Doubly logarithmic time parallel sorting

R.Sarnath
Dept. of Computer Science
226, Bell Hall
SUNY at Buffalo
Amherst NY 14260
email: sarnath@cs.buffalo.edu
Phone (716) 645-3180
Keywords: Parallel comparison sorting, Doubly logarithmic time, Constant time maximum

Abstract

Recently, attempts have been made to separate the problem of parallel sorting from that of list ranking, in order to get around the well known $\Omega(\log n/\log\log n)$ lower bound. These approaches have been of two kinds - *chain sorting* and *padded sorting*. Here we present nearly optimal, comparison based *padded* sorting algorithms that run in average case time $O(\frac{1}{\epsilon^2} + \frac{1}{\epsilon}\log\log n)$ using $n^{1+\epsilon}$ processors, and $O(n^{1+\epsilon})$ space, on an Common CRCW PRAM.From these results, algorithms for *chain sorting* within the same time and processor bounds can be easily obtained. Using a similar approach, we also give an $O(1)$ average case time, comparison based algorithm for finding the largest of n items using a linear number of processors. The algorithm for finding the maximum, runs on a Common CRCW PRAM using only $n^{3/4}$ cells of shared memory. Finally, we obtain randomised algorithms for these problems that run on Common/Tolerant CRCW PRAMs, and also satisfy the above time and processor bounds. As a consequence of our results on sorting, we can also show that the problem of sorting arbitrary input sequences can be reduced to that of sorting integer inputs, within the above mentioned time and processor bounds.

1 Introduction

The problem of sorting a sequence of numbers is well known. Considerable attention has been paid to the problem of parallel sorting on shared memory machines. On the EREW PRAM, Cole's parallel mergesort [8] acheived $\Theta(\log n)$ worst case time using n processors. Leighton's modification [11] to the AKS sorting network [1], acheived the same time and processor bounds. Rajashekaran and Reif [13] give a randomized $\Theta(\log n / \log\log n)$ time, $n \log^{\epsilon} n$ processor algorithm for general sorting. They also have a randomized algorithm for integer sorting which runs in expected time $\Theta(\log n)$ and uses $n/\log n$ processors.

Very fast parallel algorithms have been developed for several problems. We characterise very fast algorithms as those that run in time polynomial in $\log\log n$ using a polynomial number of processors. Some examples of these can be found in [5,6,12,15]. It has been shown by Beame and Hastad [4] that sorting n items using polynomially many processors requires $\Omega(\log n/\log\log n)$ time. Thus, sorting in time polynomial in $\log\log n$ using polynomially many processors is impossible. Recently, attempts have been made to separate the list ranking problem from sorting, thus accomplishing a "chain-sort" [10]. Among other results, Hagerup [10] has succeeded in sorting n integers in the range $[1, n]$ into a linked list in $O(1)$ expected time using $O(n \log n/\log\log n)$ processors and in $O(\log\log n \log^{\star} n/\log\log\log n)$ expected time using an optimal number of processors. Another attempt to sidestep this lowerbound is by doing a *padded sort* - sorting into a larger array [12]. We define the padded sorting problem as follows:

Problem : *Given an array A of length $|A| = n$, holding items $a_1, a_2, ...a_n$, construct an array $A^{\star}$ such that,*

- $|A^{\star}| \geq |A|$.
- *For every a_i there exists an $a_j^{\star} = a_i$ and vice-versa.*
- $a_j^{\star} \leq a_{j+1}^{\star}$, $\forall j$, $1 \leq j < |A^{\star}|$.

Since many problems rely simply on the relative ordering information and not the actual rank, a padded sort may be as useful as a conventional sort. For instance, padded sorting is useful in a situation where the ordering is primarily needed for fast searching. The results in [12] are for uniformly distributed inputs, i.e, any item in the specified range is equally likely to appear in the input array. Their ideas for padded sorting have also found applications in solving some problems in computational geometry. They have acheived an expected time bound of $O(\log\log n)$ using $n/\log\log n$ processors. These algorithms however, are not comparison based. Reischuk[14] has some results on Parallel Decision Trees for selection. He gives an $O(1)$ expected depth parallel decision tree for selection using the idea of multi-point sampling, but does not discuss any PRAM algorithms. An algorithm that finds the maximum in $O(1)$ expected time on a Priority CRCW PRAM has been given by Alon and Megiddo in [3]. This has been done by formulating the problem of finding the maximum as a linear programming problem in one variable.

