Parallel Processing Letters
© World Scientific Publishing Company

PRAM MEMORY ALLOCATION AND INITIALIZATION

LISA HIGHAM
Department of Computer Science, University of Calgary
Calgary, Alberta T2N 4NC, Canada

and

ERIC SCHENK
Department of Computer Science, University of Toronto
Toronto, Ontario M5A 1S5, Canada

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Two useful and practical techniques for managing memory on a parallel random access machine (PRAM) are presented. One is a scheme for an $n/\log n$ processor EREW PRAM that dynamically allocates and deallocates at most n records in $O(\log n)$ time. The other is a simulation of a PRAM with initialized memory by one with uninitialized memory. A CREW PRAM variant of the technique justifies the assumption that memory can be assumed to be appropriately initialized with no asymptotic increase in time but a factor of n increase in space. An EREW PRAM solution incurs a factor of $O(\log n)$ increase in time but only a constant factor increase in space.

Keywords: Parallel Algorithms, PRAM, Memory Management, Initialization.

1. Introduction

Procedures for memory management are commonly assumed tools for algorithms that maintain dynamic data structures. Such tools have been thoroughly studied for sequential machines for decades. This paper presents two useful and practical techniques for managing memory on a parallel random access machines (PRAM). One is a scheme for an $n/\log n$ processor Exclusive Read Exclusive Write PRAM that dynamically allocates and deallocates memory to any subset of the processors in $O(\log n)$ time. The other is a scheme for simulating PRAMs possessing initialized memory with corresponding PRAMs possessing uninitialized memory.

A PRAM is a common abstraction of a parallel machine that is useful for the design and analysis of parallel algorithms. It is a collection of synchronized independent sequential processors with unique identifiers and a shared global memory. In each time step, each processor can read a location in the global memory, perform a local computation, and then write to a location in the global memory. In an EREW (Exclusive Read Exclusive Write) PRAM no two processors may simultaneously access the same memory location for either reading or writing; in a CREW (Concurrent Read Exclusive Write) PRAM, simultaneous reading, but not writing, is permitted. (See Karp and Ramachandran [1] for an overview of PRAM results.)

PRAM algorithms that manipulate dynamic data structures need to be able to secure and release memory from the global store in parallel. For example, Paul, Vishkin and Wagener present algorithms [2] for maintaining 2-3-trees that depend upon dynamic allocation and deallocation of memory. An extension to general B-trees [3] similarly depends on parallel memory management. Section 2 describes a general scheme for an $n/\log n$ processor EREW PRAM that dynamically allocates and deallocates memory to any subset of the processors in $O(\log n)$ time.

When designing PRAM algorithms it is sometimes convenient to assume that all memory is appropriately initialized. For example, Schenk uses this assumption to detect whether a given memory cell has been written during the course of the computation [4]. Since it is conceivable that this assumption may make a problem easier to solve than it would otherwise be, it is important to determine its cost in terms of time, processors and memory. A technique from the folklore of computer science [5], simulates initialized memory with uninitialized memory for a sequential random access machine with only a constant factor increase in time and space. In section 3 we adapt this technique to the n processor CREW PRAM setting with no asymptotic increase in time but a factor of n increase in memory size, and to an EREW PRAM setting with a factor of $O(\log n)$ increase in time and a constant factor increase in space.

2. Memory Management

This section describes algorithms and data structures that maintain PRAM memory in such a way that deallocated memory is captured and reused rather than new memory being consumed. We present algorithms for dynamically allocating and deallocating sets of n or fewer records of a fixed size on an $n/\log n$ processor EREW PRAM in $O(\log n)$ time.

2.1. Data Structures

The algorithms maintain all available memory in two data structures using an additional array $w_0, \ldots, w_{3n}$ for working space. A contiguous array, $mem_{start}, mem_{start+1}, \ldots$ contains the memory that has never been allocated. A linked list of balanced binary trees called the *free list* contains deallocated records. The first tree in the free list contains at most $2n$ records and if there are two or more trees then the first tree has at least n records. All other trees in the list contain exactly n records. Initially all available memory is in the array; that is, *start* is set to one and the free-list is empty.

We assume that the size of the record, say *record-size*, is large enough to contain two pointers, *left* and *right*, which are used to maintain the free list structure. One record in each tree is used to maintain the linked list of trees by having *left* point to the actual tree stored in the linked list, and *right* point to the next record in the linked list. Other records in the free list use *left* and *right* as usual to point to the roots of left and right subtrees. The variable *head* points to the head of the free list, and *size* counts the number of records in the tree at the head of the list (including the extra record that forms the linked list).

2.2. Subroutines

Two tree manipulation routines are required by the allocation algorithm: one constructs a balanced binary tree out of an array, the other maps a tree into an array.

