BRICS

Basic Research in Computer Science

BRICS LS-95-2 S. Skyum: Introduction to Parallel Algorithms

Introduction to Parallel Algorithms

Sven Skyum

BRICS Lecture Series **LS-95-2**

ISSN 1395-2048 **June 1995**

See back inner page for a list of recent publications in the BRICS Lecture Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Introduction to Parallel Algorithms

Sven Skyum

First published 1995
Second Edition 1996

Sven Skyum
BRICS[1]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

[1] **B**asic **R**esearch **I**n **C**omputer **S**cience,
Centre of the Danish National Research Foundation.

Preface

The material in this note is used as an introduction to parallel algorithms in a third year course on algorithms and complexity theory in Aarhus. Basic data structures and algorithms including sorting and searching are introduced to the students the first year. For the analysis of algorithms the unit cost model was used. The RAM were not introduced, the analysis was based on the number of operations in a (programming) language and corresponds to unit cost at a RAM.

This note covers an introduction to various PRAM's, presents Brents scheduling principle and various algorithms such as prefix, merging and sorting building up to a general method of simulating a CRCW-PRAM on a EREW-PRAM.

Two weeks are spent on the subject which corresponds to a total of six 45 minutes lectures.

The first version of the note was written in 1993 and was inspired by a note written by Peter Bro Miltersen the year before. Some of the problems originate from it. The note has since undergone revisions each year. Some of them have been substantial.

References

[1] R.E. Ladner, M.J. Fischer: Parallel prefix computations, *J.ACM 27 (1980) pp. 831-838.*

[2] C.P. Kruskal, L. Rudolph, M. Snir: A complexity theory of efficient parallel algorithms. *TCS 71 (1990) pp. 95-132.*

[3] L.G. Valiant: General Purpose Parallel Architectures, *Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V. (1990) pp. 945-972.*

Contents

1 Models

The basic machine model behind the sequential algorithms presented up to now has been the RAM (Random Access Machine) in which each operation such as LOAD, STORE, JUMP, ADD, MULT, etc. is assumed to take one unit of time.

In this note we present algorithms to be executed on a multi-processor machine. In contrast to the sequential case there are numerous architectures for multi-processor machines and therefore many different models for parallel machines.

In multi-processor machines each processor p has a local memory which can only be accessed by the processor p. The difference in the architectures lies in the way communication between processors is handled. Communication (and computation) can be either *synchronous* or *asynchronous*. In synchronous computations we have a global clock, such that each processor executes the instructions synchronously.

In asynchronous computations there is no global clock, so communication must be handled through channels (as is known from the computer architecture course). Asynchronous computation is normally called *distributed computing* while synchronous computation is called *parallel computing*. This distinction is not sharp though.

We will consider synchronous computations in this note and refer to synchronous multi-processor machines as *parallel machines*. There are two groups of parallel machines. One is connection machines such as vector- or array-processors.

For those machines a fixed network of links or channels between processors exists and communication only takes place via those channels.

In the other group, machines have a shared memory which can be accessed by all processors.

We will concentrate on machines with a shared memory. They are the most powerful machines and therefore the easiest to program.

We model those machines by PRAMs (Parallel Random Access Machines). A PRAM consists of:

- A global random access memory where each register can hold an arbitrary integer. Initially the content of each register is 0.
- A set of processors $p_1, p_2, \ldots, p_n$. Each processor runs the same RAM program. The accumulator for a process is considered to be local. To

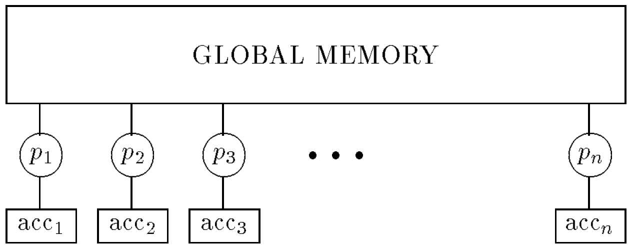

Figure 1: A PRAM

prevent the processors from all computing in exactly the same way, each has access to its identification number *id*.

The PRAM is synchronous; that is, all the processors follow a global clock and execute one instruction of the RAM program per clock signal. There is no direct communication between processors but two processors communicate if one of them reads what the other has written in the global memory.

