The Owner Concept for PRAMs*

Peter Rossmanith†

Technische Universität München, Institut für Informatik,
Arcisstr. 21, D-8000 München 2

Abstract

We analyze the owner concept for PRAMs. In OROW-PRAMs each memory cell has one distinct processor that is the only one allowed to write into this memory cell and one distinct processor that is the only one allowed to read from it. By symmetric pointer doubling, a new proof technique for OROW-PRAMs, it is shown that list ranking can be done in $O(\log n)$ time by an OROW-PRAM and that $LOGSPACE \subseteq$ OROW-TIME($\log n$). Then we prove that OROW-PRAMs are a fairly robust model and recognize the same class of languages when the model is modified in several ways and that all kinds of PRAMs intertwine with the NC-hierarchy without timeloss. Finally it is shown that EREW-PRAMs can be simulated by OREW-PRAMs and ERCW-PRAMs by ORCW-PRAMs.

*This research was partially supported by the Deutsche Forschungsgemeinschaft, SFB 342, Teilprojekt A4 "Klassifikation und Parallelisierung durch Reduktionsanalyse"

†E-mail: `rossmani@lan.informatik.tu-muenchen.dbp.de`

Introduction

Fortune and Wyllie introduced in [FW78] a parallel version of a random access machine, which they called P-RAM. These parallel RAMs consist of infinitely many processors, each a complete RAM with usual instruction set and an unbounded local memory. Additionally, there is also an unbounded global memory, which can be accessed by each processor.

The P-RAM model was intended to be a model for parallel computers and to help to classify the class of problems that are solvable by efficient parallel algorithms. These efficient parallel algorithms are required to be fast, i.e., polylogarithmic in the length of the input, and to use only a reasonable amount of hardware, i.e., only polynomially many processors must be involved. The class of problems for which such efficient parallel algorithms exist is commonly called NC. NC is a fairly robust class and can also be characterized in terms of alternating Turing machines [CKS81, Ruz80], Boolean circuits [Coo85, Ven87], and auxiliary pushdown automata [Ruz80, Ruz81].

However, though NC is a mathematically very interesting class, the underlying models are not very realistic models for parallel computers. There are several reasons for this drawback:

- The number of processors is seen as a resource. In reality there is always a fixed number of processors.
- The costs for access to global and local memory are identical, i.e., each access to global or local memory is done in exactly one step. In reality an access to global memory has to be interpreted as a communication between two processors, which takes a lot more time than an access to local memory in massively parallel computers.
- We don't only allow unbounded parallelity but also unbounded communication: Each two processors can communicate through global memory. In reality the communication topology is more restricted, e.g. to a mesh connected array or a hyper-cube.

Each of these drawbacks has been discussed in literature, but restricting P-RAMs more realistically leads to complexity classes that are hard to handle.

There is, however, another property of parallel random access machines, which separates unrealistic P-RAMs from slightly more realistic ones: Fortune and Wyllie defined that any number of processors is allowed to simultaneously read from a single global memory cell, but only one processor may write into each memory cell at each step. To distinguish this model from other ones, Snir called this type of PRAM a CREW-PRAM [Sni82] (concurrent read, exclusive write parallel random access machine). We can also consider PRAMs in which any number of processors is allowed to write simultaneously into a single memory cell. This type is called CRCW-PRAM (concurrent-read, concurrent-write parallel random access machine). Obviously, CREW-PRAMs are more realistic than CRCW-PRAMs. An even more restrictive (and realistic) model was introduced by Dymond and Ruzzo [DR86] called CROW-PRAM (concurrent-read, owner-write parallel random access machine). In a CROW-PRAM simultaneous write attempts of more than one processor to a single memory cell are also forbidden, but additionally for each memory cell there must be one distinct processor that is the only one allowed to write into this memory cell. This processor is also called the *owner* of that memory cell.

All three different models mentioned so far proved to describe reasonable and robust complexity classes. The class of languages recognized by CRCW- and CREW-PRAMs in logarithmic time can also be characterized in terms of uniform Boolean circuits [SV84, Lan90] and the language class recognized by CROW-PRAMs in logarithmic time is exactly the class of languages reducible to deterministic context-free languages in logarithmic space and there is also a characterization in terms of auxiliary pushdown automata [DR86, Sud78].

It is only natural to apply these three different methods to solve write conflicts also to read conflicts. Since the owner-write concept is fairly new, this only led to EREW-PRAMs by now (see e.g. [LPV81, Sni82]), but it seems useful to consider EROW- and finally OROW-PRAMs, too. An OROW-PRAM can be seen as a PRAM without global memory, but provided with communication channels between some pairs of processors. Certainly, OROW-PRAMs are the most realistic model mentioned by now.

In this paper we will particularly stick to OROW-PRAMs. After a more formal definition, we will show that OROW-PRAMs can solve non-trivial problems very fast, though their communication abilities are quite restricted. We show that three important problems are solvable by OROW-PRAMs in $O(\log n)$ time and, as an application of this, we prove that logarithmic space bounded Turing machines can be simulated in $O(\log n)$ time. This result is already known for EREW-PRAMs [KR87], but we need to invent a new proof technique to gain this inclusion also for OROW-PRAMs.

In the next section we will deal with the stability of the OROW model. It is shown that the class of languages accepted by OROW-PRAMs in $O(\log^k n)$ time remains the same if the model is generalized or restricted in several ways. As a major result we can show that the functions that map a memory cell to its read- or write-owner may depend on input or time, a result that is already known for CROW-PRAMs (see [DR86]).

In the last section we consider several other models with different combinations of read- and write-restrictions. In particular, we will prove that EREW-PRAMs can be simulated by OREW-PRAMs, while increasing the running time only by a constant factor. Similarly, ERCW-PRAMs can be simulated by ORCW-PRAMs.

Owner-Read PRAMs

We use a definition of PRAMs similar to Stockmeyer and Vishkin's definition in [SV84]: Each PRAM has an unbounded global memory and infinitely many processors. Each processor has an unbounded local memory and an instruction counter. The program for each processor must be the same and may consist of finitely many instructions. The available instruction set contains add, subtract, division by two, jump, jump if zero, jump if greater than zero, assignment of a constant to a local memory cell, indirect reads and writes from (resp. into) the global memory, and halt. Additionally each processor can read its number.

