Parallel Processing Letters
© World Scientific Publishing Company

FAST PARALLEL PERMUTATION ALGORITHMS

TORBEN HAGERUP*
Max-Planck-Institut für Informatik
66123 Saarbrücken, Germany

and

JÖRG KELLER†
Universität des Saarlandes, Fachbereich 14 Informatik
Postfach 151150, 66041 Saarbrücken, Germany

Received (received date)
Revised (revised date)
Communicated by (Comm)

ABSTRACT

We investigate the problem of permuting n data items on an EREW PRAM with p processors using little additional storage. We present a simple algorithm with run time $O((n/p)\log n)$ and an improved algorithm with run time $O(n/p+\log n \operatorname{loglog}(n/p))$. Both algorithms require n additional global bits and $O(1)$ local storage per processor. If prefix summation is supported at the instruction level, the run time of the improved algorithm is $O(n/p)$. The algorithms can be used to rehash the address space of a PRAM emulation.

Keywords: Parallel Algorithms, Permutations, Shared Memory, Rehashing

1. Introduction

Consider the task of permuting n data items on an EREW PRAM with $p \leq n$ processors according to a permutation π given in the form of a constant-time "black-box" program. The task is trivial if n additional (global or local) memory cells are available: The items are first moved to the additional storage, with each processor handling $O(n/p)$ items, and then written back in permuted order. We restrict attention to the case in which only $O(1)$ additional memory cells per processor are available, but the positions holding the items can be marked as visited.

*Supported by the ESPRIT Basic Research Actions Program of the EU under contract No. 7141 (project ALCOM II).

†Supported by the Dutch Science Foundation (NWO) through NFI project ALADDIN under contract number NF 62–376 and by the German Science Foundation (DFG) through SFB 124. Part of the work was done when this author was at CWI, Postbus 94079, 1090 GB Amsterdam, The Netherlands.

An application of this problem is rehashing a hashed address space in a PRAM emulation. If both old and new hash functions are bijective maps of addresses to cells, then rehashing can be described as a permutation of the PRAM address space [1]. Examples are hash functions of the form $\pi(x) = ax \bmod m$, where m is the size of the shared address space, and a is chosen relatively prime to m. While the complete address space gets rehashed, there is no additional global space available. Moreover, processors usually only have small local memories to store additional information. These considerations motivate our decision to allow only $O(1)$ additional memory cells per processor.

The problem of permuting arrays has been investigated before, both in the setting of sequential computers and in the setting of PRAMs.

Knuth [2] describes a simple sequential algorithm that runs in time $O(n^2)$ and needs only one buffer and a few counters. He also analyzes the average run time and shows it to be $O(n \log n)$. Melville [3] presents a time/space tradeoff. If t additional bits are available, his algorithm runs in time $O(n^2/t)$. Fich, Munro and Poblete [4] give an algorithm with run time $O(n \log n)$ that needs only $O((\log n)^2)$ additional bits.

Aggarwal, Chandra and Snir describe an algorithm for the Block PRAM [5]. This is a PRAM where access to a block of b consecutive cells in the shared memory takes time $l+b$ (i.e., there is a start-up delay of l, followed by a unit delay for each cell read). Their algorithm runs in time $O(n/p)$ if $n = \Omega(lp^{1+\epsilon})$, for some fixed $\epsilon > 0$. However, they assume the permutation to be known in advance[a]. Chin [7] improves their result for rational permutations, i.e., permutations that can be expressed as permutations on the bit positions if numbers are given in binary representation. Keller [1] gives an algorithm for linear permutations, i.e., permutations of the form $\pi(x) = ax \bmod 2^u$, where a is odd. This algorithm runs in time $O(n/p + \log p)$ and requires $O(\log n)$ local memory cells per processor. All these parallel algorithms take advantage of some a priori knowledge of the permutation.