Finally we allow the number of processors to depend on the size of the problem. This might seem unnatural, but we shall see that designing algorithms using a large amount of processors is very fruitful even if only few processors are present, while the opposite is not true.

It might happen that two or more processors attempt to access the same register in the memory concurrently. Various conventions can be used in this case. We will use the following convention.

- A processor is not allowed to load from a register at the same time as another processor stores to the same register.
- Two or more processors may load from the same register at the same time. Since no processor can store to the register at the same time, the result of the loads is well defined.
- Two or more processors may store to the same register at the same time. The content of the register after the stores is what the processor with the lowest *id* stored.

A PRAM settling read/write (load/store) conflicts by the above convention is called a CRCW-PRAM (Concurrent Read, Concurrent Write).

A weaker (more restrictive) model is the EREW-PRAM (Exclusive Read, Exclusive Write) where the convention is that no two processors may access the same register at the same time. Note that the two models represent two levels of abstraction. The CRCW-PRAM is easier to program while the EREW-PRAM is closer to real architectures.

By using many processors we will expect sub-linear running time for many algorithms. This would be impossible if there were a separate (sequential) input tape. We therefore assume that the input is present in the global memory initially. The output will be stored in the global memory as well.

2 Time, work and optimality

We illustrate some concepts about parallel algorithms by a simple example.

Given an array $A[1..n]$ of n numbers. Compute the maximum number. This can be done in numerous ways. We give two different sequential solutions (assume for the second solution that n is a positive power of two):

function smax1(A, n);
  $m := -\infty$;
  for $i := 1$ **to** n **do**
    $m := \max\{m, A[i]\}$
  od;
  smax1 $:= m$
end;

function smax2(A, n);
  for $i := 1$ **to** $n/2$ **do**
    $B[i] := \max\{A[2i-1], A[2i]\}$
  od;
  if $n = 2$ **then**
    smax2 := $B[1]$
  else
    smax2 := smax2($B, n/2$)
  fi
end;

Both algorithms take time $\mathrm{O}(n)$. While the first is highly sequential in nature and difficult to parallelise the other is an example of the *compress and iterate* paradigm which leads to natural parallelisations. The compression (to half size) happens in the **for**-loop and the iteration is the recursive call of smax. A parallelisation of this algorithm using n processors more or less presents itself, since the statements in the **for**-loop are independent and can be done in parallel. We write the parallel algorithm as follows:

function pmax(A, n)[$p_1, p_2, \ldots, p_{n/2}$];
  for $i := 1$ **to** $n/2$ **pardo**
    $p_i : B[i] := \max\{A[2i-1], A[2i]\}$
  od;
  if $n = 2$ **then**
    p_1 : pmax := $B[1]$
  else
    pmax := pmax($B, n/2$)[$p_1, p_2, \ldots, p_{n/4}$]
  fi
end;

The processors in brackets are the processors that are used to solve the problem at hand. A statement of the form $p_i : S$ means that p_i executes the statement S. Finally, if no processor is assigned to a statement, the processor with lowest *id* among those used to solve the problem executes the statement. Note that pmax is an EREW-PRAM algorithm.

If all processors know n initially, the *time* $t(n)$ for the algorithm satisfies

$$t(n) = \begin{cases} O(1) & \text{for } n = 2 \\ t(n/2) + O(1) & \text{for } n > 2 \end{cases}$$

which has solution $t(n) = \mathrm{O}(\log n)$.

The number of processors used is $p(n) = n/2$. The *work* done by the algorithm is defined to be $w(n) = p(n) \cdot t(n) = \mathrm{O}(n \log n)$. The work measures the time required to run the parallel algorithm on just one processor which in turn has to simulate each of the $p(n)$ involved processors.

If $w(n)$ is of the same order as the time for best known sequential algorithm, the parallel algorithm is said to be *optimal*. Thus pmax is not optimal. This is due to the fact that through the recursive calls only $n/4, n/8, \ldots, 2, 1$ processors are executing statements. Since we do not want to waste processors like that, we have the following design principle for parallel algorithms:

- Construct optimal algorithms to run as fast as possible.

This is equivalent to

- Construct optimal algorithms using as many processors as possible.