In this paper we give a deterministic comparison based algorithm for the *padded sorting* problem which runs in average case time $O(\frac{1}{\epsilon^2} + \frac{1}{\epsilon}\log\log n)$ using $n^{1+\epsilon}$ processors, and $O(n^{1+\epsilon})$ space, for some $\epsilon \leq 1$, on an Common CRCW PRAM. Using the same approach, we also give a simple algorithm for finding the largest of n elements, that runs in $O(1)$ average case time, using a linear number of processors on a Common CRCW PRAM. The algorithm uses $n^{3/4}$ cells of shared memory, whereas it is well known that any algorithm, for computing the maximum of n items, which has a worst case running time of T, requires at least n/T cells of shared memory [16]. While carrying out the average case analysis, we assume that *all permutations of the input elements are equally likely.*

Finally, we show how we can use randomisation to convert these average case time bounds into expected time bounds, i.e., for any input, irrespective of distribution, we can acheive the same time and processor bounds with *High Probability* (HP). By HP, we mean that there exists a constant c, $c > 1$, such that the probability of success of our algorithm is at least $(1 - n^{-c})$ for an input of size n. The randomised algorithms run on Tolerant/Common CRCW PRAMs. We shall adopt the definitions of these models, as given by Chlebus et al. [7]

Throughout the rest of the paper, we shall assume that all the elements are distinct. This does not result in any loss of generality in the analysis or implementation of our algorithms. We shall use superscripted or subscripted upper-case letters to denote arrays and the corresponding lower-case letters to denote elements of these arrays. We use log to denote logarithms to the base 2 unless stated otherwise.

The next section presents the algorithm for finding the maximum. Section 3 gives a padded sorting algorithm, that uses n^2 processors. In section 4, we generalise this to obtain an $n^{1+\epsilon}$ processor, $O(\frac{1}{\epsilon}\log\log n)$ time padded-sorting algorithm. In section 5, we discuss the randomised algorithms. Section 6 concludes the paper.

2 Finding the maximum in $O(1)$ time

Algorithm MAX

(* An array of n items is given. We divide the array into $\sqrt{n}$ $(= m)$ contiguous blocks of equal size (m each); initially, all items are candidates for maximum *)

While (max is not found) *do*

From each block, pick the leftmost candidate.

We have at most m such items. (lets call these items *pivots*)

Compute p_0, the maximum of the pivots.

Mark all those elements that are $\leq p_0$ as non-candidates.

end While;

end MAX.

Lemma 1: *Every execution of the while loop of the algorithm can be carried out in $O(1)$ time on a Common CRCW PRAM using $n^{3/4}$ cells of shared memory.*

Proof: Finding the maximum of $\sqrt{n}$ elememts can be done in $O(1)$ time using n processors as described in [15]. To pick the pivots, we do the following: *Divide each block into $n^{1/4}$ equal sized sub-blocks. Using a single concurrent write operation, we determine which of these sub-blocks contains a candidate. We can then find the leftmost candidate of the leftmost such sub-block in 2 applications of the algorithm in [15].* Note that this is possible since we have $\sqrt{n}$ processors, and only $n^{1/4}$ sub-blocks with $n^{1/4}$ items per sub-block. $\square$

We prove next, that in atmost 6 executions of the while loop, we would have found the maximum with high probability (HP). Let us define $P(n,r)$, to be the number of distinct permutations of n objects taken r at a time.

