Parallel Merge Sort on Concurrent-Read Owner-Write PRAM

David C. Lin, Patrick W. Dymond, Xiaotie Deng

Department of Computer Science, York University, North York, Ontario, Canada
{lin,dymond,deng}@cs.yorku.ca

Abstract. This paper discusses a variant of the CREW PRAM model introduced by Dymond and Ruzzo called *CROW* (Concurrent Read Owner Write) PRAM. in which each global memory location may only be written by its assigned *owner* processor. We show that CROW PRAMs can sort in $O(\log n)$ parallel time using $O(n \log n)$ processors.

1 Introduction

The PRAM (Parallel Random Access Machine) model for parallel computation was introduced as "a collection of synchronous processors executing in parallel and communicating via a global shared memory" by Fortune and Wyllie [FW78]. Their model satisfies the *CREW* (Concurrent-Read, Exclusive Write) convention for accesses to global memory. Dymond and Ruzzo proposed a natural and frequently occurring restriction of the CREW PRAM, called *CROW* (Concurrent Read Owner-Write) PRAM machines, for which each global memory location is assigned a unique *owner* processor and the *owner* is the only processor allowed to write into that cell throughout the course of a computation [DR86],[DR97]. By restricting the PRAM so that only its owner may write to each memory location, it is possible that algorithms become easier to implement, particularly on real machines in which global memory is simulated by dividing it among local processors e.g. in a network of workstations.

In this paper we describe a basic $O(\log n)$ sorting algorithm for the CROW PRAM. Cole's CREW PRAM merge sort algorithm [C88] does not appear to be easily modifiable to work on the CROW PRAM. In Cole's algorithm, sorting of n numbers occurs on a complete binary tree. (We assume $n = 2^m$ for some integer $m \geq 0$.) Initially these n numbers are individually distributed one per leaf to the n leaves of the tree. On each node of the tree, the computation is to merge two sorted subarrays on its children nodes into one sorted array, by computing the ranks (in the array of the node) of these items in the subarrays of the child nodes in $O(1)$ time. The item of rank i is written into the i–th position for that array in the global memory. Since the rank is not known ahead of time, it is not possible to know which processor is going to write in which memory location. In general, it may take $O(\log n)$ time to resolve this. Thus, the direct approach to implementing Cole's sorting in the CROW PRAM results in an $O(\log^2 n)$ time algorithm. This paper modifies Cole's parallel merge sort algorithm with a new method of merging, *divided-merging*, and a new method of sampling to accomplish the merging in constant time.

2 CROW PRAM sorting

Formally, a CROW PRAM algorithm is a CREW PRAM algorithm for which there exists a function $owner(i, n)$ such that on any input of length n processor p attempts to write into global memory location i only if $p = owner(i, n)$. The owner function should be "simple to compute". For example in [DR86] the owner function must be log-space computable and oblivious, i.e., the owner of a location is independent of the input, except for its length.

Instead of using arrays, we use doubly-linked lists as the basic data structure. The linked lists will be implemented in clusters of three memory locations containing contents and addresses of predecessor and successor. Each cluster is owned by a processor. All sorted items are stored in doubly-linked lists implemented this way. (Linked lists as the basic data structure have also been used by Goodrich and Kosaraju [GK89] in their $O(\log n)$ sorting algorithm on their CREW Parallel Pointer Machine model.) For sorting n distinct numbers, initially the input is distributed one item per leaf on the leaves of the n node tree. The sorting process at a node of the tree consists of repeated (pipelined) merging of two sorted linked lists into one sorted linked list, using auxiliary lists which are samples of the lists to be sorted. More specifically, let u be an internal node of the tree, and v be its left child node and w, its right child node. The computation on a node is to merge two sorted linked lists on its children nodes. The merging procedure proceeds in $O(\log n)$ stages. At stage t, node u merges in constant time sublists of sorted linked lists created on node v and w at stage $t-1$ to produce the sorted linked list on node u of stage t. The computation on node u will finish when the size of the linked list reaches its full size containing all the items initially at the subtree rooted at u. When the root becomes full, the sorting finishes.