We consider the more general case in which the permutation is not fixed and we have no knowledge of its structure. Our work can be summarized as follows: We follow an idea from the simple $O(n)$-time sequential algorithm [3] and mark as visited the original positions of items that have been moved. This idea leads to a simple algorithm that runs in time $O((n/p) \log n)$ and needs only constant space per processor. By breaking the algorithm into $O(\log\log(n/p))$ phases and redistributing work to processors after each phase, we obtain an improved algorithm with run time $O(n/p + \log n \log\log(n/p))$. The overhead comes from executing a prefix summation after each phase. If prefix instructions can be executed in constant time, the run time improves to $O(n/p)$. By using a CRCW PRAM and faster load balancing strategies that do not rely on prefix summation, we obtain run times of $O(n/p + \log^* n \log\log(n/p))$ (randomized) and $O(n/p + (\log\log n)^3 \log\log(n/p))$ (deterministic). These algorithms, however, will be less practical.

[a] They do not state this, but otherwise they would need a preprocessing phase that includes the computation of switch positions in a Clos-Network [6], not to mention the space required to store this information.

The paper is organized as follows: We describe the simple algorithm in section 2 and analyze its complexity in section 3. In section 4 we show how to improve the simple algorithm. Possible further improvements are discussed in section 5.

2. The Basic Algorithm

The standard sequential algorithm to permute n data items according to a permutation π of the n positions holding the items works as follows [3]: Search for a position that has not yet been visited. Permute along the cycle starting in this position until you reach it again. Mark all positions that you visit. Continue until all positions have been visited. This algorithm requires time $O(n)$ to move all items and time $O(n)$ to search for unvisited positions. The time bound for searching is obtained by maintaining a pointer that keeps track of how far the array holding the items has been searched so far. As marked positions are never again unmarked, this finds all unvisited positions in time $O(n)$. The space requirements are a buffer, a pointer, and n bits to mark the positions.

We adapt the idea of marking visited positions and obtain a simple parallel algorithm for an EREW PRAM with p processors:

Without loss of generality assume that p divides n. We partition the n positions in p blocks of size $B = n/p$. Each processor P takes care of one block of B positions. P starts with an unvisited position x in its block and follows the cycle of π that starts in x, moving the items encountered as it goes along, until it meets a position x' that is already marked as visited. P now searches for another unvisited position in its block and continues. It terminates when all items in its block have been visited.

A processor can be in one of three states: either it is searching for an unvisited position in its block, or it is working on a cycle, or it is terminated.

If a processor is "searching", it examines the positions in its block to test whether they have been visited. It continues until it finds an unvisited position x or until it reaches the end of its block. In the first case, it marks the position as visited, picks up the item stored there, changes its own state to "on cycle", and moves to $\pi(x)$. In the latter case, it changes its state to "terminated". Each processor maintains a pointer into its block to keep track of how far it has searched so far. Hence, if it changes its state from "on cycle" to "searching" again, it does not have to start from the beginning of the block.

If a processor is "on cycle" and has reached position x, then its action depends on the state of x. If the position has not yet been visited, then the processor will pick up the item stored in x, mark x as visited, store in x the item it picked up in the previous iteration (or in the same iteration, if the processor just switched from "searching" to "on cycle"), and move to $\pi(x)$. If the position has already been visited, then the processor will store the previous item in x and change its state to "searching".

A processor may meet a visited position either because it reaches the end of the cycle (the position where it started in its own block) or because another processor started to work on the same cycle in this position. A position x therefore is in-

spected at most twice: Once by the processor assigned to its block, and once by a processor following the cycle containing x. In order to avoid an access conflict between these two cases, we split each iteration of the algorithm into two parts such that "searching" processors and processors "on cycle" proceed alternately.

The program for the basic algorithm is shown in figure 1. There, T denotes an upper bound on the maximum number of iterations; we will compute such an upper bound in section 3. Each processor has local variables `state`, `index`, `iptr` and `buffer`. The variable `state` defines the current state of the processor, `index` counts how far it has searched its block, `iptr` points to the currently visited position, and `buffer` is used to store data items temporarily. Global arrays are `visited` and `item`. The array `visited` contains the flags of all positions, and `item` stores the actual items.