Lemma 2: *Let A be an array of n cells, containing k, $k \leq n$, candidates. If we execute one iteration of the while loop of MAX on A, then, atmost $\alpha = max(k \log n/m, 2\log^2 n)$ candidates remain, with HP.*

Proof: Consider any contiguous set of $2(m/k) * \log n$ blocks. This set will have p, $p = 2(n/k) * \log n$, cells in it. If we do not place any candidate in these p cells, the number of ways of placing k candidates in n cells is $P(n-p,k)$. If all permutations are equally likely, the probability of such an ordering is $\leq e^{-2\log n}$. From this, we can see that, with HP, every contiguous set of p array cells will have at least one candidate in it. We therefore have at least $\beta = min(k/2\log n, \sqrt{n})$ pivots with HP.

The k candidates thus occupy atleast β blocks, with HP. Now, consider the largest α elements from the k candidates. If none of these were to be chosen as a pivot, then the probability of such a permutation would be atmost $P(k-\alpha,\beta)*(k-\beta)!/k!$. (This count arises because the first position in each of the β blocks can be filled in only by one of the $k-\alpha$ candidates.) From this, we have that at least one of the largest α candidates is chosen as a pivot with probability $\geq (1 - n^{-\log e})$. $\square$

Lemma 3: *After 2 executions of the loop, there are no more than $2\log^2 n$ candidates with HP.*

Proof : From lemma 2, we have that after 1 iteration of the while loop, no more than $m * \log n$ candidates remain, with HP. Applying lemma 2 to this fact, gives us the required result.$\square$

Lemma 4: *After 2 executions of the while loop, no block contains more than 4 elements with HP.*

Proof: Look at the $2log^2n(=q)$ items that are still candidates. Each of these is equally likely to be in any block. Consider some particular block. The probability that we have at least l candidates in the block is $\leq [(q/m)^l]$ (From a straightforward counting argument.) From this we can see that the probability that any block has more than 4 elements is small ($< n^{-c}$). $\square$

We thus have the following result:

Theorem 1: *The algorithm MAX finds the maximum of n items in $O(1)$ average case time, using n processors and $n^{3/4}$ cells of shared memory, on a Common CRCW PRAM.*

3 A padded sorting algorithm

The idea behind the sorting algorithm is similar to that of a "multi-way" quicksort. We choose a large number of pivots at each step, so that the input can be divided into partitions of small size ($O(\log^c n)$). If we sort these pivots and the partitions separately, we can then put them together to get a sorted array. The sorting of pivots is done by recursive calls. We can sort the small partitions, using the algorithm in [8], in $O(\log \log n)$ time. However, we must also ensure that the size of the input to the recursive calls is no more than $O(\sqrt{n})$, so that the recursion depth is no more than $O(\log \log n)$. We do this in two stages - each stage involving a $\sqrt{n}$-way partitioning. The details of this are given in the algorithm *padded sort.*

After the recursive calls have been completed, we have to take care of the merging. If such a merging were to be done without increasing the size of the final array, it would entail adding $O(\sqrt{n})$ items, which cannot be done very quickly. We therefore do a "padded merge". The crucial observation being that *if all the partitions were of equal size, we could get away with a single multiplication instead of computing a prefix sum.* We therefore *pad* all the partitions, and make each of them as large as the largest one. (By *padding* we mean putting all the elements into a larger array, and appropriately filling the gaps). These details have been outlined in the algorithm $MERGE$.

We shall now give an algorithm that does a padded sort of n items in $O(\log \log n)$ time using n^2 processors. Using this as a subroutine, we shall obtain an algorithm that runs within the time and processor bounds claimed earlier.

Algorithm Padded Sort (A, n, δ, A^*)

(* A is an input array of size n; A^* is the output produced; we divide A into $\sqrt{n}$ contiguous, equal sized blocks; every cell of A need not contain an element, i.e, A has n cells, and atmost n elements *)

IF $n < \delta$, THEN sort A using the algorithm in [8]

ELSE,

- Step 1: Pick the leftmost element of each block to be a pivot; partition the elements of A using these (atmost) $\sqrt{n}$ pivots. Let p_j denote the pivot picked from the j-th block. Each element a_i chooses j such that $|a_i - p_j|$ is minimized. We then mark a_i as belonging to partition j. Place these pivots in an array of size $\sqrt{n}$ and sort them recursively.