Initially all global and local memory cells contain zero except the first n global memory cells that contain each one bit of the input string and the first local memory cell of each processor, which contains the input length n. The first $p(n)$ processors are started simultaneously at the beginning of their programs, where $p(n)$ is some polynomial.

To distinguish between PRAMs, in which write conflicts are allowed (resp. forbidden) and read conflicts are allowed (resp. forbidden), we will use the nomenclature introduced by Snir in [Sni82]. This results in CRCW, CREW, EREW, and ERCW models. Here "CR" means *concurrent-read*, "ER" means *exclusive-read*, "CW" means *concurrent-write*, and finally "EW", which means *exclusive-write*. A PRAM is exclusive-write (resp. exclusive-read), if two processors never attempt to write into the same memory cell (resp. read the same memory cell) at the same time. This restriction must hold for all possible inputs!

Dymond and Ruzzo defined the owner-write restriction, which is more restrictive than the exclusive-write restriction:

Definition 1 [DR86] A CRCW-algorithm is called *owner-write*, if there is a function $\mathit{write\text{-}owner}(i,n)$ that maps memory cells into the set of processor numbers and if $p = \mathit{write\text{-}owner}(i,n)$, then p is the only processor allowed to write into memory cell i. This function must only depend on i and the input length, i.e., n, and must be log-space computable.

Dymond and Ruzzo introduced the class CROW-TIME($\log n$) of languages for which an concurrent-read, owner-write algorithm exists running in $O(\log n)$ time and using polynomially many processors. They further observed that most CREW algorithms are in fact CROW algorithms or can be simply modified to become owner-write. Analogously, the class EROW-TIME($\log n$) of languages recognizable by exclusive-read, owner-write algorithms can be considered. Again we see that most EREW algorithms are in fact owner-write, i.e., EROW algorithms (we will see this later by some examples).

The owner-write restriction seems to be a fairly reasonable concept and is also a step into the direction of more realistic models of parallel computation. It is only a natural consequence to consider a dual *owner-read restriction*, too.

Definition 2 A CRCW-algorithm is called *owner-read*, if there exists a function $\mathit{read\text{-}owner}(i,n)$ that maps memory cells into the set of processor numbers and if $p = \mathit{read\text{-}owner}(i, n)$, then p is the only processor allowed to read from memory cell i. Again this function must only depend on i and n, and must be log-space computable.

PRAMs that respect the owner-write (resp. owner-read) restriction are denoted according to Snir's nomenclature (extended by Dymond and Ruzzo) by the letters OW (resp. OR). Obviously, PRAMs can be defined obeying any combination of read and write restrictions mentioned above. There are nine possible combinations: CRCW, CREW, CROW, ERCW, EREW, EROW, ORCW, OREW, and finally OROW, the most restrictive one. We denote the class of languages recognizable by XRYW algorithms, $X, Y \in \{C, E, O\}$, in $O(\log^k n)$ time using polynomially many processors by XRYW-TIME($\log^k n$). In diagrams this class will eventually be abbreviated by XRYW^k.

Since each owner-write algorithm is exclusive-write and each exclusive-write algorithm is concurrent-write and analogous results hold for the corresponding read constraints, we can immediately give some relations between the nine XRYW^k-classes. These straightforward inclusions are illustrated in Figure 1.

The Power of OROW-PRAMs

To demonstrate the power of OROW-PRAMs we will show that three basic problems can be solved by an OROW-PRAM in $O(\log n)$ time: prefix sums, list ranking, and expression evaluation. Algorithms for these basic problems are needed when attacking certain graph theoretical problems, e.g., spanning forests, maximal matchings, expression evaluation, and connected components (see [KR87]). The inclusion of *LOGSPACE* in OROW-TIME($\log n$) will be an important implication of this observation.

Let $+$ be an associative operation over a domain D and $(x_1, \ldots, x_n)$ an array of n elements from D. The *prefix sums problem* is to compute the n sums $s_i = \sum_{k=1}^{i} x_k$, $i = 1, \ldots, n$ (see also [Fic83, LF80]).

A generalization of prefix sums is *list ranking*. The input variables are given in the form of a linked list and the sums are computed from

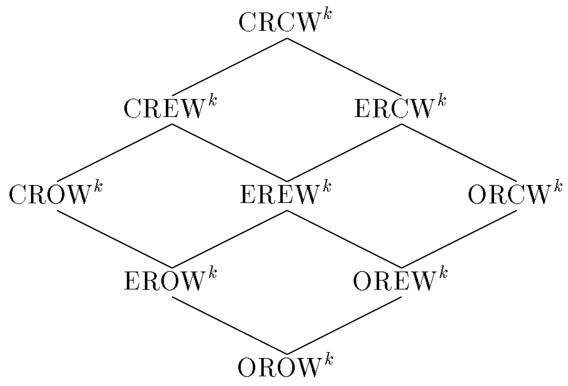

Figure 1: Relations between several types of PRAMs

the end rather than from the beginning, i.e., $s_i = \sum_{k=i}^{n} x_k$.

Let $v(i)$ be the value of the ith node in the linked list and $F(i)$ its successor, then the following simple algorithm solves the prefix sums problem [Wyl81]:

do $\lceil \log n \rceil$ **times**
  for $i = 1, \ldots, n$ **pardo**
    $v(i) := v(i) + v(F(i))$
    $F(i) := F(F(i))$

W.l.o.g. we have to assume $F(l) = 0$, $F(0) = 0$, and $v(l) = 0$, where l is the last element in the list.

This algorithm was intended to be EREW, but it is in fact EROW. However, it is in no way an *owner read* algorithm. The owner read constraint can be achieved when dropping the owner write property: First each node tells its successor its own address and then each node can tell its predecessor the address of its successor. So the instruction $F(i) := F(F(i))$ is substituted by the two instructions $B(F(i)) := i$

and $F(B(i)) := F(i)$. They are clearly owner read and serve the same purpose as the single instruction $F(i) := F(F(i))$.