On the CROW PRAM model with its owner write restriction, the method of merging two sorted linked lists at each stage is called divided-merging. In divided-merging, no processor writes into any other processors' memory. The basic idea of divided-merging is that two sorted linked lists are broken down into smaller sorted linked lists. Then the smaller linked lists are merged in parallel. Those results are connected together producing a final sorted linked list. As in Cole's algorithm, the merge is accomplished in constant time with the aid of auxiliary lists of samples of the lists to be merged, which have been computed as the result of the previous stage, and which are linked to the corresponding elements of the lists to be merged. We will describe the divided-merging algorithm in more detail below.

We say a processor i *spins out* a new processor j when i creates j and initializes its data. In the CROW PRAM model, the processor j is a new processor with its own global memory locations. When processor i spins processor j out, processor i set up processor j's data, connects it into linked lists, etc. Alternatively, one can view this processor j, on initially becoming active, as entering a state in which it reads its initialization data from i.

3 Divided-Merging

Let u be an internal node of the tree with left child node v and right child node w. $LL_t(node)$ indicates the sorted linked list on *node* at end of time t. $SLL_t(node)$ indicates the sample list of $LL_t(node)$. The sample list is a sublist of the original list. It is used at the parent node to build the sorted linked list for the stage.

There are some inductive assumptions at time t. For each item e in $LL_t(u)$, there are left and right predecessor-pointers, $LPtr_t(u)[e]$ and $RPtr_t(u)[e]$. They are pointing to e's predecessors in $SLL_{t-1}(v)$ and $SLL_{t-1}(w)$. One of the predecessors is e itself, because of the invariant $LL_t(u) = SLL_{t-1}(v) \cup SLL_{t-1}(w)$ i.e. $LL_t(u)$ is the result of the merge between $SLL_{t-1}(v)$ and $SLL_{t-1}(w)$. Each item of $SLL_{t-1}(v)$ has a predecessor pointer pointing to its predecessor in $SLL_t(v)$, and similarly for node w.

For time $t+1$, the computation on node u is to merge $SLL_t(v)$ and $SLL_t(w)$ into $LL_{t+1}(u)$. Let e be an item of $LL_t(u)$. By the above assumptions, we know that e has predecessor pointers pointing to its predecessors in $SLL_{t-1}(v)$ and $SLL_{t-1}(w)$. We also know that each item of $SLL_{t-1}(v)$ has predecessor pointer pointing to the item's predecessor in $SLL_t(v)$ (and similar for node w). In Cole's sorting algorithm, $SLL_{t-1}(v)$ would be a 3-cover of $SLL_t(v)$, so the item e of $LL_t(u)$ can easily find its predecessors in $SLL_t(v)$ and $SLL_t(w)$. (That is, between two elements of the array at a node, there are at most three elements in the new array on the next time unit.)