- Step 2: For each partition j, $1 \leq j \leq \sqrt{n}$, do the following:

  From each block, pick the leftmost element belonging to partition j. Let p_{jl} denote the leftmost element of the l-th block. Every a_i in partition j then chooses an l such that $|a_i - p_{jl}|$ is minimized. We mark a_i as belonging to partition $((j-1) * \sqrt{n} + l)$. All these pivots (atmost $\sqrt{n}$) are placed in an array of size $\sqrt{n}$ and sorted recursively.

- Step 3: Chain the elements of each partition created in step 2, into a linked list.(see section 3.1) We then compress this list into an array and sort the elements using the algorithms in [8,9].

- Step 4: Using the sorted arrays of pivots obtained from the recursive calls in step 2, merge the partitions created in step 2 into a single padded sorted array with $\sqrt{n}$ parallel calls to the algorithm $MERGE$

- Step 5: Using the sorted arrays returned from the recursive call in step 1, merge all the partitions into the array A^* by making a single call to $MERGE$.

END Padded Sort

The structure of the first two steps, is very similar to the first two iterations of the while loop in MAX. The only difference is that, in this case, we have to prove bounds on the sizes of all the partitions, whereas in MAX, we concerned ourselves only with the partition that contained the largest element . Using an argument similar to lemmas 2 and 3, we have,

Lemma 5: *The size of the partitions created in step 2 of the algorithm Padded Sort is* $\leq c\log^2 n$ *with HP, for a suitable choice of the constant c. More specifically, the probability that any partition is of size* $> c\log^2 n$ *is* $\leq e^{-c\log n/2}$.

We allow a larger constant factor here, since we need a high confidence bound for all the partitions. Note that there is a chance of failure here, i.e, a partition may be of size $> c\log^2 n$. However, this is detected within our claimed resource bounds, as explained in section 3.1. If this occurs, we abort the exercise, and sort the input using the algorithm in [8]. Since the probability of this event is very small, our average case claim holds.

We shall now show how all these steps can be carried out in $O(1)$ time using n^2 processors. To begin with, we make the following observation about the recursive steps in our algorithm.

Observation : *Given an input array A, of size k, the algorithm generates* $\sqrt{k}+1$ *recursive calls, each of which solves a problem of size* $\sqrt{k}$. *(1 recursive call in step 1, and* $\sqrt{k}$ *recursive calls in step 2).*

From this observation we can deduce the following lemma.

Lemma 6 : *Let the initial input to the padded sorting algorithm be an array of size n. At the i-th level of the recursion, we shall have* $O(n^{1-1/2^i})$ *sub-problems, each of size* $n^{1/2^i}$. *(Here, the first execution of the algorithm is level 0 of the recursion.)*

We can now establish the processor complexity of steps 1 and 2.

Lemma 7 : *Steps 1 and 2 can can be carried out in* $O(1)$ *time using* n^2 *processors.*

Proof : Consider the i-th level of the recursion, where each subproblem is of size $n^{1/2^i}$. We choose $n^{1/2^{i+1}}$ pivots to partition each subproblem; this implies that we have to compute the minimum of $n^{1/2^{i+1}}$ differences for each element. This minimum can be computed in $O(1)$ time using $n^{1/2^i}$ processors, using the algorithm in [15]. This implies that we can carry out steps 1 and

2 for each subproblem in $O(1)$ time using $n^{1/2^{i-1}}$ processors. From this, we deduce that steps 1 and 2 for all the subproblems in the i-th level of the recursion, can be solved in $O(1)$ time using $n^{1+1/2^i}$ processors. At $i = 0$, we get the result of the lemma. □

In the next subsection, we shall discuss the problem of chaining the elements, i.e, step 3 of the algorithm padded-sort.