Loosing the owner write property we had to pay a high price for this "improvement", but we observe that temporarily constructing reverse pointers $B(i)$ was an essential trick to gain an owner read algorithm.†

We will now try to start with a doubly linked list and keep pointers into both directions during the advance of the algorithm rather than constructing reverse pointers in each step. Since the *presence* of reverse pointers gains owner read, but their *construction* destroys owner write, we can hope to get a both owner read and owner write algorithm by this method. Let $1, \ldots, n$ be the nodes of a doubly linked list. The successor of i is represented by $F(i)$ and its predecessor by $B(i)$. If a node i has no successor (resp. predecessor), we define $F(i) = 0$ (resp. $B(i) = 0$). Additionally, a value $v(i)$ is assigned to each node i. The sums we want to compute can be recursively defined as

$$s(i) := \begin{cases} v(i) & \text{if } F(i) = 0, \\ v(i) + s(F(i)) & \text{otherwise.} \end{cases}$$

For this task we can immediately write down an algorithm that keeps pointers in both directions during its advance (yet not using these reverse pointers):

> **do** $\lceil \log n \rceil$ **times**
>   **for** $i = 1, \ldots, n$ **pardo**
> $v(i) := v(i) + v(F(i))$
> $F(i) := F(F(i))$
> $B(i) := B(B(i))$

Here we will assume $F(l) = 0$, $F(0) = 0$, $v(l) = 0$, and additionally $B(f) = 0$ and $B(0) = 0$, where f is the first element in the linked list. It is not easy to see that this algorithm keeps forward and backward pointers in such a way that there is always a corresponding backward pointer for each forward pointer and vice versa, i.e., $F(B(i)) = i$ and

† We will see later that this trick can be applied to every EREW-algorithm yielding an OREW-algorithm.

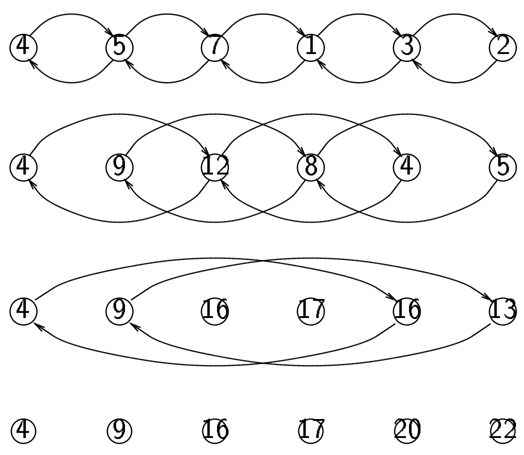

Figure 2: Computing prefix sums of a doubly linked list

$B(F(i)) = i$ (if $B(i) \neq 0$, resp. $F(i) \neq 0$). We can verify this behavior on an example given in Figure 2. A formal proof of this invariant can be given for the final algorithm.

By now we took no advantage of the presence of pointers in both directions. In order to do so we introduce memory cells labeled $S(i, j)$ that will serve as communication cells between the nodes i and j. For this purpose we define

- $\textit{write-owner}(S(i, j)) = j$,
- $\textit{read-owner}(S(i, j)) = i$.

With the help of these communication cells we can implement the instruction $F(i) := F(F(i))$ in an OROW way:

Let $j = F(i)$. The instruction $F(i) := F(F(i))$ gives information, i.e., the value $F(j)$, from node j to node i. Node i knows from where the information comes and node j knows where it goes. Now the communication cells come into play. We substitute $F(i) := F(F(i))$ by two instructions: Node j performs $S(i,j) := F(j)$ and afterwards node i performs $F(i) := S(i,j)$. Using the equations $j = F(i)$ and $i = B(j)$, which will be proved as an invariant of the algorithm, these instructions can also be written as $S(B(j),j) := F(j)$ and $F(i) := S(i,F(i))$. Analogously we can substitute $v(i) := v(i) + v(F(i))$ by $S(B(j),j) := v(j)$ and $v(i) := v(i) + S(i,F(i))$, and finally $B(i) := B(B(i))$ can be substituted by $S(F(k),k) := B(k)$ and $B(i) := S(i,B(i))$, if we assume $k = B(i)$. Applying these substitutions to the algorithm we obtain:

> **do** $\lceil \log n \rceil$ **times**
>   **for** $j = 1,\ldots,n$ **pardo** $S(B(j),j) := v(j)$
>   **for** $i = 1,\ldots,n$ **pardo** $v(i) := v(i) + S(i,F(i))$
>   **for** $j = 1,\ldots,n$ **pardo** $S(B(j),j) := F(j)$
>   **for** $k = 1,\ldots,n$ **pardo** $S(F(k),k) := B(k)$
>   **for** $i = 1,\ldots,n$ **pardo** $F(i) := S(i,F(i))$
>   **for** $i = 1,\ldots,n$ **pardo** $B(i) := S(i,B(i))$

It is straightforward that this algorithm can be simplified yielding the desired OROW algorithm for list ranking:

> **do** $\lceil \log n \rceil$ **times**
>   **for** $i = 1,\ldots,n$ **pardo**
> $S(B(i),i) := v(i)$; $v(i) := v(i) + S(i,F(i))$
> $S(B(i),i) := F(i)$; $S(F(i),i) := B(i)$
> $B(i) := S(i,B(i))$; $F(i) := S(i,F(i))$

This algorithm is, however, not working correct in all cases by now. The instruction $F(i) := F(F(i))$ works perfectly well if $F(i) = 0$ since we defined $F(0) = 0$, but this behavior is not mimicked by the new algorithm. Here $F(i) := F(F(i))$ was substituted by $S(B(j),j) := F(j)$ and $F(i) := S(i,F(i))$ by $j = F(i)$ and $i = B(j)$. But if $F(i) = 0$ then there exists no j with $j = F(i)$, so $S(i,F(i)) = S(i,0)$ was never initialized!

Since we want the instruction $F(i) := F(F(i))$ to assign zero to $F(i)$ in case $F(i)$ is already zero, all we have to do is to preassign zero to $S(i,0)$ for all i. This can be done once before the main loop:

for $i = 1, \dots, n$ **pardo** $S(i,0) := 0$
do $\lceil \log n \rceil$ **times**
  for $i = 1, \dots, n$ **pardo**
    $S(B(i), i) := v(i);\ v(i) := v(i) + S(i, F(i))$
    $S(B(i), i) := F(i);\ S(F(i), i) := B(i)$
    $B(i) := S(i, B(i));\ F(i) := S(i, F(i))$

We can now prove the correctness. We denote the value of $F(i)$ before the kth execution of the main loop by $F(i)^{(k)}$. If $k > 0$ then $F(i)^{(k)}$ is also the value of $F(i)$ right after the $(k-1)$st execution. The denotations $B(i)^{(k)}$, $v(i)^{(k)}$, and $S(i,j)^{(k)}$ are defined analogously.

