largest value of an array.}

\text{1.3.2 Simulation Theorem}

In the previous section, for each model, we have exhibited algorithms for which the separation factor between the different PRAM models was $\Omega(\log p)$. One may, however, wonder whether larger factors could be found. The following theorem proves that these factors are in fact maximal.

THEOREM 1.2. Any CRCW algorithm with $p$ PUs has an execution time of at most $O(\log p)$ times lower than the best EREW algorithm with $p$ PUs for solving the same problem.

Proof. Let us assume that the CRCW PRAM uses a consistent mode (only concurrent writes of the same values are authorized). We show a method to simulate concurrent writes with only exclusive writes (concurrent reads can be handled in a similar fashion).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure_1.5.png}
\caption{Simulation of concurrent writes with exclusive writes.}
\end{figure}

Let us consider a given step of the CRCW algorithm and simulate it with $O(\log p)$ steps of an EREW algorithm. Both PRAMs have the same computing power (i.e., the same number of processors). Therefore, we only need to focus on memory accesses. Our EREW algorithm requires a temporary array $A$ of size $p$. When PU $P_i$ of the CRCW algorithm writes a value $x_i$ to address

\begin{algorithm}
\begin{algorithmic}[1]
  \State \textbf{COMPUTE-MAXIMUM}(\(A, n\))
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \State \(m[i] \leftarrow True\)
  \EndFor
  \ForAll{\(i, j \in \{1, \ldots, n\}^2, i \neq j\)} \textbf{in parallel} do
    \If{\(A[i] < A[j]\)} \State \(m[i] \leftarrow False\)
  \EndIf
  \EndFor
  \ForAll{\(i \in \{1, \ldots, n\}\)} \textbf{in parallel} do
    \If{\(m[i] = True\)} \State \(\text{max} \leftarrow A[i]\)
  \EndIf
  \EndFor
  \Return \text{max}
\end{algorithmic}
\end{algorithm}

\text{ALGORITHM 1.3: CRCW algorithm to compute the largest value of an array.}
1.4 Sorting Machine

In 1986, Cole [43] proposed a very elegant CREW PRAM algorithm to sort $n$ numbers in $O(n \log n)$ time with $O(n)$ PUs. This algorithm is optimal as the sequential complexity of sorting (using only comparisons) is $O(n \log n)$. Cole’s algorithm is based on the classical merge sort algorithm whose representation as a binary tree (see Figure 1.6) reveals potential parallelism: All merging steps of a given level of the tree can be done in parallel. Let us assume that we process each level in sequence. Since there are $\log n$ levels, we need to be able to process each level in $O(1)$ time to get a $O(n \log n)$ total execution time. How can two arbitrary-sized lists be merged in $O(1)$ time? This is where the beauty of Cole’s algorithm resides. Partial information from previous merges is used to compute current merges in only a constant number of steps, a very clever but by no means straightforward algorithm. Some readers may thus wish to skip this section due to its rather intricate mathematical content.

1.4.1 Merge

In all that follows we assume that we sort/merge arrays of integers. The merge of two sorted sequences $J$ and $K$ is denoted $J/K$. We say an integer $x$ is between $a$ and $b$ if and only if $a < x \leq b$.

**DEFINITION 1.3 (Rank).** The rank of an element $x$ in a sequence $J$ is defined as the number of elements of $J$ that are smaller than $x$:

$$\text{rank}(x, J) = \text{card}\{ j \in J \mid j < x \} .$$

Likewise, the cross-rank of $A$ in $B$ is the function $B/A : A \to \mathbb{N}$

$$e \mapsto \text{rank}(e, B) .$$

This function can be represented as an array of size $|A|$ whose $i$-th entry is the rank of the $i$-th element of $A$ in $B$.

---

**FIGURE 1.6:** Example binary tree for Cole’s parallel merge sort.

**DEFINITION 1.4 (Good Sampler).** A sequence $L$ is said to be a good sampler (GS) of a sequence $J$ if, for any $k > 0$, there are at most $2k + 1$ elements of $J$ between $k + 1$ (arbitrary) consecutive elements of $\{-\infty\} \cup L \cup \{+\infty\}$.