3.1 Chaining the elements

We now deal with the problem of chaining together the elements of the partitions created in step 2 of each call to *padded sort*. In its most general form, we can state the problem as follows:

Problem: *Given an array of n elements, each of which belongs to one of m possible sets, we wish to chain the elements of each set into a linked list. More formally, if a_1, a_2,..., a_n were the n elements, then, for each set S, and for each $a_i \in S$, find j such that $a_j \in S$ and $\forall k$, $i < k < j$, a_k is not in S.*

However, we do not have to concern ourselves with the general problem, and examine instead, only those situations, where the size of each set is no more than $\alpha = c \log^2 n$. At the start of the separation process, we do not know if this is true of every set, but we can check for this condition as we go along, aborting if we find that the condition does not hold. This check does not add to the asymptotic time complexity of our algorithm. The algorithm is described below.

Algorithm Chain:

Consider the i-th stage of the recursion, where we have $O(n^{1-1/2^i})$ subproblems, each of size $n^{1/2^i}$. With n^2 processors, we can assign $n^{1/2^{i-1}}$ processors to each subproblem, i.e., we can assign $n^{1/2^i}$ processors to each item. The algorithm proceeds as follows:

- Divide the subproblem into $n^{1/2^{i+1}}$ *parts*, each of size $n^{1/2^{i+1}}$. Each item can then find its nearest right neighbor that lies within its part in $O(1)$ time.
- Within each part, the last item of a chain can then find the nearest part to its right that contains an item from its own set. This can also be done in $O(1)$ time.

end Chain

We therefore have:

Lemma 8: *At each stage of the recursion, the chaining in step 3 can be can be carried out in $O(1)$ time.*

Also, since the size of each chain is no longer than α, we have:

Lemma 9: *The process of compressing each partition and then sorting it can be carried out in time $O(\log \log n)$ for all partitions, in parallel, using no more than n^2 processors.* [8,9].

In the next subsection, we deal with the complexity of steps 4 and 5 of our algorithm.

3.2 Merging the partitions

Algorithm $MERGE$ (A_0, A_1, ... , A_k, A);

(* A_0 is a pivot array; A_1 to A_k are the arrays corresponding to the partitions created by pivots of A_0; all the A_i's are padded sorted arrays; A is the final merged array *)

Let $max\{|A_i| \ : \ 1 \leq i \leq k\} = M$. All the arrays A_1 through A_k pad themselves to be of size M.

Let x_i be the pivot corresponding to A_i. x_i occupies cells numbered y_i through z_i in A_0. The merging operation is carried out as follows:

Let a^i_j be the element in the j-th cell of A_i. a^i_j makes $z_i - y_i + 1$ copies of itself, and stores these in the A, in the cells numbered

$[M \times (y_i - 1) + (z_i - y_i + 1) \times (j - 1) + 1]$ to $[M \times (y_i - 1) + (z_i - y_i + 1) \times (j - 1) + (z_i - y_i + 1)]$.

end $MERGE$.

Clearly, all the steps in algorithm $MERGE$ can be carried out in $O(1)$ time. The following lemma needs no proof.

Lemma 10: *The array A formed by the merging is of size $M \times A_0$. Furthermore, A is itself a padded sorted array.*

The size of the padded sorted array produced, is given by the recurrences

$$S_n = (S_{\sqrt{n}})^2 c \log^2 n \tag{1}$$

$$S_\delta = \delta \tag{2}$$

where S_n denotes the size of the padded sorted array when the input is of size n. The idea here is that, at the end of step 2, we are left with partitions of size $c \log^2 n$, which are then merged together in two stages. In each stage, the sorted pivot arrays are of size $S_{\sqrt{n}}$. Applying the result of lemma 10 to these facts, we get the above recurrences. Choosing $\delta = \log^d n$, for constant d, we have

$$S_n = n^{1+\frac{2}{d}} c^{\frac{\log n}{d \log \log n}} \tag{3}$$

Theorem 2: *Algorithm padded sort runs in average case time $O(\log \log n)$ using n^2 processors on a Common CRCW PRAM.*

Proof : We have shown that each step in every recursive call can be carried out in $O(1)$ time. Since the depth of recursion is no more than $\log \log n$, we have the required time bound.