Lemma 3 *For* $k = 0, \dots, \lceil \log n \rceil$*:*

(i) $F(F(i)^{(k)})^{(k)} = F(i)^{(k+1)}$

(ii) $B(B(i)^{(k)})^{(k)} = B(i)^{(k+1)}$

(iii) $B(i)^{(k)} \neq 0 \Rightarrow \exists j\colon j = B(i)^{(k)} \wedge i = F(j)^{(k)}$

(iv) $F(i)^{(k)} \neq 0 \Rightarrow \exists j\colon j = F(i)^{(k)} \wedge i = B(j)^{(k)}$

Proof. We give a proof by induction on k. Suppose (i)–(iv) are true for some k. We will show that (i)–(iv) also hold for $k+1$:

(iii) Suppose $B(i)^{(k+1)} \neq 0$. Then $B(i)^{(k)} \neq 0$ also holds. Because of the induction hypothesis there is a j such that $i = F(j)^{(k)}$ and $j = B(i)^{(k)}$ holds. Again because of the induction hypothesis we have $B(i)^{(k+1)} = B(B(i)^{(k)})^{(k)}$ and therefore $B(i)^{(k+1)} = B(j)^{(k)} =: m$. We can further conclude

$$F(m)^{(k+1)} = F(F(m)^{(k)})^{(k)} = F(\underbrace{F(B(j)^{(k)})^{(k)}}_{=\, j \text{ by ind. hyp.}})^{(k)} = F(j)^{(k)} = i.$$

We proved: $B(i)^{(k+1)} \neq 0 \Rightarrow \exists m\colon m = B(i)^{(k+1)} \wedge i = F(m)^{(k+1)}$, i.e., (iii) for $k+1$.

(iv) similar to (iii).

(i) Let us suppose $F(i)^{(k)} = 0$. Then instruction $F(i) := S(i, F(i))$ yields

$$\begin{aligned} F(i)^{(k+1)} &= S(i, F(i)^{(k)})^{(k+1)} \\ &= S(i, 0)^{(k+1)} \\ &= 0 \\ &= F(0)^{(k)} \\ &= F(F(i)^{(k)})^{(k)}. \end{aligned}$$

If $F(i)^{(k)} \neq 0$ then there exists a j such that $j = F(i)^{(k)}$ and $i = B(j)^{(k)}$. Instruction

$$\textbf{for } i = 1, \ldots, n \textbf{ pardo } S(B(i), i) := F(i)$$

yields $S(B(j)^{(k)}, j)^{(k+1)} = F(j)^{(k)}$, i.e.,
$S(i, F(i)^{(k)})^{(k+1)} = F(F(i)^{(k)})^{(k)}$, and instruction

$$\textbf{for } i = 1, \ldots, n \textbf{ pardo } F(i) := S(i, F(i))$$

yields $F(i)^{(k+1)} = S(i, F(i)^{(k)})^{(k+1)} = F(F(i)^{(k)})^{(k)}$.

(ii) similar to (i). □

Remark 4 Thus we have shown that $B(i) \neq 0 \Rightarrow \exists j\colon j = B(i) \wedge i = F(j)$, $F(i) \neq 0 \Rightarrow \exists j\colon j = F(i) \wedge i = B(j)$ and therefor $B(i) \neq 0 \Rightarrow F(B(i)) = i$, $F(i) \neq 0 \Rightarrow B(F(i)) = i$ are invariants of the main loop. The latter will be used in the next proof and was already used in the proof of Lemma 3.

The next step is to prove that the sums are computed correctly. We denote the l-times application $F(F \ldots F(i) \ldots))$ of F by $F^l(i)$. Using this notation the next lemma follows straightforwardly from the equation $F(i)^{(k+1)} = F(F(i)^{(k)})^{(k)}$:

Lemma 5 $F(i)^{(k)} = F^{2^k}(i)^{(0)}$.

We are now able to prove an explicit formula for the value of $v(i)$ before the kth execution of the main loop.

Lemma 6 $v(i)^{(k)} = \sum_{l=0}^{2^k-1} v(F^l(i)^{(0)})^{(0)}$.

Proof. (by induction on k)

$k=0$: immediate.

$k \to k+1$: Assume $F(i)^{(k)} = 0$. Then we can conclude using Lemma 5

$$(*) \qquad F(i)^{(k)} = 0 \Rightarrow F^{2^k}(i)^{(0)} = 0 \Rightarrow \forall l \geq 2^k\colon F^l(i)^{(0)} = 0.$$

The value $v(i)^{(k+1)}$ is computed as follows:

$$\begin{aligned}
v(i)^{(k+1)} &= v(i)^{(k)} + S(i, F(i)^{(k)})^{(k+1)} \\
&= v(i)^{(k)} + \underbrace{S(i,0)^{(k+1)}}_{=0} \\
&= v(i)^{(k)} \\
&= \textstyle\sum_{l=0}^{2^k-1} v(F^l(i)^{(0)})^{(0)} \\
&= \textstyle\sum_{l=0}^{2^k-1} v(F^l(i)^{(0)})^{(0)} + \underbrace{\textstyle\sum_{l=2^k}^{2^{k+1}-1} v(F^l(i)^{(0)})^{(0)}}_{=0,\ \text{see } (*)} \\
&= \textstyle\sum_{l=0}^{2^{k+1}-1} v(F^l(i)^{(0)})^{(0)}
\end{aligned}$$

If $F(i)^{(k)} \neq 0$, we know that there is some j with $j = F(i)^{(k)}$ and $B(j)^{(k)} = i$.

$$\begin{aligned}
v(i)^{(k+1)} &= v(i)^{(k)} + S(i, F(i)^{(k)})^{(k+1)} \\
&= v(i)^{(k)} + S(B(j)^{(k)}, j)^{(k+1)} \\
&= v(i)^{(k)} + v(j)^{(k)} \\
&= v(i)^{(k)} + v(F(i)^{(k)})^{(k)} \\
&= v(i)^{(k)} + v(F^{2^k}(i)^{(0)})^{(k)} \\
&= \textstyle\sum_{l=0}^{2^k-1} v(F^l(i)^{(0)})^{(0)} + \sum_{l=0}^{2^k-1} v(F^l(F^{2^k}(i)^{(0)})^{(0)})^{(0)} \\
&= \textstyle\sum_{l=0}^{2^{k+1}-1} v(F^l(i)^{(0)})^{(0)}
\end{aligned}$$

□

Since the main loop is iterated $\lceil \log n \rceil$ times, the contents of $v(i)$ will be $\sum_{l=0}^{n} v(F^l(i)^{(0)})^{(0)}$ upon termination, what is exactly the ith prefix sum. This implies a major result of this paper:

Theorem 7 *List ranking and prefix sums can be solved by an OROW-PRAM in time $O(\log n)$ using polynomially many processors.*

Proof. The above algorithm can be implemented on an OROW-PRAM. We assign one processor to each node. Processor i is write-owner *and* read-owner of $F(i)$, $B(i)$, and $S(i,0)$. The write-owner of $S(i,j)$ is processor j and the read-owner of $S(i,j)$ is processor i. □

Remark 8 The list ranking algorithm starts with a doubly linked list, but in the original specification only a simple list is available. However, an OROW-PRAM can easily construct reverse pointers in time $O(\log n)$ by broadcasting $F(i)$ from node i to all other nodes (for $i = 1, \ldots, n$).