Now we describe how $SLL_t(v)$ and $SLL_t(w)$ are partitioned into divisions by predecessor pointers from $LL_t(u)$ to $SLL_t(v)$ and $SLL_t(w)$. Let e_1 and e_2 be two adjacent items in $LL_t(u)$, ev_1 be e_1's predecessor in $SLL_t(v)$ and ev_2 be e_2's predecessor in $SLL_t(v)$ (it is analogous for node w). By 3-cover property ($SLL_{t-1}(u)$ is 3-cover of $SLL_t(u)$ by corollary 3.1), so we know that there are less than a constant number of items in $SLL_t(v)$ between ev_1 and ev_2 (it is analogous for node w). Here, we call the range $(ev_1, ev_2]$ a *division*. In this way, linked lists $SLL_t(v)$ and $SLL_t(w)$ are partitioned into divisions by the predecessor pointers from $LL_t(u)$ to $SLL_t(v)$ and $SLL_t(w)$. Divisions $(ev_1, ev_2]$ and $(ew_1, ew_2]$ include all the items of $LL_{t+1}(u)$ within $(e_1, e_2]$. Therefore, the processors associated with e_2 can read the items within the divisions $(ev_1, ev_2]$ and $(ew_1, ew_2]$ over to its own memory, sort those items and then spin out new processors each with one item. These items form a small sorted linked list. These linked lists spun out by all the items of $LL_t(u)$ connect with their neighbor sorted linked lists to form $LL_{t+1}(u)$. A new processor associated with item e of $LL_{t+1}(u)$ spun out by e_2 will easily find out its predecessors in $SLL_t(v)$ and $SLL_t(w)$. One of the predecessors is itself. Let e come from $SLL_t(v)$ (coming from $SLL_t(w)$ is analogous). Through the predecessor pointer of e_2 to $SLL_t(v)$, the processor associated with e of $LL_{t+1}(u)$ can find item e in $SLL_t(v)$ and set a predecessor pointer to it. It is easy to see that e's predecessor in $SLL_t(w)$ must be in the range $[ew_1, ew_2]$. Through the predecessor pointer of e_2 to $SLL_t(w)$, the processor associated with e of $LL_{t+1}(u)$ can find its predecessor in $SLL_t(w)$ and set a predecessor pointer to it in constant time.

4 Sampling

The number of items in the original array or list between two adjacent items of the sample array or list is called the *sample-range.* In CROW PRAM sorting, a non-fixed sample-range is used. Let $[x, y]$ denote the numbers between x and y, inclusive. The inductive assumptions are:

Assumption 1. At time t, the sample-range of $SLL_t(node)$ over $LL_t(node)$ is $[3, 6]$.

Assumption 2. Each item of $LL_t(u)$ has predecessor pointers to $SLL_{t-1}(v)$ and $SLL_{t-1}(w)$. Each item of $SLL_{t-1}(node)$ has a predecessor pointer pointing to its predecessor in $SLL_t(node)$. $SLL_{t-1}(node)$ is 2-cover of $SLL_t(node)$.

Given the above assumptions of linked lists for time t, the following 2 steps show how $SLL_t(v)$ and $SLL_t(w)$ are divided-merged into $LL_{t+1}(u)$ and how $SLL_{t+1}(u)$ is found from $LL_{t+1}(u)$.

Step 1 Using divided-merging to construct $LL_{t+1}(u)$: If u is full, then we simply make $LL_{t+1}(u) = LL_t(u)$. If u is not full, then we divided-merge $SLL_t(v)$ and $SLL_t(w)$ using $LL_t(u)$. In the last section we described using $LL_t(u)$ to divided-merge $SLL_t(v)$ and $SLL_t(w)$ into $LL_{t+1}(u)$. Then each item of $LL_{t+1}(u)$ finds its predecessors in $SLL_t(v)$ and $SLL_t(w)$. Lemma 1 below states that $LL_{t+1}(u)$ can be produced in constant time.

Step 2 To construct $SLL_{t+1}(u)$: In step 1, we had $LL_{t+1}(u)$. It was spun out by $LL_t(u)$ using divided-merging. The processors associated with items of $SLL_t(u)$ can find their predecessors in $LL_{t+1}(u)$ in constant time.

In order to maintain the number of items between two adjacent elements r_i and r_j of $LL_{t+1}(u)$ within $[3, 13]$, we initiate *pointer moving.* Predecessor pointers from items of $SLL_t(u)$ to items of $LL_{t+1}(u)$ are going to move. In an example, the predecessor pointer from item e of $SLL_t(u)$ to item r of $LL_{t+1}(u)$ is going to point to r' which is on the left side of r in $LL_{t+1}(u)$. After the pointer moving, the currently pointed item r' is not e's predecessor in $LL_{t+1}(u)$. But r' will be e's predecessor in $SLL_{t+1}(u)$. Let item e of $LL_t(u)$ come from $SLL_{t-1}(v)$ (coming from $SLL_{t-1}(w)$ is analogous) and its predecessor in $SLL_t(v)$ be rv. ew_1 is e's predecessor in $SLL_{t-1}(w)$ and ew_2 is e's successor in $SLL_{t-1}(w)$. rw_1 is ew_1's predecessor in $SLL_t(w)$ and rw_2 is ew_2's predecessor in $SLLt(w)$. From Assumption 2, r is e's predecessor in $LL_{t+1}(u)$. There could be an item x between rw_1 and rw_2. When to move, and how far to move, depends on whether or not x exists, and the cases are described fully in the complete paper.