Note that if we have an optimal parallel algorithm which runs in time $t(n)$ using $p(n)$ processors then there exist optimal algorithms using $p'(n) < p(n)$ processors running in time $\mathrm{O}(t(n) \cdot p(n)/p'(n))$. Each of the $p'(n)$ "new" or "physical" processors can simulate $\lceil p(n)/p'(n) \rceil$ "old" or "virtual" processors. Each step in the original algorithm then takes $\mathrm{O}(p(n)/p'(n))$ steps to simulate in the new algorithm.

2.1 Brent's scheduling principle

Theorem 2.1 (**Brent's scheduling principle**) If a parallel computation consists of k phases, taking time $t_1, t_2, \ldots, t_k$ using $a_1, a_2, \ldots, a_k$ processors in phase $1, 2, \ldots, k$, then the computation can be done in time $\mathrm{O}(a/p + t)$ using p processors where $t = \sum t_i$, $a = \sum a_i t_i$, and p can be chosen arbitrarily.

Proof In the i'th phase the p "physical" processors have to simulate a_i "virtual" processors. Each of the "physical" processors has to simulate at most $\lceil a_i/p \rceil \leq a_i/p + 1$ "virtual" processors so phase i takes time at most $c(a_i/p + 1)t_i$ for some constant c (independent of i). All phases then take at most $c \cdot \sum((a_i/p) + 1)t_i = \mathrm{O}(a/p + t)$.

□

Note that the Theorem can only be used if we know the phases at "compile time". Otherwise it is not obvious how to handle the scheduling problem of assigning physical processors to virtual processors in the various phases.

Note also that the number of phases k does not have to be constant.

As an example we apply Brent's principle on pmax from the previous section.

The computation consists of $\log n$ phases running in constant time and using $n/2, n/4, \ldots, 1$ processors respectively. Thus with p processors we can do the computation in time $O(\log n + n/p)$. If we choose $p = O(n/\log n)$ we obtain the following result.

Theorem 2.2 *The maximum of n integers can be found in time $O(\log n)$ with $O(n/\log n)$ processors on an EREW-PRAM, which is optimal.*

It can be shown that no parallel algorithm, computing the maximum of n numbers, exists which runs in time $o(\log n)$ on an EREW-PRAM. In contrast it can be done on a CRCW-PRAM in constant time (see Problem 5.1).

Exercise 2.1 *Consider the divide and conquer algorithm to compute the sum of n elements.*

function sum$(A, n_1, n_2)[p_{n_1}, \ldots, p_{n_2}]$
  if $n_1 = n_2$ **then**
    sum := $A[n_1]$
  else
    $m := \lfloor \frac{n_1+n_2}{2} \rfloor$;
    sum := sum$(A, n_1, m)[p_{n_1}, \ldots, p_m]$+
        sum$(A, m+1, n_2)[p_{m+1}, \ldots, p_{n_2}]$
  fi
end;

Show that sum *for n elements can be computed in time $O(\log n)$ on an EREW-PRAM using $O(n \log n)$ processors.*

3 Merging and sorting

Before presenting algorithms for merging and sorting, we present a parallel solution to a problem which turns out to be very useful as a subroutine in many applications, in merging and sorting.

3.1 Prefix computations

Assume that we have an array $A[1..n]$ of numbers and we want to compute $B[1..n]$ such that $B[k] = \sum_{i=1}^{k} A[i]$ for $1 \leq k \leq n$. We call this *the prefix sum problem* and the function prefix$^+$.

A parallel solution is the following (again we assume that n is a power of two):

function prefix$^+(A, n)[p_1, p_2, \ldots, p_n]$;
  $p_1 : B[1] := A[1]$;
  if $n > 1$ **then**
    for $i = 1$ **to** $n/2$ **pardo**
      $p_i : C[i] := A[2i-1] + A[2i]$
    od;
    $D :=$ prefix$^+(C, n/2)[p_1, p_2, \ldots, p_{n/2}]$;
    for $i = 1$ **to** $n/2$ **pardo**
      $p_i : B[2i] := D[i]$;
    od;
    for $i = 2$ **to** $n/2$ **pardo**
      $p_i : B[2i-1] := D[i-1] + A[2i-1]$
    od
  fi
  prefix$^+ := B$
end;

The correctness of prefix$^+$ follows by observing that when the recursive call of prefix$^+$ has been executed, $D[k] = \sum_{i=1}^{2k} A[i]$ for $1 \leq k \leq n/2$. The algorithm runs in time $\mathrm{O}(\log n)$. The only important property of addition $(+)$ in connection with prefix$^+$ is that it is associative. For any other associative operation $\circ$ we could use the same algorithm with $+$ replaced by $\circ$.