Another detail of the list ranking algorithm has not been discussed, yet. How are the memory locations $S(i,j)$ mapped into the linear address space? Problems of this kind are often skipped, but since we introduce a new model of computation we have to take a little more care on these details.

A straightforward approach is to map $S(i,j)$ into memory cell $n \cdot i + j$. This mapping has to be carried out in constant time, without using multiplication. We can do this after a little preprocessing: Before the actual list ranking algorithm is initiated, the OROW-PRAM computes an array $(n, 2n, 3n, \ldots, n^2)$. If each entry is computed by a single processor this will take only $O(\log n)$ time. Such a look-up table has to be provided separately for each node in order to work owner read during list ranking, so we have to use n^2 processors for this task.

Having access to this table, the mapping $(i,j) \mapsto n \cdot i + j$ can easily be done in constant time. Besides $S(i,j)$ we also have to map $v(i)$, $F(i)$, $B(i)$, and the look-up tables into a linear address space. In detail this can be done according to Table 1. If unit-cost multiplication is available (only for polynomially bounded values), list-ranking clearly requires only $O(n)$ processors.

The next problem that we show to be solvable by an OROW-PRAM is tree contraction, which has expression evaluation as an important application.

Given a complete, rooted, and ordered, binary tree, whose internal nodes are labeled by operation symbols and its leaves by numbers, we

real address	pseudo address	read owner	write owner
$0 \quad \ldots \quad n-1$	$v(1)\ldots v(n)$	$1\ldots n$	$1\ldots n$
$n \quad \ldots \quad 2n-1$	$F(1)\ldots F(n)$	$1\ldots n$	$1\ldots n$
$2n \quad \ldots \quad 3n-1$	$B(1)\ldots B(n)$	$1\ldots n$	$1\ldots n$
$3n \quad \ldots \quad 4n-1$	$S(1,0)\ldots S(n,0)$	$1\ldots n$	$1\ldots n$
$4n \quad \ldots \quad 5n-1$	$S(1,1)\ldots S(n,1)$	$1\ldots n$	1
$5n \quad \ldots \quad 6n-1$	$S(1,2)\ldots S(n,2)$	$1\ldots n$	2
$\vdots \quad \vdots$	$\vdots$	$\vdots$	$\vdots$
$(n+3)n \quad \ldots \quad (n+4)n-1$	$S(1,n)\ldots S(n,n)$	$1\ldots n$	n
$(n+4)n \quad \ldots \quad (n+5)n-1$	look-up table $(n, 2n, \ldots, n^2)$ for node 1	1	$1\ldots n$
$\vdots \quad \vdots$	$\vdots$	$\vdots$	$\vdots$
$(2n+3)n \quad \ldots \quad (2n+4)n-1$	look-up table $(n, 2n, \ldots, n^2)$ for node n	n	$(n-1)n+1\ldots n^2$

Table 1: Mapping to a linear address space

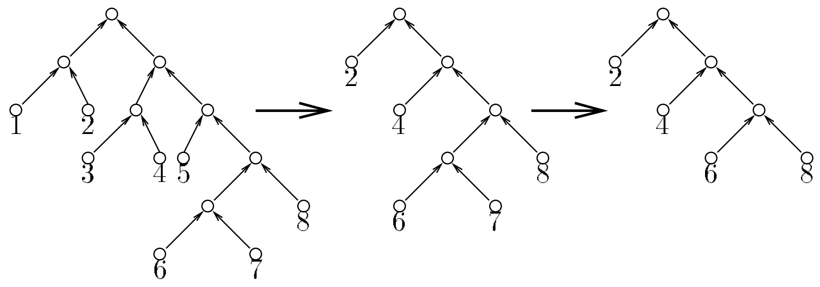

Figure 3: The SHUNT operation

want to compute the value of its root, i.e., the value that is computed by applying the operations of the internal nodes in the natural way. Tree contraction is an EREW-algorithm introduced in [MR85], which solves this problem. Many variants of this method can be found in [GR86, CV88, CV89, ADKP89, KD87, GMT87, KRS, SS89]. We only sketch a variant taken from [KR87]:

1. Label the leaves in order from left to right as $1, \ldots, n$.
2. do $\lceil \log n \rceil$ times
   (a) Apply SHUNT in parallel to all odd numbered leaves that are the left child of their parent.
   (b) Apply SHUNT in parallel to all odd numbered leaves that are the right child of their parent.
   (c) Shift out the rightmost bit in the labels of all remaining leaves.

The SHUNT operation is illustrated in Figure 3†.

This algorithm works in $O(\log n)$ time and is again in fact an EROW-algorithm. Similarly to list ranking we can modify tree contraction to become OROW.

†This example is taken from [KR87]

The first step, i.e., to number the leaves from left to right, can be done by list ranking after generating an Euler tour on the tree (see [TV85]), which is clearly OROW. The SHUNT operation can be carried out in an OROW way, too, if we provide reverse pointers. However, we leave out the details here.

Theorem 9 *Tree contraction can be performed in $O(\log n)$ steps by an OROW-PRAM using polynomially many processors.*

The ability to do list ranking allows an OROW-PRAM to simulate a log-space bounded Turing machine:

Theorem 10 *LOGSPACE* $\subseteq$ OROW-TIME$(\log n)$.