There is a probability of failure associated with each recursive call, i.e., a partition created in step 2 may be of size $> c \log^2 n$. The probability of such an event is $\leq n^{-c \log e/2}$. Since there are no more than $2n^2$ partitions being created in the entire course of the algorithm, the combined failure probability is $\leq O(n^{-c \log e/2+2})$. In case of failure, we can resort to the algorithm in [8], to sort in $O(\log n)$ time, giving us an average case time of $O(\log \log n)$. □

4 An $n^{1+\epsilon}$ processor algorithm

Algorithm Efficient-sort

Divide the array into $n^{\epsilon/2}$ blocks, each of size $n^{1-\epsilon/2}$. Initially, all n items belong to the same partition.

1. For $i := 1$ to $2/\epsilon$ do

   - For each partition created in step $i-1$, pick $n^{\epsilon/2}$ pivots. (The first item in each block) Use these pivots to subdivide the partition into $n^{\epsilon/2}$ parts. Place these pivots in an array of size $n^{\epsilon/2}$, and sort this array using algorithm padded sort.

2. At the end of step 1, we are left with a maximum of n partitions. Chain the elements of each partition into a linked list, compress and sort them using the algorithms in [8,9].

3. For $i := 2/\epsilon$ to 1 do

   - Merge all the partitions created in the i-th iteration of step 1, using the algorithm MERGE.

end Padded-sort2

We have the following lemma about the partitions created in step 1. This is proved using counting arguments similar to the ones used for lemmas 2 and 3.

Lemma 11: *Size of any partition created in the final iteration of step 1 is no more than $6^{\frac{2}{\epsilon}} \log^{\frac{4}{\epsilon}} n$ with HP.*

From this we can easily see that the size of the final array is no more than

$$6^{2/\epsilon}(\log^{4/\epsilon} n)(S_{n^{\epsilon/2}})^{2/\epsilon} \tag{4}$$

which is $O(n^{1+\epsilon})$ for $d = \frac{3}{\epsilon}$.

Lemma 12: *The process of choosing pivots, and partitioning the arrays, can be done in time $O(1/\epsilon)$ on a Common CRCW PRAM.*

Proof: We can state the problem of choosing pivots as follows:

- Given an array of size n, divided into n^r, $r < 1$, equal-sized blocks, and a set of k elements, $k \leq n$, scattered among the n array cells, find the leftmost element in each block.

Observe that we have atleast n^{2r} processsors available, per array cell. Also each block is of size n^{1-r}. We divide each block into n^r pieces, and determine the leftmost piece that contains one of the k elements. This can be done in $O(1)$ time using n^{2r} processors. We repeat this operation for $(1/r - 1)$ steps, after which, we are left with pieces of size n^r. The leftmost in each piece can then be computed in $O(1)$ time. Since r is atleast $\epsilon/2$, this operation takes no more than $2/\epsilon$ steps.

In order to partition the array, note that we can find the minimum of n^r quantities using n^{2r} processors on a Common CRCW PRAM in $O(1)$ time. [15] $\square$

Lemma 13 *Step 2 of algorithm efficient sort can be carried out using $n^{1+\epsilon}$ processors in $O(\frac{1}{\epsilon}\log\log n)$ time.*

Proof: For each partition S, do the following:

Divide the array into $n^{\epsilon/2}$ blocks. Chain together the blocks that contain an element of S, and count them. The chaining can be done in time $O(\log\log n)$ using $n^{\epsilon/2}$ processors [6]. We can now assign $n^{\epsilon/2}$ processors to each such block and continue the process. This process terminates when the block size drops to $n^{\epsilon/2}$ i.e., in no more than $2/\epsilon$ steps, each step requiring time $O(\log\log n)$. Each element is now assigned $n^{\epsilon/2}$ processors to find its nearest right neighbor within its block. This can be accomplished in $O(\log\log n)$ time. (Note that since all the blocks are chained together, we have succeeded in chaining together all the elements of S.) The total time required is no more than $O(\frac{2}{\epsilon}\log\log n)$. These operations can clearly be done as stated on a Common model CRCW PRAM. $\square$