Using Brent's scheduling principle we get the following Theorem:

Theorem 3.1 *If $\circ$ is an associative operation such that $x \circ y$ can be computed in time $\mathrm{O}(1)$ on a RAM, then prefix$^\circ$ can be computed in time $\mathrm{O}(\log n)$ on an EREW-PRAM using $\mathrm{O}(n/\log n)$ processors, which is optimal.*

Exercise 3.1 *Show that initialising an array $A[1..n]$ to the value of variable c can be done in time $\mathrm{O}(\log n)$ on an EREW-PRAM using $\mathrm{O}(n/\log n)$ processors.*

Exercise 3.2 *Given an array $A[1..n]$ of numbers. Construct an EREW-PRAM algorithm which computes $B[1..n]$ such that $B[j] = (k, x)$ where $x = \min_{i \le j}\{A[i]\}$ and $k = \max_{i \le j}\{i \mid A[i] = x\}$.*

3.2 Merge on a CRCW-PRAM

First we show how to merge two arrays $A[1..n]$ and $B[1..m]$ of numbers into an array $C[1..n+m]$ in time $\mathrm{O}(\log(n+m))$ using $n+m$ processors. Given an array $A[1..n]$ of numbers and a number x, the rank of x in A is

$$\mathrm{rank}(x, A, n) = \begin{cases} 0 & \text{if } x < A[1] \\ \max\{i \mid A[i] \le x,\ 1 \le i \le n\} & \text{otherwise} \end{cases}$$

$\mathrm{rank}(x, A, n)$ can be found sequentially by binary search in time $\mathrm{O}(\log n)$.

function merge1$(A, B, n, m)[p_1, p_2, \ldots, p_{n+m}]$
  for $i = 1$ **to** n **pardo**
    $IA[i] := \mathrm{rank}(A[i] - 1, B, m)$
    $C[i + IA[i]] := A[i]$
  od;
  $\boxed{B[IA[i]] < A[i] \le B[IA[i]+1], \text{ (if } B[0] \text{ equals } -\infty)}$
  for $i = 1$ **to** m **pardo**
    $IB[i] := \mathrm{rank}(B[i], A, n)$
    $C[i + IB[i]] := B[i]$
  od;
  $\boxed{A[IB[i]] \le B[i] < A[IB[i]+1], \text{ (if } A[0] \text{ equals } -\infty)}$
  merge1 $:= C$
end;

Since the assignment of processors to statements is obvious, it is left out for simplicity. It is left as an exercise to prove the correctness of merge1. The time is $\mathrm{O}(\log(n+m))$ since the calls of rank are the dominating factor.

Exercise 3.3 *Show the correctness of merge1.*

merge1 is not optimal so we want to improve on it. Brent's scheduling principle is of no help since, if $n = m$, half of the processors are busy all the time.

Therefore we have to be inventive to bring down the number of processors in use.

The idea behind the algorithm to be presented is first to find the positions in the output for every $\lceil \log n \rceil$ element in $A[1..n]$ and every $\lceil \log m \rceil$ element in $B[1..m]$ using the method of merge1. This can be done in time $O(\log(n+m))$ with $O(\frac{n+m}{\log(n+m)})$ processors. What is left to be done is to merge small lists together two by two. As we shall see, it can be done in time $O(\log(n+m))$ with $O(\frac{n+m}{\log(n+m)})$ as well.

Let $q : N^2 \rightarrow N$ be the function defined by

$$q(n,m) = \begin{cases} 2 & \text{if } n = m = 1 \\ \lceil \frac{n}{\log n} \rceil + 1 & \text{if } n > 1 \text{ and } m = 1 \\ \lceil \frac{m}{\log m} \rceil + 1 & \text{if } n = 1 \text{ and } m > 1 \\ \lceil \frac{n}{\log n} \rceil + \lceil \frac{m}{\log m} \rceil + 1 & \text{if } n, m > 1 \end{cases}$$

Exercise 3.4 *Prove that* $q(n,m) = O(\frac{n+m}{\log(n+m)})$.