Intuitively, $L$ is a GS of $J$ if elements of $L$ are (almost) uniformly distributed among the elements of $J$. For $k = 1$, the definition imposes that there is no more than three elements of $J$ between two consecutive elements of $\{-\infty\} \cup L \cup \{+\infty\}$. For example, the set $\text{Even}(J)$ (resp. $\text{Odd}(J)$) of elements of even (resp. odd) indices in $J$ is a GS of $J$. Let us consider $J = \{j_1, \ldots, j_n\}$ and check that $\text{Even}(J)$ is a GS of $J$. Consider $k + 1$ consecutive elements $\{j_{2k}, j_{2k+1}, \ldots, j_{2k+k}\}$ of $\{-\infty\} \cup \text{Even}(J) \cup \{+\infty\}$ (we use the convention $j_0 = -\infty$ and $j_{k+1} = +\infty$ if $2(i + k) > n$). We can easily check that there are either $2k - 1$ or $2k$ elements of $J$ between $j_{2k}$ and $j_{2k+k}$.

Using good samplers, two sorted sequences can be quickly merged: Let us consider two sorted sequences $J$ and $K$, and let us assume that $L$ is a GS of $J$ and $K$ (we set $l_n = -\infty$ and $l_{k+1} = +\infty$). Figure 1.7 illustrates the partitioning of $J$ and $K$.

We can now define a function $\text{Merge}_{\text{with\,help}}(\cdot)$, which is shown in Algorithm 1.4. Let us explain how this function works with an example:

Let $J = [2, 3, 7, 8, 10, 14, 15, 17, 18, 21]$ and $K = [1, 4, 6, 9, 11, 12, 13, 16, 19, 20]$. $L = [5, 10, 12, 17]$ is a GS of both $J$ and $K$, which can be checked exhaustively for $k = 1$ (5 checks for $\{-\infty, 5\}$, (5, 10), (10, 12), (12, 17), and $\{17, +\infty\}$), for $k = 2$ (4 checks for $\{-8, 10\}$, (4, 12), (10, 17), and $\{17, +\infty\}$), and so on. We obtain:

- $J(1) = [2, 3]$, $J(2) = [7, 8, 10]$, $J(3) = \emptyset$, $J(4) = [14, 15, 17]$, and $J(5) = [18, 21]$.
LEMMA 1.1. We can sort write accesses in exclusive mode. We proceed likewise to

\[
\begin{align*}
L &= L_1 L_2 L_3 L_4 L_5 L_6 L_7 L_8 \\
K &= K(1) K(2) K(3) K(4) K(5) K(6) K(7) K(8) K(9) \\
J[K] &= (J(1)K(1), J(2)K(2), \ldots, J(9)K(9))
\end{align*}
\]

FIGURE 1.7: Merging J and K with the help of L.

ALGORITHM 1.4: Merge with help algorithm.

\begin{align*}
\text{res}_1 &= \text{Merge}(J, [2, 3], [1, 4]) = [1, 2, 3, 4] \\
\text{res}_2 &= \text{Merge}(J, [7, 8, 10], [6, 7, 8, 9, 10]) \\
\text{res}_3 &= \text{Merge}(L, [11, 12]) = [11, 12] \\
\text{res}_4 &= \text{Merge}(J, [14, 15, 17], [13, 16]) = [13, 14, 15, 16, 17] \\
\text{res}_5 &= \text{Merge}(L, [18, 21], [19, 20]) = [18, 19, 20, 21].
\end{align*}

LEMMA 1.1. If L is a GS of the sorted sequences J and K, and if cross-ranks \(R(J, L)\), \(R(J, K)\), \(R(K, L)\), \(R[K, L]\) are known, then Merge_with_help(J, K, L) runs in \(O(1)\) time with \(|J| + |K|\) PUs on a CREW PRAM.

PROOF. Each of the three steps can be done in \(O(1)\) time using \(R(J, L)\), \(R(J, K)\), \(R[K, L]\) and \(R(J, L)\) and \(R(K, L)\).

- Step 1: J is partitioned using \(|J|\) PUs. Each \(P_j, j \in J\), reads \(rank(j, L) = r\) and inserts \(j\) in \(J(r)\). As there are at most three PUs writing into \(J(r)\), we can sort write accesses in exclusive mode. We proceed likewise to
The proof of correctness of Cole’s algorithm is based on three main invariants:

- The size of the input of nodes at level \( k \) in the tree doubles at each step (from 1 to \( 2^k \)) until the node is complete.
- If \( X(t) \) is a GS of \( X(t+1) \) and \( Y(t) \) is a GS of \( Y(t+1) \), then \( Z(t) \) is a GS of \( Z(t+1) \).
- We can compute \( R[S(t+1), S(t)] \) for any sequence of input or output of a node.

Study of the Example

![Diagram of Cole's parallel merge sort](image)

**FIGURE 1.8**: Sorting an array of size 8 with Cole’s parallel merge sort.

In this section, we describe in detail how the algorithm works with an input of size 8 (see Figure 1.8). At step \( t = 0 \), leaves have sequences of size \( 2^0 \) and are hence complete. At steps \( t = 1 \) and \( t = 2 \), they send one element out of four, then one out of two, i.e., no element at all. At step \( t = 3 \), they send their unique value to their father and stop working.