An improvement in practical terms, omitted in the interest of clarity, would be to let even processors "on cycle" use the first part of each iteration to continue the search in their blocks for unvisited positions.

3. Analysis

We will now analyze the run time and the memory requirements of the basic algorithm. The results are described in Theorem 1.

Theorem 1 *The basic algorithm runs in time $O((n/p)\log n)$ and requires n global bits and $O(1)$ local memory cells per processor.*

Proof. In order to analyze the run time, we define a potential Φ as the sum of the lengths of the block parts that have not yet been searched plus the number of items that have not yet been moved to their final positions. It is easy to see that Φ never increases and that the algorithm may finish when $\Phi = 0$. Also $\Phi = 2n$ initially, since the sum of the block lengths is n and there are n items. For $i = 0, 1, 2, \ldots$, denote by Φ_i the value of Φ at the end of iteration i (for $i = 0$: before the first iteration) and by p_i the number of processors that have not terminated at that time. Clearly, $p_0 = p$.

For $i = 1, 2, \ldots$, each of the p_i processors that have not terminated after iteration i decreases Φ by at least one in iteration i; hence $\Phi_i \leq \Phi_{i-1} - p_i$. To see this, note that a "searching" processor decreases Φ by increasing its pointer `index` in line (1) of figure 1, while a processor "on cycle" moves one item to its final position in line (2). (A processor may decrease Φ by two in a particular iteration, namely if it switches from "searching" to "on cycle" in that iteration.)

Also $\Phi_i \leq 2Bp_i$, for $i = 1, 2, \ldots$, since there are only Bp_i positions in the blocks of the active processors that could be unsearched, and also at most Bp_i items that are not yet in their final positions. Then

$$\Phi_{i-1}/\Phi_i \geq (\Phi_i + p_i)/\Phi_i = 1 + p_i/\Phi_i \geq 1 + p_i/(2Bp_i) = 1 + 1/(2B) \quad .$$

It follows that $\Phi_i \leq \Phi_0 \cdot (1 + 1/(2B))^{-i}$, for $i = 1, 2, \ldots$, so that the number of iterations can be bounded by the smallest i with $\Phi_0 \cdot (1 + 1/(2B))^{-i} < 1$. This relation can be transformed into $i > \log(2n)/\log(1 + 1/(2B))$. Since each iteration

```
for i := 0 to p − 1 pardo (* initialization *)
  P_i.state := SEARCHING ; P_i.index := 0 ;
  for j := 0 to B − 1 do visited[iB + j] := 0 od
od ;
for t := 1 to T do (* iteration t *)
  for i := 0 to p − 1 pardo
    if P_i.state = SEARCHING then (* first part *)
      if P_i.index = B then P_i.state := TERMINATED
      else
        P_i.iptr := iB + P_i.index ;
(1)     P_i.index := P_i.index + 1 ;
        if visited[P_i.iptr] = 0 then
          visited[P_i.iptr] := 1 ;
          P_i.state := ON CYCLE ;
          P_i.buffer := item[P_i.iptr] ;
          P_i.iptr := π(P_i.iptr)
        fi
      fi
    fi ;

    if P_i.state = ON CYCLE then (* second part *)
(2)   P_i.buffer :=: item[P_i.iptr] ; (* exchange contents *)
      if visited[P_i.iptr] = 0 then
        visited[P_i.iptr] := 1 ;
        P_i.iptr := π(P_i.iptr)
      else
        P_i.state := SEARCHING
      fi
    fi
  od
od ;
```

Fig. 1. The basic algorithm

takes constant time, $\log(1+1/(2B)) = \Omega(1/B)$ and $B = n/p$, we obtain a run time of $O((n/p)\log n)$. From the description of the algorithm, it is clear that it needs n global bits, and that $O(1)$ local memory cells per processor are sufficient. □

4. An Improved Algorithm

The basic algorithm does not run in optimal time mainly because many processors could terminate early, causing the work load to be severely unbalanced. We improve the basic algorithm by breaking it into several phases and re-allocating processors to unvisited positions after each phase. The array of items is dynamically partitioned into *active* and *passive* blocks. In a passive block all positions have already been visited. Active blocks are split into smaller ones as the algorithm proceeds. In the beginning, the whole array forms one active block.