Let $l : N \rightarrow N$ be defined by

$$l(n) = \begin{cases} 1 & \text{if } n = 1 \\ \lceil \log n \rceil & \text{otherwise} \end{cases}$$

function merge$(A, B, n, m)[p_1, p_2, \ldots, p_{q(n,m)}]$
  $a := l(n); b := l(m); na := \lfloor n/a \rfloor; mb := \lfloor m/b \rfloor;$

First we compute the ranks of every $\lceil \log n \rceil$'th element of A with respect to B and the rank of every $\lceil \log m \rceil$'th element of B with respect to A. Time: $O(1)$.

  for $i = 1$ **to** na, $j = 1$ **to** mb **pardo**
    $IA[i] :=$ rank$(A[i * a] - 1, B, m)$,
    $IB[j] :=$ rank$(B[j * b], A, n)$
  od;
  $IA[0] := 0;$

Now
$$B[IA[i]] < A[a \cdot i] \leq B[IA[i]+1]]$$
and
$$A[IB[j] \leq B[b \cdot j] < A[IB[j]+1].$$
Thus we know the position in the result of na and mb elements in A and B. The corresponding elements of C can be set to the values of those elements.
Time: O(1).

for $i = 1$ **to** na, $j = 1$ **to** mb **pardo**
  $C[IA[i] + a * i] := A[a * i]$, $C[IB[j] + b * j] := B[b * j]$
od;

What remains to be done is to fill in the segments of elements of C that are not filled by the statement above. These segments are no longer than $a + b = \mathrm{O}(\log(n+m))$ since a subarray of C of length $a + b + 1$ must contain at least two elements that are already set to a value.
The segment starting at $IA[i] + a \cdot i + 1$ $(1 \leq i \leq na)$ is filled by merging A starting at $a \cdot i + 1$ and B starting at $IA[i] + 1$ into it. Segments starting at $IB[j] + b \cdot j + 1$ $(1 \leq j \leq nb)$ are filled analogously. Finally the two leftmost segments in A and B are to be merged. Note that one of them might be empty. Segments are filled in by sequentially merging elements from A and B.
We use the procedure seqmerge(i, j) to do that. It works on A, B and C. It merges $A[i+1], A[i+2], \ldots$ and $B[j+1], B[j+2], \ldots$ into $C[i+j+1], C[i+j+2], \ldots$ until the next element in C is already set.
Time: $\mathrm{O}(\log(n+m))$.

  for $i = 0$ **to** na, $j = 1$ **to** mb **pardo**
    seqmerge$(a * i, IA[i])$, seqmerge$(IB[j], b * j])$
  od
od
end;

The above algorithm is optimal and implies the following Theorem.

Theorem 3.2 Merging two lists with n and m elements can be done in time $\mathrm{O}(\log(n+m))$ using $\mathrm{O}(\frac{n+m}{\log(n+m)})$ on a CRCW-PRAM.

Since the CRCW-PRAM is less realistic than the EREW-PRAM, we would like the algorithm to be transformed into an EREW-PRAM algorithm. There is a general method for doing that, as we shall see in section 4.

Exercise 3.5 *Give an optimal algorithm for sorting on a CRCW-PRAM based on merge.*

3.3 Bucket- and radix-sort

Let $e_1, e_2, \ldots, e_n$ be n elements with integer keys $k_1, k_2, \ldots, k_n$, where $0 \leq k_i < n$ for $1 \leq i \leq n$. Sorting the elements according to their keys can be done sequentially by first throwing the elements into n buckets according to their keys and then concatenating the elements in the buckets in increasing order.

E contains the elements and K the keys and the buckets are organised as lists (L).

```
function sbucketsort(E, K, n)
   initialise L[0..n − 1] to empty lists;
   for i := 1 to n do
      append E[i] to the list L[K[i]]
   od;
   concatenate the n lists in L into A;
   sbucketsort := A
end;
```

The running time is $O(n)$ and the algorithm is *stable*. That is, elements with the same key occur in the same order in the output as in the input.

If integer division can be done in constant time then the time for sorting n elements with integer keys bounded by a polynomial in n is also linear.