Lemma 3 shows that after moving the pointers, the number of items between any two adjacent currently pointed items in $LL_{t+1}(u)$ is $[3, 13]$.

After moving the pointers, we can build the linked list $SLL_{t+1}(u)$. These currently pointed items and if the number of items between any two adjacent currently pointed items is greater than 6 then the median item of the two adjacent currently pointed items are the sample list $SLL_{t+1}(u)$. The processors associated with items of SLL_t read those items over and spin out a new list $SLL_{t+1}(u)$. It is easy to see $SLL_t(u)$ and $SLL_{t+1}(u)$ keep in line with Assumption 2.

After the completion of step 2, the algorithm is ready to begin the next step.

Lemma 1. *In divided-merging, if $LL_t(u)$, $SLL_{t-1}(v)$, $SLL_{t-1}(w)$, $SLL_t(v)$ and $SLL_t(w)$ are satisfied with Assumption 2, then we can merge $SLL_t(v)$ and $SLL_t(w)$ into $LL_{t+1}(u)$ in constant time.*

Proof: Due to space requirements, proofs are omitted in this version.

Lemma 2. *The number of items in $LL_{t+1}(u)$ between r_i and r_j is $[2, 15]$.*

Lemma 3. *After pointer moving, in $LL_{t+1}(u)$, between any two adjacent pointed items there are $[3, 13]$ items.*

Theorem 4. *There is a CROW PRAM merge sort algorithm that sorts n items in $O(\log n)$ time using $O(n \log n)$ processors.*

Notes: We thank T. Papadakis for helpful comments and suggestions on an early version of the manuscript. This paper is based on results in the M.Sc. thesis of the first author, supervised by the second and third authors. Research supported by Natural Sciences and Engineering Research Council. A full version of this paper may be found at **http://www.cs.yorku.ca/~dymond/papers**

5 References

[**AKS83**] Ajtai, M., Komlos, J., Szemeredi, E.: An $O(n \log n)$ Sorting Network. Combinatorica **3** (1983) 1–19

[**C88**] Cole, R.: Parallel Merge Sort. SIAM Journal on Computing **17** (1988) 770–785

[**C93**] Cole, R.: Parallel merge sort. Synthesis of Parallel Algorithms (edited by Reif, J.) Morgan Kaufmann Publishers (1993) 453–496

[**DR86**] Dymond,P.W. and Ruzzo, W.L.: Parallel Random Access Machines With Owned Global Memory And Deterministic Context-Free Language Recognition (Extended Abstract). Automata, Languages, and Programming: 13th International Colloquium. Springer Verlag Lecture Notes in Computer Science **226** (1986) 95–104

[**DR97**] Dymond,P.W. and Ruzzo, W.L.: Deterministic context-free language recognition and parallel RAMs with owned global memory. York University Department of Computer Science Technical Report CS-97-02 and University of Washington Department of Computer Science Technical Report 97-02 (1997)

[**FW78**] Fortune, S. and Wyllie, J.C.: Parallelism in random access machines. Tenth Annual ACM Symposium on Theory of Computing (1978) 114–118

[**GK89**] Goodrich, M.T. and Kosaraju, S.R.: Sorting on a parallel pointer machine with applications to set expression evaluation. Thirtieth Annual IEEE Symposium on Foundations of Computer Science (1989) 190–195

[**Ro91**] Rossmanith, P.: The owner concept for PRAMs. Eighth Annual Symposium on Theoretical Aspects of Computer Science. Springer Verlag Lecture Notes in Computer Science **480** (1991) 172–183