Theorem 3: *The algorithm efficient sort runs in average case time $O(\frac{1}{\epsilon^2}+\frac{1}{\epsilon}\log\log n)$ using $n^{1+\epsilon}$ processors, with HP.*

Proof : At each iteration of step 1, each element has to compute the minimum of $n^{\frac{\epsilon}{2}}$ values, which can be computed in $O(1)$ time, since we have n^{ϵ} processors per element. There are atmost $\frac{2}{\epsilon}$ steps, so this takes $O(\frac{2}{\epsilon})$ time. The process of picking the pivots in each step takes $O(\frac{1}{\epsilon})$ time, so step 1 takes $O(\frac{1}{\epsilon^2})$ time.

Each of the calls made to padded sort require n^{ϵ} processors, and we have $O(n)$ of these. These can therefore be carried out in $O(\log\log n)$ time using $n^{1+\epsilon}$ processors.

The chaining takes time $O(\frac{1}{\epsilon}\log\log n)$, and so do the list ranking and sorting [8,9]. We have to do one such process in step 2.

To carry out the average case analysis, we once again look at the total number of partitions created by efficient sort. We have $O(n)$ partitions created in step 1 of efficient sort. Each call to padded sort results in $O(n^{\epsilon})$ partitions. We can add the failure probabilities of each of the "partitioning events" and get the required result, using an argument similar to the one used for theorem 2. $\square$

5 Randomised Algorithms

In order to get an algorithm that performs well on any input with HP, (independent of distribution) we first perform the following randomisation. Using $n/4\log n$ processors, we sample from A, uniformly at random. The sampling can be clearly carried out in $O(1)$ time on a Common CRCW PRAM. We shall place these $n/4\log n$ items in the array A'. The following facts follow from a direct application of the Chernoff bounds:

1. No element of A is sampled more than once, with HP.

2. Given any set of $16 \log^2 n$ elements of A, at least one element of the set has been sampled with HP.

In order to find the maximum, we apply the algorith MAX to the array A'. Let this maximum be e. *Mark* all those elements in A that are larger than e. Now, consider the set of the largest $16 \log^2 n$ elements of A. At least one of these was sampled with HP. Therefore, atmost $16 \log^2 n$ elements of A are *marked*, with HP. We then generate, corresponding to each marked element i, a random number r_i, $1 \leq r_i \leq \sqrt{n}$. The element i is then written into $A'[r_i]$. It can be easily seen, that with HP, no write conflicts will occur. We can then compute the maximum of these $\sqrt{n}$ items in $O(1)$ time. All the above steps can be clearly carried out as stated on a Tolerant CRCW PRAM.

In order to do a padded sort, we shall first do a padded sort of the array A'. We now use the items in A' as pivots for partitioning A. It is immediately obvious from the facts stated above, that no partition thus created, can have size greater than $32 \log^2 n$. Also, since we have $n^{1+\epsilon}$ processors, each element can find its pivot in $O(1/\epsilon)$ steps, using a "multi-way" binary search. Each of these partitions can then be chained, compressed and sorted in time $O(\log \log n)$. We can then merge all these partitions, with a single call to $MERGE$, in $O(1)$ time.

6 Remarks and Conclusions

We now show how this algorithm can be used to transform any instance of the sorting problem into one of sorting integer inputs. For each element a_i in the input array A, we find the smallest index x_i, such that $A^\star[x_i] = a_i$. Now create a new array A' such that a'_i holds the value x_i. Clearly, A' contains only integers. Also, sorting A' would amount to sorting A. To sort the elements into a linked list, all we have to do is chain the elements of $A^\star$ such that x_i is linked to x_{i+1}. This can be done in $O(1)$ time using n processors on a Common CRCW PRAM in the obvious way.