If m is an integer between 0 and $n^c - 1$ we can write m in n-ary notation as $\sum_{i=0}^{c-1} m_i \cdot n^i$ where $0 \leq m_j < n$ for $0 \leq j < c$.

The idea behind the algorithm below is to do sbucketsort c times using the c n-ary digits as keys, one by one starting with the least significant. The correctness then follows from the stability of sbucketsort.

function sradixsort(E, K, n, c);
  if $c = 1$ **then**
    sradixsort := sbucketsort(E,K,n)
  else
    for $j := 1$ **to** c **do**
      for $i := 1$ **to** n **do**
        $R[i] := K[i]$ **mod** n;
        $K[i] := K[i]$ **div** n
      od;
      $(E, K) :=$ sbucketsort($(E, K), R, n$)
    od;
  fi;
  sbucketsort:= E
end;

(E, K) denotes an array where the i'th element has two fields $E[i]$ and $K[i]$. The proof of correctness of sbucketsort is left as an exercise. The running time is $O(n)$ because c is a constant.

Exercise 3.6 *Prove the correctness of sbucketsort.*

3.4 Bucket- and radix-sort on an EREW-PRAM

We start by presenting an algorithm which sorts elements with small keys. Let $m < n$. $r = m \cdot \lceil \frac{n}{m} \rceil = O(n)$ processors will be used to sort n elements with keys bounded by m. Most of the time, less than r processors are used, so we will apply Brent's scheduling principle afterwards.

A sequential algorithm sbs is used. sbs(E, K, n) takes n elements E with keys K in the range 0 through $n - 1$ and places them in buckets $0, 1, \ldots, n - 1$. Furthermore it computes for each key the number of elements with that key. It can all be done sequentially in time $O(n)$.

function pbucketsort(E, K, n, m)[$p_1, \ldots, p_{m \cdot \lceil n/m \rceil}$];
  $b := \lceil n/m \rceil$;
  divide E into b blocks $B_1, B_2, \ldots, B_b$ of size m (the last possibly smaller) and K into $K_1, K_2, \ldots, K_b$ correspondingly;
  for $i = 1$ **to** b **pardo**
    sbs(B_i, K_i, m);
    for $j = 0$ **to** $m - 1$ **pardo**

$C[j * b + i]$:= number of elements in block B_i with key j

$\boxed{\text{these values were calculated by sbs above}}$

od
od;
$C := \text{prefix}^+(C, m * b)[p_1, p_2, \ldots, p_{m \cdot b}]$

$C[j \cdot b + i]$ is now the total number of elements with keys $0, 1, \ldots, j-1$ plus the number of elements in blocks $B_1, B_2, \ldots, B_i$ with key j. Therefore the elements in block $i+1$ with key j are to be placed in $E[C[j \cdot b + i] + 1..?]$. This is done sequentially for each block.

for $i = 1$ **to** b **pardo**
place the elements in B_i in their right positions in E
od
fi
pbucketsort := E
end;

The correctness proof is once again left for an exercise.

Exercise 3.7 *Prove the correctness of pbucketsort.*

Note that to be able to divide E into blocks efficiently, m has to be known to all the processors involved in the job. Broadcasting m to b processors takes optimal time $O(\log b)$ (See Exercise 3.1).

pbucketsort runs in $2 + 2m + O(\log n)$ phases (see below) each taking time $O(1)$. The phases are:

- $O(\log b)$ phases to divide E and K into blocks.
  $1, 2, 4, \ldots$ processors are used.
- m phases to do $\text{sbs}(B_i, K_i, m)$.
  b processors are used.
- One phase to assign values to C.
  $m \cdot b$ processors are used.
- $O(\log n)$ phases to do prefix$^+$ on C.
  $\frac{1}{2}m \cdot b, \frac{1}{4}m \cdot b, \frac{1}{8}m \cdot b, \ldots$ processors are used (see subsection 3.1).

- m phases to perform the last parallel **for**-loop of the algorithm.
  b processors are used.

Using Brent's scheduling principle we get for $\log n \leq m$ that pbucketsort runs in time $\mathrm{O}(m + \log n + a/p) = \mathrm{O}(m + a/p)$ using p processors where $a = b(1 + 4m) = \mathrm{O}(n)$. If we use $p = \lceil n/m \rceil$ processors we obtain an EREW-PRAM algorithm running in time $\mathrm{O}(m)$ since.