Constructing a balanced binary tree out of the records pointed to by an array $w_0, \ldots, w_m$ is done by mapping element 1 onto the root, and elements $2i$ and $2i+1$ onto the children of i. Note that the first element of the array stores the link to the next tree in the free list.

```
procedure construct-tree($w_0, \ldots, w_m$)
    for $i \in \{1, \ldots, m\}$ pardo
        if $2i \leq m$ then $w_i.left \leftarrow w_{2i}$ else $w_i.left \leftarrow$ null
        if $2i+1 \leq m$ then $w_i.right \leftarrow w_{2i+1}$ else $w_i.right \leftarrow$ null
    $w_0.left \leftarrow w_1$, $w_0.right \leftarrow$ null.
```

On an $n/\log n$ processor EREW PRAM, this procedure can be performed optimally in $O(m \log n/n + \log n)$ time by assigning $\lceil m \log n/n \rceil$ iterations of the pardo loop to each processor. The extra $\log n$ term is required to broadcast the value of m to all processors.

Mapping the nodes of a balanced binary tree of size m onto an array $w_0, \ldots, w_m$ can be accomplished in $O(m/p + \log m)$ time on a p processor EREW PRAM as follows. First the root of the tree is mapped onto element w_1. Then, using $\lfloor \log p \rfloor$ iterations of a parallel loop, the top $\lfloor \log p \rfloor + 1$ levels of the binary tree are mapped onto the array. The mapping process is completed by assigning a processor to each node at level $\lfloor \log p \rfloor + 1$. Each processor then recursively maps the subtree found at its assigned node into the array. The node that maintains the tree's position in the linked list is mapped into element w_0 of the array.

```
procedure $map\text{-}tree(T, w_0, \ldots, w_m)$
  for $i \in \{1, \ldots, m\}$ pardo $w_i \leftarrow$ null
  $w_0 \leftarrow T, w_1 \leftarrow T.left$
  for $i \in \{1, \ldots, p\}$ pardo
    loop $\lfloor \log p \rfloor$ times
      if $w_i \neq$ null then
        if $w_i.left \neq$ null then $w_{2i} \leftarrow w_i.left$
        if $w_i.right \neq$ null then $w_{2i+1} \leftarrow w_i.right$
    if $w_{2i} =$ null and $w_i.left \neq$ null then
      $sequential\text{-}map\text{-}tree(w_i.left, 2i, w_0, \ldots, w_m)$
    if $w_{2i+1} =$ null and $w_i.right \neq$ null then
      $sequential\text{-}map\text{-}tree(w_i.right, 2i+1, w_0, \ldots, w_m)$.

procedure $sequential\text{-}map\text{-}tree(T, i, w_0, \ldots, w_m)$
  $w_i \leftarrow T$
  if $T.left \neq$ null then $sequential\text{-}map\text{-}tree(T.left, 2i, w_0, \ldots, w_m)$
  if $T.right \neq$ null then $sequential\text{-}map\text{-}tree(T.right, 2i+1, w_0, \ldots, w_m)$.
```

Setting p to $n/\log n$ processors, this procedure takes $O(m \log n/n + \log m)$ time.

2.3. Allocation and Deallocation

Allocation of $k \leq n$ records, if $k > size$, is from the front of the array *mem*. Otherwise the k required records are allocated from the tree at the head of the free list. If this tree contains fewer than n records after k have been removed, then it is merged with the next tree (if any) in the list. Using the procedures given in subsection 2.2, this requires $O(\log n)$ time on an $n/\log n$ processor EREW PRAM.

```
procedure $allocate(t, r_1, \ldots, r_k)$
  if $size < k$ then [Allocate from the memory array]
    for $i \in \{1, \ldots, k\}$ pardo set $r_i$ to point to $mem_{start+(i-1)\cdot record\text{-}size+1}$
    $start \leftarrow start + k \cdot record\text{-}size$
  else [Allocate from the head of the free list]
    $map\text{-}tree(head, w_1, \ldots, w_{size})$
    $head \leftarrow head.right, size \leftarrow size - k$
    for $i \in \{1, \ldots, k\}$ pardo $r_i \leftarrow w_{size+i}$
    if $size > 0$ then
      if $size < n$ and $head \neq$ null then [Add the next tree to the array as well]
        $map\text{-}tree(head, w_{size+1}, \ldots, w_{size+n})$
        $head \leftarrow head.right, size \leftarrow size + n$
      [Put the array back into a tree]
      $construct\text{-}tree(w_1, \ldots, w_{size})$
      $w_1.right \leftarrow head, head \leftarrow w_1$.
```

Deallocation of $k \leq n$ records is accomplished by adding them to the tree at the head of the free list. If the resulting tree has more than $2n$ records, n records are formed into a separate tree and inserted just after the head of the free list. Using the procedures given in subsection 2.2, this requires $O(\log n)$ time on an $n/\log n$ processor EREW PRAM.

```
procedure $deallocate(t, r_1, \ldots, r_k)$
    if $head \neq$ null then
        $map\text{-}tree(head, w_1, \ldots, w_{size})$
        $head \leftarrow head.right$
    for $i \in \{1, \ldots, k\}$ pardo $w_{size+i} \leftarrow r_i$
    $size \leftarrow size + k$
    if $size > 2n$ then
        $construct\text{-}tree(w_{size-n+1}, \ldots, w_{size})$
        $w_{size-n+1}.right \leftarrow head$, $head \leftarrow w_{size-n+1}$, $size \leftarrow size - n$
    $construct\text{-}tree(w_1, \ldots, w_{size})$
    $w_1.right \leftarrow head$, $head \leftarrow w_1$.
```

Taking these results together, we have the following theorem.