Proof. Let M be a log-space bounded Turing machine. We assign a group of memory cells to each configuration of M. Each processor computes the successor and the predecessors of the configuration it is assigned to. This way a configuration graph of M can be built in the global memory. Since M is a deterministic Turing machine, this graph is in fact a forest. The trees of this forest can be provided with Euler tours. Finally, we can test by list ranking for each node, whether the root of its tree is an accepting or rejecting configuration. $\square$

Corollary 11 OROW-TIME$(\log^k n)$ *is closed under log-space reductions for $k \geq 1$.*

By Theorem 10 we have shown that OROW-PRAMs are powerful devices though their communication ability is rather restricted. We will also observe that OROW-PRAMs can characterize the NC-hierarchy and intertwine within it.

Theorem 12

1. $NC^k \subseteq$ OROW-TIME$(\log^k n) \subseteq NC^{k+1}$, $k \in \mathbb{N}$
2. $NC = \bigcup_{k>0}$ OROW-TIME$(\log^k n)$

Proof. (sketch) OROW-TIME($\log^k n$) $\subseteq NC^{k+1}$ is an immediate implication of OROW-TIME($\log^k n$) $\subseteq$ CRCW-TIME($\log^k n$) and CRCW-TIME($\log^k n$) $\subseteq NC^{k+1}$ (see e.g. [SV84]).

An NC^k-circuit can be divided into $O(\log^{k-1} n)$ layers of depth $\log n$. Each of those layers can be simulated by an OROW-PRAM in $O(\log n)$ steps yielding $O(\log^k n)$ steps in total. □

Robustness of the OROW Model

Reasonable models of computation are required to be robust, i.e., the classes of languages they accept must be invariant at a broad range of modifications and parameter changes. Dymond and Ruzzo showed in [DR86] that CROW-PRAMs are fairly robust. They introduced several generalizations and restrictions of the CROW model and proved equality of the most generalized and most restricted model as follows: Deterministic context-free languages can be recognized by the most restricted CROW-PRAM and, on the other hand, $LOGDCFL$ contains all languages recognized by the most general model of CROW-PRAMs.

We will show that all generalizations and restrictions of CROW-PRAMs can also be applied to OROW-PRAMs without changing their language classes. Furthermore, all modifications that are also reasonable for the owner read concept are considered. However, in contrast to [DR86] we have to simulate directly the generalized model by the restricted one since we do not have different characterizations of OROW-PRAMs.

Theorem 13 *The OROW model is invariant under following generalizations and restrictions:*

G1 *The owner functions depend on time.*

G2 *The owner functions depend on the input.*

G3 *"Limited multiple ownership": Each memory cell has a set of constant many owners.*

G4 *"Ill behaved programs I": Violation of owner constraints is allowed. If an owner constraint is violated, the PRAM will halt and reject.*

G5 *"Ill behaved programs II": Violation of owner constraints is allowed. If an owner constraint is violated, the corresponding write instruction (resp. read instruction) is ignored.*

G6 *The owner functions are* OROW-TIME($\log n$) *computable.*

R1 *"Communication channels": For each pair $P_1 \neq P_2$ of processors there must be at most one memory cell M with read-owner$(M) = P_1$ and write-owner$(M) = P_2$.*

R2 *Require the owner functions to be NC^1 computable.*

R3 *Require each processor to use only $O(1)$ memory locations.*

Proof. In the first step a simulation of an OROW-PRAM with generalizations G1, G2, and G6 by an OROW-PRAM with restrictions R1 and R2 is given. After this we separately deal with G3, G4, and G5 in a second step.

Let P be an OROW-PRAM using $q(n)$ memory locations and $p(n)$ processors, where p and q are polynomials. P may be generalized according to G1, G2, and G6 (or all of them). P' is an OROW-PRAM obeying restrictions R1 and R2, which will simulate P.

P' uses memory cells m_j, w_{ij}, and r_{ij} and processors P_i, R_j, and W_j, where $1 \leq i \leq p(n)$ and $1 \leq j \leq q(n)$. The idea of the simulation is as follows: P_i simulates the ith processor of P and m_j is used to hold the value of $G(j)$, i.e., the jth memory cell of P. If the ith processor of P writes into memory location j, then P_i does not write directly into m_j, but writes into memory cell w_{ij} and then the contents of w_{ij} is copied into memory cell m_j by processor W_j. Similarly, P_i does not read directly from memory cell m_j, but reads from r_{ij} instead. Before this is done, the right value is copied from m_j to r_{ij} by processor R_j.

As can be seen easily, this algorithm is owner-read and owner-write, if we choose the owner functions according to Table 2.

The only question that remains open by now is how processor W_j knows from where to read the value that has to be copied into memory

Memory location	Read-owner	write-owner
m_j	R_j	W_j
w_{ij}	W_j	P_i
r_{ij}	P_i	R_j

Table 2: Owner functions

cell m_j. A similar question is, how processor R_j knows where to write the value read from m_j. However, though P_i cannot tell R_j to read from w_{ij}, R_j can compute all by itself that w_{ij} contains a newly written value. Since the function $\mathit{write\text{-}owner}(j,n)$ is OROW-TIME($\log n$) computable, we can assume that $\mathit{write\text{-}owner}(j,n)$ was computed in a preprocessing step, and that R_j knows the write-owner of j, i.e., i. Obviously, this is also possible if the owner functions depend on the input. If the owner functions depend on time, then R_j can be provided with a table that contains $\mathit{write\text{-}owner}(j,n)$ for each time step. This table has only a polylogarithmic size. Since R_j can count the simulated steps, it can compute the right memory cell even if the owner functions are both time and input dependent. Processor W_j can compute analogously, where to write.