In phase i, for $i = 1, 2, \ldots$, we form p active blocks out of the remaining active blocks from the last phase. Then we execute $q_i = \lceil (49/9)2^{4-i}n/p \rceil$ iterations of the original algorithm. We proceed until fewer than $3p$ unvisited positions remain. It is easy to see that at this point the remaining items can be collected and moved in time $O(n/p + \log n)$.

The improvements of the new algorithm are summarized in the following Theorem 2.

Theorem 2 *The improved algorithm works in $O(\log\log(n/p))$ phases and runs in time $O(n/p + \log n \log\log(n/p))$. Its storage requirements are n global bits and $O(1)$ local memory cells per processor.*

Corollary 1 *The algorithm is optimal for $p = O\left(n/(\log n \log\log\log n)\right)$.*

We first show how to partition r remaining blocks into p blocks of roughly equal sizes if p is not a multiple of r. Then we prove Theorem 2.

4.1. Partitioning blocks

At the beginning of each phase, we want to partition the r blocks that were still active by the end of the previous phase into p new blocks. We do this by decomposing each remaining block into approximately p/r new blocks of roughly equal sizes. Suppose that each of the r blocks is of size at most s. We assume that $r \leq p/4$ and that $rs \geq p$. If we ignore any rounding problems, we obtain rs/p as the new block size. However, when we implement the permutation algorithm, we have to cope with the fact that p may not be a multiple of the number r of remaining blocks, and that s may not be a multiple of the number of new blocks to be formed out of an old block. Then the new block size will be larger than rs/p. Lemma 1 guarantees that the new block size will not be too large.

Lemma 1 *The partitioning described above can be done in such a way that the maximum size of the new blocks is at most $\lceil s/\lfloor p/r \rfloor \rceil$, which is less than $(7/3)\cdot rs/p$.*

We prove Lemma 1 using the following simple fact.

Lemma 2 *For any two integers u and v with $1 \leq v \leq u$, u can be written as a sum of v integers, each of which is $\lceil u/v \rceil$ or $\lfloor u/v \rfloor$. More precisely, $u = c \cdot \lceil u/v \rceil + (v - c) \cdot \lfloor u/v \rfloor$, where $c = u \bmod v$.*

Proof of Lemma 1. We apply Lemma 2 with $u = p$ and $v = r$ and see that we can split each remaining block into either $\lceil p/r \rceil$ or $\lfloor p/r \rfloor$ new blocks. To find the maximum size of the new blocks, we consider a block that is split into $\lfloor p/r \rfloor$ new blocks. We apply Lemma 2 with $u = s$ and $v = \lfloor p/r \rfloor$ and see that the maximum size of a new block is at most $s' = \lceil u/v \rceil = \lceil s/\lfloor p/r \rfloor \rceil$.

Using that $p/r - 1 < \lfloor p/r \rfloor$ and $\lceil s/v \rceil < s/v + 1$, we get $s' < rs/(p-r) + 1$. By the assumptions $r \leq p/4$ and $rs \geq p$, we have $s' < (4/3) \cdot rs/p + 1 \leq (4/3) \cdot rs/p + rs/p = (7/3) \cdot rs/p$. □

The computation to partition the remaining r active blocks into p blocks can be done as follows. First the r active blocks are numbered consecutively by means of a parallel prefix summation, and the value of r is broadcast to all processors. Then the processor with the ith active block writes the start address and the length of its block to position i of a global array A, for $i = 1, \ldots, r$. Each processor can compute locally from which remaining block j it will receive its new block (cf. Lemma 2). Using the information in $A[j]$, the processor then determines the start address and the length of its new block. Concurrent access to A can be avoided by implementing the reading from A as a generalized parallel prefix summation known as segmented broadcasting.

Each prefix summation requires $O(p)$ global memory cells. However, these cells can be "made local" by copying $O(p)$ global cells to local memories in $O(1)$ time and restoring them after the prefix summation.