Let us focus on the father \([8,7] \) of leaves 8 and 7. At step \( t = 3 \), it computes \( \text{val}(5) \), i.e., \( \text{MERGE\_WITH\_HELP}(\{8\}, \{7\}, 0) \), and becomes complete (it is a level 1 node). At step \( t = 4 \), it does not send anything to its father. At step \( t = 5 \), it sends one element out of every two, i.e., \([8\]) \. At step \( t = 6 \), it sends its two elements, and then stops working.

Let us now focus on the root node of the subtree \([8,7,6,5] \). At step \( t = 5 \), it receives \([8\]) and \([6\]) and computes \( \text{val}(5) = \text{MERGE\_WITH\_HELP}(\{8\}, \{6\}, 0) \). At step \( t = 6 \), it receives \( X(6) = \{7, 8\} \) and \( Y(6) = \{5, 6\} \) and computes \( \text{val}(6) = \text{MERGE\_WITH\_HELP}(\{7, 8\}, \{5, 6\}, \text{val}(5)) \). We can check that \( \text{val}(5) \) is a GS of \( X(6) \) and \( Y(6) \). At step \( t = 6 \), the node is complete (it is a level 2 node). At step \( t = 7 \), it sends \([8\]) and at step \( t = 8 \), it sends \([6, 8\]) and \( \text{val}(9) = \text{MERGE\_WITH\_HELP}(\{5, 6, 7, 8\}, \{1, 2, 3, 4\}, \text{val}(8)) \). On can easily check that \( \text{val}(t) \) is a GS of \( X(t+1) \) and \( Y(t+1) \) for \( t = 6, 7, 8 \). At the end of step \( t = 9 \), the root is complete and all values are sorted.

### 1.4.3 Complexity and Correctness

The following lemma gives the relationship between the algorithm’s execution time and the number of processors.

**LEMMA 1.2.** The pipelined sorting tree algorithm runs in \( O(\log n) \) time with \( O(n) \) PUs.

**Proof.** A level \( k \) node is complete at step \( t = 3k \). This is easily proved by induction on \( k \) using the fact that children send all their data in three steps once they are complete. The execution time for the tree is hence \( O(\log n) \).

At level \( k \) of the tree, \( \frac{n}{2^k} \) nodes merge lists of size smaller than \( 2^k \). Therefore, \( O(n) \) PUs are required to process a level in \( O(1) \) time. One could hastily conclude that \( O(n \log n) \) PUs are required to implement the pipelined sorting tree. However, a finer analysis reveals that (i) all levels are not active at the same time and (ii) the amount of work per level doubles at each step. Level \( k \) PUs merge lists of size \( 2^k \) at step \( t = 3k, 2^k-1 \) at step \( t = t-1, 2^{k-1} \) at step \( t - 2 \), etc. The number of PUs needed for each step of the whole tree is thus \( O(n) \).

Let us now prove the following good sampler invariant.
LEMMA 1.3. Let $X$, $X'$, $Y$, and $Y'$ be four sorted sequences. If $X$ is a GS of $X'$ and if $Y$ is a GS of $Y'$, REDUCE($X|Y$) is a GS of REDUCE($X'|Y'$).

Proof. We still use the notation $X|Y$ to denote the fusion of $X$ and $Y$. If $X$ is a GS of $X'$, then $X|W$ is clearly still a GS of $X'$ for any set $W$. However, if $X$ is a GS of $X'$ and $Y$ is a GS of $Y'$, then $X|Y$ is not necessarily a GS of $X'|Y'$.

Indeed, let us consider for example $X = [2,7]$, $X' = [2,5,6,7]$, $Y = [1,8]$, and $Y' = [1,3,4,8]$. Then we have $X|Y' = [1,2,7,8]$, and $X'|Y' = [1,2,3,4,5,6,7,8]$ but there are five elements of $X'|Y'$ between 2 and 7 (that are yet consecutive elements of $X|Y$). This is the reason why we resort to the reduce operator.

Let us prove the following property: There are at most 2 elements of $X'|Y'$ between $r$ consecutive elements of $X|Y$ (we assume that $-\infty$ and $+\infty$ are in $X$ and $Y$).

Proof. Let us consider a sequence $e_1, e_2, \ldots, e_r$ of $r$ consecutive elements of $X|Y$. $h_1$ elements among these $r$ elements come from $X$ and $h_2$ come from $Y$ (with $h_1 + h_2 = r$). Without loss of generality, we can assume that $e_1 \in X$.