In a more formal notation, a global write step can be written as follows:

for $i = 1, \ldots, p(n)$ **pardo** $G(j_i) := L_i(k_i)$

We assume the global memory to consist of $G(1)$, $G(2)$,... and the ith processor's local memory of $L_i(1)$, $L_i(2)$,.... This global write is simulated by two new statements:

for $i = 1, \ldots, p(n)$ **pardo** $w_{ij_i} := L_i(k_i)$
for $j = 1, \ldots, q(n)$ **pardo** $m_j := w_{\mathit{read\text{-}owner}(j,n),j}$

The first statement is carried out by processors $P_1, \ldots, P_{p(n)}$ and the second one by processors $W_1, \ldots, W_{q(n)}$. Analogously, a global read of all processors can be written as

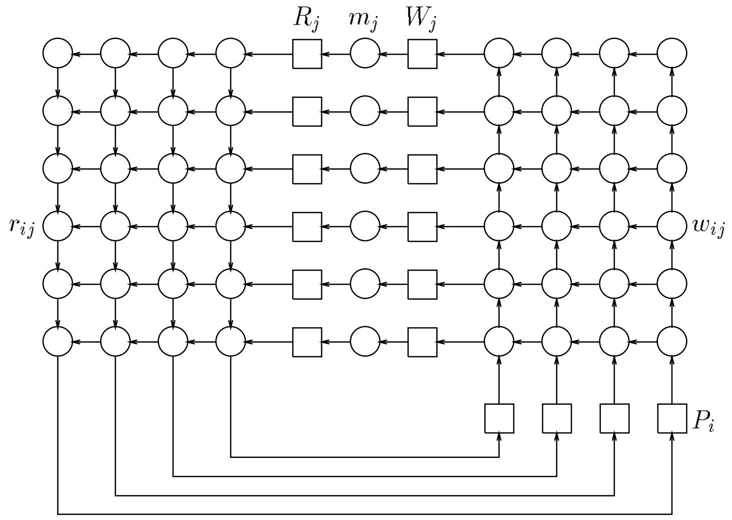

Figure 4: Simulation of generalized OROW-PRAMs by restricted ones

for $i = 1, \ldots, p(n)$ **pardo** $L_i(k_i) := G(j_i)$

and is simulated by

for $j = 1, \ldots, q(n)$ **pardo** $r_{write\text{-}owner(j,n),j} := m_j$
for $i = 1, \ldots, p(n)$ **pardo** $L_i(k_i) := r_{ij_i}$.

This time the first statement is executed by $R_1, \ldots, R_{q(n)}$ and the second one by $P_1, \ldots, P_{p(n)}$.

The construction given by now is capable to simulate an OROW-PRAM generalized according to G1, G2, and G6 (see also Figure 4). We will now observe how to modify this construction to allow generalizations G3, G4, and G5, too.

G3 "Limited multiple ownership": Instead of one single w_{ij}, now processor W_j has to check $O(1)$ memory cells $w_{i_1 j}$, $w_{i_2 j}$,...,$w_{i_k j}$. Similarly, R_j has to write into $O(1)$ memory cells $r_{i_1 j}$, $r_{i_2 j}$,...,$r_{i_k j}$. This increases the computation time only by a constant factor.

G4 "Ill behaved programs I": We provide each memory cell w_{ij} with an extra processor that checks if someone, i.e., P_i, writes into w_{ij}. Since the owner functions are at least OROW-TIME($\log n$) computable, it can be checked if this write attempt was illegal. Also, each memory cell r_{ij} is provided with an extra processor that writes a special "forbidden"-symbol into r_{ij} if $\textit{read-owner}(j,n) \neq i$. When processor P_i reads from r_{ij}, then this "forbidden"-symbol will be read, if this read attempt violates the owner restriction of the simulated PRAM P. In this way, violations of both owner restrictions can be detected.

G5 "Ill behaved programs II": The standard simulation for generalizations G1, G2, and G6 does already ignore illegal write attempts. By the method of "forbidden"-symbols illegal read attempts can be detected and, after detection, be ignored.

R3: Restriction R3 holds since we assumed that there are no instructions for indirect access to local memory. If such instructions exist, R3 is also possible: Instead of local memory cells, global memory cells can be used (the read- and write-owner of these memory cells will be the same processor). Only $O(1)$ local memory cells are necessary to establish indirect memory access and arithmetic and binary operations. □

Further Results On Owner-Read PRAMs

PRAMs using concurrent-write memory access, i.e., CRCW-, CREW-, and CROW-PRAMs have been intensively investigated in many papers. Also, EREW-PRAMs are one of the most often used models. In this paper we examined particularly OROW-PRAMs, the most realistic model, if communication capabilities are considered. However, there are also

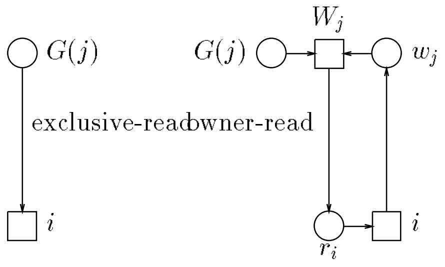

Figure 5: Simulating exclusive read by exclusive write and owner read

the "exotic" combinations ORCW, ERCW, OREW, and EROW. These PRAM models were taken into account only rarely.

Equality could not be proved between any two of the "classical" PRAM models, i.e., CRCW-, CREW-, EREW-, and CROW-PRAMs. (Due to the lower bounds obtained by Cook, Dwork, and Reischuk the class of languages recognized by CREW-PRAMs in *constant time* is a proper subclass of that recognized by CRCW-PRAMs in *constant time*, see [CDR86].) It is, however, possible to simulate exclusive-read by owner-read and exclusive-write:

Theorem 14 *For $k \geq 1$ it holds:*

1. EREW-TIME($\log^k n$) = OREW-TIME($\log^k n$)
2. ERCW-TIME($\log^k n$) = ORCW-TIME($\log^k n$)

Proof. As suggested above we simulate each exclusive read by an exclusive write followed by an owner read. In order to do so, we provide each processor i with a special memory cell r_i and each memory cell $G(j)$ with an processor W_j and another memory cell w_j. The read-owner of r_i is processor i and the read-owner of $G(j)$ and w_j is W_j.

If processor i reads from memory cell $G(j)$, we will simulate this exclusive read as follows: In the first step, processor i *writes* the address

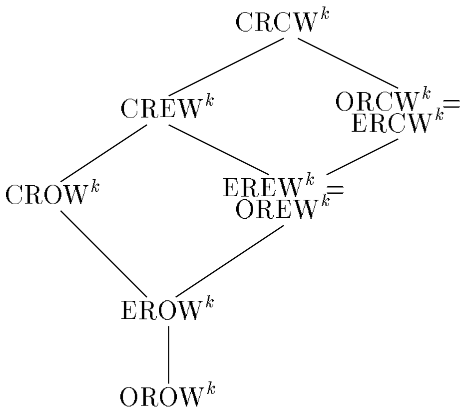

Figure 6: Relations between several types of PRAMs

of r_i into w_j. This is an exclusive write since we simulate an exclusive read, i.e., only one processor attempts to read from $G(j)$. In the second step processor W_j reads from $G(j)$ and w_j, such receiving the contents of $G(j)$ and the address of r_i. Then W_j writes the value of $G(j)$ into r_i (this is again an *exclusive* write).