4.2. Analysis of the improved algorithm

As in section 3, we denote by Φ the sum of the lengths of the block parts that have not yet been searched plus the number of items that have not yet been moved to their final positions. Let T be the number of stages necessary to reduce Φ to at most $3p$, and denote by Φ_i the value of Φ at the end of phase i, for $i = 0, 1, \ldots, T$; by definition, $\Phi_i > 3p$ for all $i < T$. Let B_i be the maximum block size in phase i, for $i = 1, 2, \ldots, T$. Arguing as in section 3, one can see that for $i = 1, 2, \ldots, T$, we have $\Phi_{i-1} \leq 2B_i p + p$; the extra term of p accounts for the fact that a processor may hold an item picked up in the previous phase. In order to prove Theorem 2, we show that the block size shrinks very fast as the algorithm proceeds. This is formalized in Lemma 3.

Lemma 3 *For $i = 1, 2, \ldots, T$, the maximum block size B_i in phase i is less than*

$$\widetilde{B}_i = \frac{7}{3} \cdot \frac{n}{2^{2^{i-1}+i-2} p}.$$

Proof. (by induction on i)

$i = 1$: In phase 1 we can choose a block size of n/p, which is less than $\widetilde{B}_1$.

$i \to i + 1$: Since this case is relevant only for $i < T$, we can assume that $\Phi_i > 3p$. Moreover, by the induction hypothesis, $\Phi_{i-1} \leq 2B_i p + p \leq (7/3) n/2^{2^{i-1}+i-3} + p$. Denote by p_i the number of processors active at the end of phase i.

Since $\Phi_{i-1} - \Phi_i \geq p_i \cdot q_i$ (recall that $q_i = \lceil (49/9)2^{4-i}n/p \rceil$ is the number of iterations executed in phase i), we obtain $p_i \leq (\Phi_{i-1} - \Phi_i)/q_i \leq ((7/3)n/2^{2^{i-1}+i-3})/((49/9)2^{4-i}n/p) = (3/7)p/2^{2^{i-1}+1}$. An argument used above shows that $\Phi_i \leq 2p_iB_i + p_i$, i.e., $p_iB_i > p$. Since also $p_i < p/4$, we can apply Lemma 1, which shows that the maximum block size B_{i+1} in phase $i+1$ is less than $(7/3)p_iB_i/p$. By the induction hypothesis and the upper bound on p_i established above, $(7/3)p_iB_i/p < (7/3)n/(2^{2^i+i-1}p) = \widetilde{B}_{i+1}$.

□

Proof of Theorem 2. By Lemma 3, $\Phi_{i-1} \leq 2pB_i + p \leq (7/3)n/2^{2^{i-1}+i-3} + p$, for $i = 1, \ldots, T$. It follows that $O(\log\log(n/p))$ phases suffice to reduce Φ and hence the number of unvisited positions below $3p$. The remaining items can be moved in time $O(n/p+\log n)$. The phases consist of $\sum_{1\leq i\leq T} \lceil (49/9)2^{4-i}n/p \rceil = O(n/p)$ iterations, each of which takes constant time. The prefix summation takes time $O(\log p) = O(\log n)$ per phase. Hence, the total run time is $O(n/p + \log n \log\log(n/p))$.

The n global bits are required by the original algorithm. Local cells are needed to back up one item during the permutation and to back up a constant number of global memory cells during the parallel prefix summation. Hence $O(1)$ local cells are sufficient. □

5. Discussion

Some further improvements are possible. First, from the proof of Theorem 2, we can immediately derive the following Corollary 2.

Corollary 2 *If prefix summation can be realized in constant time, then the improved algorithm runs in time $O(n/p)$ and hence is optimal for $p \leq n$.*

This is important for architectures that emulate the PRAM model and support prefix computation at the instruction level. Examples are the Fluent Machine [8], the NYU Ultracomputer [9] and the SB-PRAM [10].