We consider two cases:

Case 1 ($e_1 \in X$): As $X$ is a GS of $X'$, there are at most 2($h_1+1$) elements of $X'$ between $e_1$ and $e_r$. These are also at most 2($h_2+1$) elements of $Y'$ as these elements are between $h_1+2$ elements of $X$, which is a GS of $X'$. Hence, there are at most 2($h_1+1$) + 2($h_2+1$) + 1 = 2$r+2$ elements of $X'|Y'$ between $e_1, e_2, \ldots, e_r$.

Case 2 ($e_1 \in Y$): Let us add an element $e_0 \in Y$ preceding $e_1$ and an element $e_{r+1} \in X$ following $e_r$. Then, elements from $X'$ and $Y'$ lying between $e_0, e_1, e_2, \ldots, e_r$ come from elements for $X'$ lying between $h_1+1$ elements of $X$ and from elements of $Y'$ lying between $h_1+1$ elements of $Y$.

Therefore, we have $(2h_1+1) + (2h_2+1) = 2r+2$ elements of $X'|Y'$.

Coming back to the proof of the lemma, let us define $Z = REDUCE(X|Y)$ and $Z' = REDUCE(X'|Y')$. Let us consider $k+1$ consecutive elements $z_1, z_2, \ldots, z_{k+1}$ of $Z$. Since the reduce operator keeps one element out of every four, we have $z_1 = e_0$, $z_2 = e_k(a+1), \ldots, z_{k+1} = e_k(k+4)$, where $X|Y = \{e_1, e_2, \ldots, e_k\}$. Thus, there are $4k+1$ elements of $X|Y$ between $z_1, z_2, \ldots, z_{k+1}$. Using the previous property with $r = 4k+1$, we know that there are at most $8k+4$ elements of $X'|Y'$ between these $4k+1$ elements. Since the reduce operator keeps one element out of every four, there are at most $\frac{1}{4}(8k+4) = 2k+1$ elements of $Z'$ between the $k+1$ consecutive elements of $Z$, proving that $Z'$ is a GS of $Z$.

In steady-state, a node receives a sorted sequence $X(t+1)$ from its left child and a sorted sequence $Y(t+1)$ from its right child. It computes $val(t+1) = REDUCE(val(t), X(t+1), Y(t+1), val(t))$ and sends $Z(t+1) = REDUCE(val(t+1))$ to its father. Since Lemma 1.3 shows that $Z(t)$ is a GS of $Z(t+1)$, we have the following invariants:

1. $val(t) = X(t)|Y(t)$.
2. $X(t)$ is a GS of $X(t+1)$.
3. $Y(t)$ is a GS of $Y(t+1)$.

Note that the property still holds true for the last two communications (one element out of every two is sent, then finally all the elements).

Lastly, we have to ensure that the requirements of Lemma 1.1 hold true: We need to know the cross-ranks for $\text{MERGE}_{\text{with help}}(*)$ to run in $O(1)$ time. At a given step, $X$ is a GS of $X'$ and $Y$ is a GS of $Y'$, $U = X|Y$, and $Z = REDUCE(U)$. We can assume that cross-ranks $R[X', X]$ and $R[Y', Y]$ are known by the induction hypothesis. To compute $U' = X'|Y'$ with $\text{MERGE}_{\text{with help}}(*)$, we need to know the cross-ranks $R[X', U]$, $R[Y', U]$, $R[R, U]'$, and $R[R', U]'$. Finally, we can compute $Z' = REDUCE(U')$ and $R[Z', Z]$ to get our invariant. Of course, we assume that for each sorted sequence $S$ we know the cross-rank $R[S, S]$. In other words, we know the index of each element in the sorted sequence. This is computed as part of the internal representation of $S$.

LEMMA 1.4. If $S = [b_1, b_2, \ldots, b_n]$ is a sorted sequence, then the rank of a given element $a$ in $S$ can be computed in $O(1)$ time with $O(k)$ PUs on a CREW PRAM.

Proof. Take $b_0 = -\infty$ and $b_{n+1} = +\infty$. The rank of $a$ is then computed with the following loop:

$\text{forall } i, 0 \leq i \leq k \text{ in parallel do}$

$\text{if } b_i < a \leq b_{i+1} \text{ then}$

$L.$ rank $+1$

There are no write conflicts because only one processor stores the value of $rank$.

LEMMA 1.5. If we have three sorted sequences $S_1, S_2, S$ such that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$, then we can compute the cross-ranks $R[S_1, S_2]$ and $R[S_1, S]$ in $O(1)$ time with $O(S)$ PUs.