We have proved the following theorem.

Theorem 3.3 *For all* $\log n \leq m$, *n elements with integer keys between 0 and* $m-1$ *can be sorted in time* $\mathrm{O}(m)$ *on an EREW-PRAM using* $\lceil n/m \rceil$ *processors, which is optimal when* $m \leq n$.

Corollary 3.1 *For all* $0 < \epsilon < 1$ *and constants c, n elements with integer keys between 0 and* $n^c - 1$ *can be sorted in time* $\mathrm{O}(n^\epsilon)$ *on an EREW-PRAM using* $\lceil n^{1-\epsilon} \rceil$ *processors, which is optimal.*

Proof For a fixed ϵ it follows from Theorem 3.3 that n elements with integer keys less than n^ϵ can be sorted in time $\mathrm{O}(n^\epsilon)$ using $\lceil n^{1-\epsilon} \rceil$ processors. Since pbucketsort is stable, we can iterate pbucketsort $\lceil c/\epsilon \rceil$ times to sort n element with integer keys less than n^c in time n^ϵ, in analogy with what we did in sradixsort.

4 Simulation of CRCW-PRAMs on EREW-PRAMs

To simulate a CRCW-PRAM on an EREW-PRAM we have to resolve read and write conflicts. We may assume that the CRCW-PRAM works in "macro" steps, consisting of a read from global memory followed by an internal computation and finally a write to the global memory. If the algorithm to simulate does not work in macro steps, we can make it do so by at most tripling the running time.

Let $\mathcal{A}$ be a CRCW-PRAM algorithm running in time $\mathrm{O}(t(n))$ using $p(n)$ processors. Assume that there exists a constant c such that the addresses used by $\mathcal{A}$ are bounded by $p(n)^c$.

Below we show how to simulate $\mathcal{A}$ by an EREW-PRAM algorithm using the same number of processors. We do this by showing how to resolve possible read conflicts. Write conflicts are resolved in a similar fashion.

One read step in $\mathcal{A}$ is implemented on an EREW-PRAM as follows:

1. If processor j in $\mathcal{A}$ reads cell r_j, then processor j sets $R[j] := (j, r_j)$.
2. Sort R using the second elements as keys.
3. If $R[j]_2 \neq R[j-1]_2$ are different, then processor j assigns the content of cell $R[j]_2$ to $C[j]$, otherwise it assigns undefined ($\perp$) to $C[j]$.

   > If $R[j]_2 \neq R[j-1]_2$ ($R[1]_2 \neq R[0]_2$) then $C[j]$ contains the value to be read by processors $R[j]_1, R[j+1]_1, \ldots, R[m-1]_1$, where m is the smallest index greater than j such that $R[m]_2 \neq R[m-1]_2$. Broadcasting $C[j]$ to $C[j+1], \ldots, C[m-1]$ is done by the following prefix computation.

4. Compute $C :=$ prefix$^\circ(C)$, where $\circ$ is the associative (see Exercise 4.1 below) operator given by
   $$x \circ y = \begin{cases} x & \text{if } y = \perp \\ y & \text{otherwise} \end{cases}$$

   > Now $C[j]$ contains the value to be read by processor $R[j]_1$, for $1 \leq j \leq p(n)$.

5. Processor j computes $D[R[j]_1] := C[j]$.

   > $D[j]$ now contains the value to be read by processor j.

6. Processor j reads $D[j]$.

Let $0 < \epsilon < 1$ be a constant. Then 1 through 6 can be done in time $\mathrm{O}(p(n)^\epsilon)$ using $\lceil p(n)^{1-\epsilon} \rceil$ processors. 1, 3, 5 and 6 take time $\mathrm{O}(1)$ with $p(n)$ processors, so those steps can also be done in time $\mathrm{O}(p(n)^\epsilon)$ with $\lceil p(n)^{1-\epsilon} \rceil$ processors.

That 2 can be done in time $\mathrm{O}(p(n)^\epsilon)$ with $\lceil p(n)^{1-\epsilon} \rceil$ processors follows by Corollary 3.1 since the keys are bounded by $p(n)^c$. Finally $\lceil p(n)^{1-\epsilon} \rceil = \mathrm{O}(\frac{p(n)}{\log(p(n))})$ implies that the prefix sum of $p(n)$ elements can be computed in time $\mathrm{O}(p(n)^\epsilon)$ with $\lceil p(n)^{1-\epsilon} \rceil$ processors (4).