Improvements are also possible if a CRCW PRAM is used and the prefix summation is replaced by faster load balancing subroutines. Using the techniques from [11] for a randomized PRAM and from [12] for a deterministic PRAM, the run time of the algorithm can be reduced to $O(n/p + \log^* n \log\log(n/p))$ and $O(n/p + (\log\log n)^3 \log\log(n/p))$, respectively. However, these improvements seem to be less practical because of larger constant factors in the advanced load balancing algorithms.

In our analysis, we have distinguished between bits and memory cells. Bits are considered different, because implementing them often will not increase the storage used. In the representation of the items, there will often be an unused bit that can be used to encode the "visited" flags. Also, many memory subsystems today provide each cell with additional bits that are used for parity, access control, etc. One of these probably could be used for implementing the flags.

The behaviour of our simple algorithm depends on the permutation π. For many permutations the behaviour should be much better than indicated by our (worst-case) bound of $O((n/p)\log n)$. We support this belief by simulation results. For

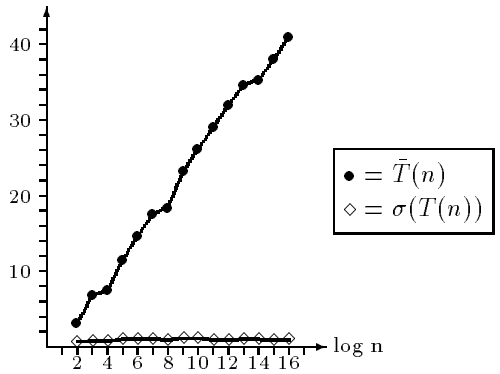

Fig. 2. Number of iterations (average and standard deviation) of the simple algorithm

$n = 2^i$, where $2 \leq i \leq 16$, and $p = \lfloor n/\log n \rfloor$, we simulated the algorithm on 100 randomly chosen permutations. The average and standard deviation of the number of iterations needed are shown in figure 2. The standard deviation is small, and the number of iterations is always smaller than $3 \log n$. This hints at the average behaviour of the simple algorithm being much better than its worst behaviour. However, the average run time still needs to be analyzed.

Acknowledgements

We want to thank Dany Breslauer and John Tromp for helpful suggestions.

References

1. J. Keller, Fast rehashing in PRAM emulations, in *Proc. 5th Symp. on Parallel and Distributed Processing*, Dallas, TX, Dec. 1993, 626–632.
2. D. E. Knuth, Mathematical analysis of algorithms, in *Proc. of IFIP Congress 1971*, Information Processing 71, (North-Holland, Amsterdam, 1972) 19-27.
3. R. Melville, A time/space tradeoff for in-place array permutation, *J. Algorithms* **2** (1981) 139–143.
4. F. E. Fich, J. I. Munro and P. V. Poblete, Permuting, in *Proc. 31st Symp. on Foundations of Computer Science*, St. Louis, Miss., Oct. 1990, 372–379.
5. A. Aggarwal, A. K. Chandra and M. Snir, On communication latency in PRAM computations, in *Proc. 1st Symp. on Parallel Algorithms and Architectures*, Santa Fe, NM, June 1989, 11–21.
6. M. Snir, Personal communication, 1992.
7. A. Chin, Permutations on the block PRAM, *Inform. Process. Lett.* **45** (1993) 69–73.
8. A. G. Ranade, S. N. Bhatt and S. L. Johnson, The Fluent Abstract Machine, in *Proc. 5th MIT Conference on Advanced Research in VLSI*, Boston, Mass., Mar. 1988, 71–93.

9. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU ultracomputer — designing an MIMD shared memory parallel computer, *IEEE Trans. Comput.* **32** (1983) 175–189.

10. F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul and D. Scheerer, On the physical design of PRAMs, *Comput. J.* **36**(8) (1993) 756–762.

11. H. Bast and T. Hagerup, Fast parallel space allocation, estimation and integer sorting (revised), Technical Report MPI-I-93-123, Max-Planck-Institut für Informatik, Saarbrücken, June 1993.

12. T. Hagerup, Fast deterministic processor allocation, in *Proc. 4th Symp. on Discrete Algorithms*, Austin, TX, Jan. 1993, 1–10.