Proof. We assume that whenever a sequence $S$ is sorted, we also know $R[S, S]$. For any $a \in S_i \subset S$ we have:

$\text{rank}(a, S_2) = \text{rank}(a, S) - \text{rank}(a, S_1)$

Hence the results.

We can now prove our invariant on cross-ranks for $\text{MERGE}_{\text{with help}}(*)$.}
LEMMA 1.6. Suppose we have sorted sequences \(X, Y, U = X \cup Y, X',\) and \(Y'\) such that \(X\) is a GS of \(X', Y\) is a GS of \(Y',\) and we know the cross-ranks \(R[X', X]\) and \(R[Y', Y]\). Then, we can compute the cross-ranks \(R[X', U], R[Y', U], R[U, X']\), and \(R[U, Y']\) in \(O(1)\) time with \(O(|X| + |Y|)\) PUs.

Proof. We first show how to compute \(R[X', U]\). Let \(X = \{a_0, a_2, \ldots, a_k\}\) and take \(a_0 = -\infty\) and \(a_{k+1} = +\infty\). We partition the sequence \(X'\) with \(X\):
\[
X'(i) = \{x' \in X', a_{i-1} < x' \leq a_i\} \quad \text{for} \quad 1 \leq i \leq k + 1.
\]

This partition is computed in \(O(1)\) time with \(O(|X|)\) PUs because we know \(R[X, X]\). We also partition \(U\) with \(X\) (which is the same as partitioning \(Y\) with \(X\) because \(U = X \cup Y\)):
\[
U(i) = \{y \in Y, a_{i-1} < y \leq a_i\} \quad \text{for} \quad 1 \leq i \leq k + 1.
\]
We compute the rank of \(x\) in \(U\) as follows:

\[
\text{forall } i, 1 \leq i \leq k + 1 \text{ in parallel do}
\]
\[
\text{forall } x' \in X'(i) \text{ do}
\]
\[
\text{Compute } \text{rank}(x', U(i)) \quad \{ \text{with Lemma 1.4} \}
\]
\[
\text{rank}(x', U) = \text{rank}(a_{i-1}, U) + \text{rank}(x', U(i))
\]

For each \(i\) we use \(O(|U(i)|)\) PUs. Altogether we thus require \(O(|U|)\) PUs. As \(X\) is a GS of \(X',\) each \(X'(i)\) consists of at most three elements and thus the computation runs in \(O(1)\) time. We have therefore computed \(R[X', U], R[Y', U]\) is of course computed in a similar fashion.

To compute \(R(U, X')\) we need \(R[X, X']\) and \(R[Y, X']\). Let us see how to compute \(R[X, X']\). Consider an element \(a_i\) from \(X \setminus X'\) and search for the minimal element \(a'\) of \(X'(i+1)\). The rank of \(a_i\) in \(X'\) is the same as the rank of \(a'\) in \(X'\). This rank is already computed as part of the internal representation of the sorted sequence \(X'\). Thus, we can compute \(\text{rank}(a_i, X')\) in \(O(1)\) time with a single processor. To compute \(R[Y, X']\) consider \(y \in Y\). We compute \(\text{rank}(y, X)\) using Lemma 1.5, because \(U = X \cup Y\) is already computed. Then, we compute \(\text{rank}(y, X)\) using \(\text{rank}(y, X)\) and \(R[X, X']\). This way, we compute \(R(U, X')\) in \(O(1)\) time with \(O(|U|)\) PUs. We can compute \(R(U, Y')\) in a similar way.

\[\text{LEMMA 1.7. Using Lemma 1.6's notation, let}
\]
\[Z = \text{Reduction}(U)\] and \[Z' = \text{Reduction}(U').\]

We can compute \(R[Z', Z]\) in \(O(1)\) time with \(O(|X| + |Y|)\) PUs.

Proof. The proof is straightforward. \(R[X', Z]\) is computed from \(R[X', U]\). Likewise, \(R[Y', Z]\) is computed from \(R[Y', U]\) and we obtain \(R[U', Z]\) as a subset of \(Z\), hence the result.

We have at last proved the following theorem:
1.5. Bibliographical Notes

Some of the introduction material in this chapter is inspired by the books by Cormen, Leiserson, and Rivest [44] and by Gengler, Ubeda, and Desprez [59]. The presentation of Cole’s parallel merge sort algorithm is taken from the book by Gibbons and Rytter [61]. The original article by Cole [43] also presents an EREW version of the sorting machine with the same performance. For additional information on the PRAM model, we refer the reader to the book by Reif [100].