By observing that each computation step in $\mathcal{A}$ can be simulated in time $\mathrm{O}(p(n)^\epsilon)$ by $\lceil p(n)^{1-\epsilon} \rceil$ processors, we obtain the following Theorem.

Theorem 4.1 *For any $0 < \epsilon < 1$ the following holds: If a problem can be solved by a CRCW-PRAM algorithm $\mathcal{A}$ in time $\mathrm{O}(t(n))$ using $p(n)$ processors and for some constant $c > 0$ the addresses of cells accessed by $\mathcal{A}$ are bounded by $p(n)^c$, then the problem can be solved on an EREW-PRAM in time $\mathrm{O}(t(n)p(n)^\epsilon)$ using $\lceil p(n)^{1-\epsilon} \rceil$ processors.*

Exercise 4.1 *Prove that the operator $\circ$ in 4 above is associative.*

Exercise 4.2 *Show how to simulate a write step in $\mathcal{A}$.*

Theorem 4.1 is an example of the *parallel slackness phenomenon*:

- Powerful (maybe unrealistic) parallel machines can be simulated on weaker (more realistic) parallel machines *without loss of work* but with fewer processors.

Another example is that an EREW-PRAM can be simulated without loss of work on certain processor networks which can be built without problems. The parallel slackness phenomenon legitimates designing fast optimal algorithms running on a PRAM. Those algorithms can then be simulated optimally on realistic machines but with fewer processors. The simulation could (or ought to) be part of the operating systems for the realistic machines.

5 Problems

Problem 5.1

a) Show that the maximum of n numbers can be found in time $\mathrm{O}(1)$ on a CRCW-PRAM. (Hint: More than n processors may be used.)

b) Show that it can be done in time $\mathrm{O}(\log\log n)$ with n processors. (Hint: Divide the elements into $\sqrt{n}$ groups.)

c) Show that it can be done in time $\mathrm{O}(\log\log n)$ with $\mathrm{O}(\frac{n}{\log\log n})$ processors.

d) Show that for all $\epsilon > 0$, it can be done in time $\mathrm{O}(1)$ using $\mathrm{O}(n^{1+\epsilon})$ processors.

Problem 5.2 *Construct an EREW-PRAM algorithm for matrix multiplication of $n \times n$ matrices, running in time $\mathrm{O}(\log n)$ using $\mathrm{O}(n^3/\log n)$ processors.*

Problem 5.3 *Let $L \subseteq \sum^*$ be a regular language. Show that the membership problem for L (testing whether $a_1a_2\ldots a_n \in L$) can be done in time $\mathrm{O}(\log n)$ on an EREW-PRAM using $\mathrm{O}(n/\log n)$ processors. (Hint: Use a prefix computation.)*

Problem 5.4 *Let M be a finite set and $\circ$ a binary operation on M. Show that prefix$^\circ$ can be computed in time* $\mathrm{O}(\log n)$ *on an EREW-PRAM using* $\mathrm{O}(n/\log n)$ *processors.*
Hint: Consider the functions $f_a : M \to M$ for $a \in M$ given by $f_a(b) = b \circ a$ and use the fact that composition of functions is associative.

Problem 5.5 *Construct an EREW-PRAM algorithm implementing procedure split, with running time* $\mathrm{O}(\log n)$ *using* $\mathrm{O}(n/\log n)$ *processors:*

procedure split*(A, n, x)*

Input: $A[1..n], x$

Output: $A[1..n], n_1, n_2$ *such that* $A[j] < x$ *for* $1 \le j \le n_1$, $A[j] = x$ *for* $n_1 < j < n_2$ *and* $A[j] > x$ *for* $n_2 \le j \le n$ *and the elements of A are a permutation of the elements of A before executing the procedure*

end;

Problem 5.6 *Give a parallel version of quicksort.*

Recent Publications in the BRICS Lecture Series

LS-95-2 **Sven Skyum.** ***Introduction to Parallel Algorithms*****. June 1995. viii+16 pp.**

LS-95-1 **Jaap van Oosten.** ***Basic Category Theory*****. January 1995. vi+75 pp.**