In the third and final step, processor i simply reads from r_i instead directly from $G(j)$ (see also Figure 5).

□

We will write this simulation also in a more formal way. Let $L_i(1)$, $L_i(2)$,... be the local memory locations of the ith processor and $G(1)$, $G(2)$,... all global memory cells. We assume that $q(n)$ global memory locations are used and $p(n)$ processors. A simultaneous exclusive read of all processors from global memory could be written as

for $i = 1, \ldots, p(n)$ **pardo** $L_i(k_i) := G(j_i)$.

The simulation of this global read can be done in three steps on an

OREW-PRAM using additional global memory cells r_i, $i = 1, \ldots, p(n)$ and w_j, $j = 1, \ldots, q(n)$:

for $i = 1, \ldots, p(n)$ **pardo** $W_{j_i} := i$
for $j = 1, \ldots, q(n)$ **pardo** $r_{w_j} := G(j)$
for $i = 1, \ldots, p(n)$ **pardo** $L_i(k_i) := r_i$

The first and third statement are carried out by regular processors and the second statement by additional processors $W_1, \ldots, W_{q(n)}$.

We assumed that every processor attempts to write and that it is written into every memory cell. This is not true in general. However, our simulation works also correctly if it is not written into every memory cell. Assume $S \subseteq \{1, \ldots, p(n)\}$ is the set of processors that want to read a global memory cell:

for $i \in S$ **pardo** $L_i(k_i) := G(j_i)$.

Obviously, this limited global read can be simulated by restricting the simulating statements accordingly:

for $i \in S$ **pardo** $W_{j_i} := i$
for $j = 1, \ldots, q(n)$ **pardo** $r_{w_j} := G(j)$
for $i \in S$ **pardo** $L_i(k_i) := r_i$

Conclusion

We introduced a new PRAM model for parallel computation, which seems to be useful. We always claimed the number of processors to be bounded by some polynomial, but acted quite generous within this bound: It takes only $O(n/\log n)$ processors to do list ranking for an EREW-PRAM, for instance, but our OROW algorithm requires $O(n^2)$ processors, if no unit-cost multiplication is available. A question that arises in this context is, whether it is possible and how to reduce the number of processors required. In particular, are there optimal OROW algorithms for list ranking and tree contraction?

Another question is, whether OROW-PRAMs can be characterized in terms of other parallel models. At first sight, one can be tempted to assume that NC^1 = OROW-TIME($\log n$), since data flow in OROW-PRAMs is quite similar to circuits with bounded fan-in and fan-out. But this can't be true, unless $NC^1 = LOGSPACE$, because of $LOGSPACE \subseteq$ OROW-TIME($\log n$). In this context it could be very useful to exhibit natural complete languages for OROW-TIME($\log n$).

One result of this paper is that exclusive reads can be simulated by exclusive writes and owner reads yielding EREW-TIME($\log^k n$) = OREW-TIME($\log^k n$) and ERCW-TIME($\log^k n$) = ORCW-TIME($\log^k n$), i.e., one of the most popular models of parallel computation (EREW-PRAM) can be simulated by a significantly weaker one without loss of time! This was quite surprising since no similar result holds between CREW- and CROW-PRAMs. Maybe we can conclude from this that write ability is more determining for a PRAM than its read ability. In this context it would also be interesting whether EROW-TIME($\log^k n$) = EREW-TIME($\log^k n$) holds or maybe even EROW-TIME($\log^k n$) = OROW-TIME($\log^k n$), or are there reasons for strict inclusions between these classes? Are there any problems in ORCW-TIME($\log^k n$) that use owner-read, concurrent-write memory access in a natural way?

Acknowledgment This research was launched by Torben Hagerup, Inga Niepel, Klaus-Jörn Lange, and the author during a visit of Torben Hagerup at the Institut für Informatik, Technische Universität München. I like to thank all of them for many helpful discussions and comments on this paper.

References

[ADKP89] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[CDR86] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[CKS81] A. K. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[Coo85] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CV88] R. Cole and U. Vishkin. Methodological parallel evaluation of expression trees. *Algorithmica*, 3:329–346, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, 1989.

[DR86] P. Dymond and W. L. Ruzzo. Parallel RAMs with owned global memory and deterministic language recognition. In *Proc. of 13th International Colloquium on Automata, Languages and Programming*, number 226 in Lecture Notes in Computer Science, pages 95–104. Springer, 1986.

[Fic83] F. E. Fich. New bounds for parallel prefix circuits. In *Proc. of 15th ACM Symposium on Theory of Computing*, pages 27–36, 1983.

[FW78] S. Fortune and J. Willie. Parallelism in random access machines. In *Proc. of 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GMT87] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in EREW model. In *Proc. Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation*, 1987.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In

Symposium on Foundations of Software Technology and Theoretical Computer Science, pages 453–469. Springer Verlag, 1986.

[KD87] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. Extended abstract, The Johns Hopkins University, 1987.

[KR87] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California at Berkeley, California 94720, March 1987. To appear in J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Elsevier.

[KRS] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10):964–968.

[Lan90] K.-J. Lange. Unambiguity of circuits. In *Proc. of 5th Structure in Complexity Conf.*, pages 130–137, 1990.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[LPV81] G. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. on Computers*, C-30:93–100, 1981.

[MR85] G. L. Miller and J. Reif. Parallel tree contraction and its applications. In *Proc. of 26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[Ruz80] W. L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[Ruz81] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[Sni82] M. Snir. On parallel searching. In *Proc. SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 242–253, Ottawa, Canada, 1982.

[SS89] F. Springsteel and I. Stojmenovic. Parallel general prefix computations with geometric, algebraic, and other applications. In J. Csirik, J. Demetrovics, and F. Gecseg, editors, *Proc. Fundamentals of Computation Theory*. Springer, August 1989.

[Sud78] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the ACM*, 25:405–414, 1978.

[SV84] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, May 1984.

[TV85] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing*, 14:862–874, 1985.

[Ven87] H. Venkateswaran. Properties that characterize LOGCFL. In *Proc. of 19th ACM Symposium on Theory of Computing*, pages 141–150, 1987. (to appear in JCSS).

[Wyl81] J. C. Wyllie. *The Complexity of Parallel Computations*. Ph.d. dissertation, Cornell University, Computer Science Department, Ithaca, NY, 1981.