There is a well known result due to Alon and Azar [2] on sorting using the Parallel Comparison Tree (PCT) model. Their result states that every PCT with p processors for sorting n elements requires average case time $\Omega(f(n,p))$, where $f(n,p)$ is given by

$$f(n,p) = \log n/(\log(1+p/n)), \quad for \quad p < \binom{n}{2}, \quad and \quad f(n,p) = 1, \quad otherwise. \tag{5}$$

This would imply that sorting in time $O(\frac{1}{\epsilon} \log \log n)$ would require at least $n^{1+\frac{\epsilon}{\log \log n}}$ processors on a PCT model, and therefore on a CRCW PRAM as well. Although these results have been derived for conventional sorting, they apply to padded and chain sorting as well. The results in this paper are therefore within a factor of $\log \log n$ of the optimal.

It is clear that a padded sort provides more information than a chain-sort, but not as much as a conventional sort. It would therefore be interesting to see if we can derive tight bounds on both the time as well as space complexity of padded sorting. The only obvious bound we can place on the size of the final array, is that it cannot be less than $n + polylog(n)$, since we can then compress and eliminate the duplicates to get a conventional sort, thus beating the lower bound in [4]. It would also be an interesting open question to find a deterministic padded sorting algorithm, that has worst case running time better than $\Omega(\log n / \log\log n)$.

References

[1] Ajtai M., Komlos J. and Szemeredi E., An $O(n \log n)$ sorting network. Combinatorica, 3:1-19, 1983.

[2] Alon N. and Azar Y., The average case complexity of deterministic and randomized parallel comparison sorting algorithms. SIAM J Comput., 17, 1988, 1178-1192.

[3] Alon N. and Megiddo N., Parallel linear programming in fixed dimension almost surely in constant time. Proc. 31st annual IEEE FOCS, pp 574-582, 1990.

[4] Beame P. and Hastad J., Optimal bounds for decision problems on the CRCW PRAM. J of the ACM, 36(3):643-670, July 1989.

[5] Berkman O., Breslauer D., Galil Z., Schieber B., Vishkin U., Highly parallelizable problems (extended abstract). Proc. 21st annual ACM STOC, pp 309, 1989.

[6] Borodin A. and Hopcroft J.E., Routing merging and sorting on parallel models of computation, J of Comput. Syst. Sci., 30, 1985, pp 130-145.

[7] Chlebus B.S., Diks K., Hagerup T., and Radzik T., New Simulations between CRCW PRAMs, in Proc 7th Int'l Conf. on Fundamentals of Computation Theory, Springer lecture notes in computer science, vol 380, pp 95-104.

[8] Cole R., Parallel merge sort. Proc. 27th annual IEEE FOCS, pp 511-516, 1986.

[9] Cole R. and Vishkin U., Approximate and exact parallel scheduling, with applications to list, tree and graph problems. Proc. 27th annual IEEE FOCS, pp 478-491, 1986.

[10] Hagerup T., Fast parallel space allocation, estimation and integer sorting, tech rep MPI-I-91-106, Max Planck Institut fur Informatik, June 1991.

[11] Leighton T., Tight bounds on the complexity of parallel sorting. IEEE trans. on computers, c-34:344-354, April 1985.

[12] Mackenzie P. and Stout Q., Ultra-fast expected time parallel algorithms. Proc. of symposium on discrete algorithms, 414-424, 1991.

[13] Rajasekaran S. and Reif J.H., Optimal and sublogarithmic time randomized parallel sorting algorithms. SIAM J of computing, 18(3):594-607, June 1989.

[14] Reischuk R., Probabilistic parallel algorithms for sorting and selection, SIAM J of comput, vol 14, no. 2, May 1985.

[15] Shiloach Y. and Vishkin U., Finding the maximum, merging, and sorting in a parallel computation model, J. Algorithms, vol. 2, 1981, pp 88.

[16] Vishkin U. and Wigderson A., Trade-offs between depth and width in parallel computation, SIAM J comput, vol 14, no.2, May 1985, 303-314.