Theorem 1 *For $k \leq n$, k records of the same size, say s, can be allocated or deallocated in* $O(\log n)$ *time on an $n/\log n$ processor EREW PRAM. Furthermore, for all times t, the total space used is $(A_t - D_t)s + O(n)$, where A_t (respectively D_t) is the total number of records allocated (respectively deallocated) at or before time t.*

To manage different record sizes, separate free lists can be maintained for each size of record. It is not necessary to maintain more than one array *mem*. Allocations of each record size still require the cooperation of all $n/\log n$ processors.

By replacing the linked list of trees with a linked list of arrays, our algorithms can be transformed into constant time n processor CREW PRAM algorithms for allocating and deallocating exactly n records. This entails separately allocating arrays of size n and allocating and deallocating them as the free list grows or shrinks. This increases the space usage by at most m. However, if fewer than n records are allocated or deallocated, the time required by our CREW variants increases beyond constant time.

3. Initialization

This section describes two simulations of an n processor PRAM with initialized memory by an n processor PRAM with uninitialized memory. Let α be an algorithm for an n processor EREW PRAM with initialized memory and let $t_\alpha(n)$ be the time complexity and $s_\alpha(n)$ be the space complexity of α on inputs of size n. The first simulation, $S(\alpha)$, simulates α on an n processor EREW PRAM with uninitialized memory in time $O(t_\alpha(n)\log n)$ and $3s_\alpha(n) + 1$ space. Let β be an algorithm for an n processor CREW PRAM with initialized memory and let $t_\beta(n)$ be the time complexity and $s_\beta(n)$ be the space complexity of β on inputs of size n. The second simulation, $R(\beta)$, simulates β on an n processor CREW PRAM with uninitialized memory in $O(t_\beta(n))$ time and $(n+3)s_\beta(n)$ space. Both simulations maintain a data structure in such a way that it is easy to distinguish between a value that has been written and uninitialized garbage.

3.1. EREW PRAM simulation

Denote the memory array of the n processor EREW PRAM with initialized memory by M. On the n processor EREW PRAM with uninitialized memory, create one extra integer variable, *next-space*, and $3n$ cells of working space, and partition the remaining memory into three arrays, A, B and C by interleaving. Array A is used to simulate memory M. Arrays B and C and the variable *next-space* are used to keep track of those positions of M that have been written. Location l of array X is denoted $X[l]$.

Simulation S maintains the following two part invariant for every location l in M: (1) $M[l]$ has been written by α if and only if $B[l] \in \{1, \ldots, next\text{-}space - 1\}$ and $C[B[l]] = l$; and (2) if $M[l]$ has been written by α then $A[l]$ contains the last value written to $M[l]$.

Setting *next-space* to one ensures that the invariant holds initially. Suppose, at some time step of algorithm α, processors $1, \ldots, n$ access memory locations $M[l_1], \ldots, M[l_n]$ respectively, and suppose the invariant holds before simulating this step. S has each processor j first determine whether the $M[l_j]$ has already been written. If the memory access instruction is a read, then for each processor j, if $M[l_j]$ has been previously written then j reads $A[l_j]$; otherwise $M[l_j]$ has not been written so j assumes the initial value. If the memory access instruction is a write, then first those processors writing to memory locations that have not been previously written, cooperate to update arrays B and C and the value of *next-space* to ensure that the invariant is maintained. Then each processor j writes to $A[l_j]$.

More specifically, define P_j to be the predicate $(B[l_j] \in \{1, \dots, \textit{next-space} - 1\} \wedge C[B[l_j]] = l_j)$. According to the invariant, $M[l_j]$ has been written if and only if P_j. Let *write-check*$(l_1, \dots, l_n,\ x_1, \dots, x_n,\ y_1, \dots, y_n)$ be a procedure that for $j \in \{1, \dots, n\}$ sets x_j to P_j and sets y_j to the size of the set $\{i \leq j : \neg P_i\}$.

Given *write-check*, the following procedures perform, respectively, one parallel read and write step of the simulation S. Procedure *read* sets v_j to the value in $M[l_j]$. Procedure *write* sets $A[l_j]$, which simulates $M[l_j]$, to the value v_j.

```
    procedure read$((l_1, v_1), \dots, (l_n, v_n))$
1.    write-check$(l_1, \dots, l_n,\ x_1, \dots, x_n,\ y_1, \dots, y_n)$
2.    for $j \in \{1, \dots, n\}$ pardo
        if $x_j$ then $v_j \leftarrow A[l_j]$ else $v_j \leftarrow$ initial value.

    procedure write$((l_1, v_1), \dots, (l_n, v_n))$
1.    write-check$(l_1, \dots, l_n,\ x_1, \dots, x_n,\ y_1, \dots, y_n)$
2.    for $j \in \{1, \dots, n\}$ pardo
        if $\neg x_j$ then
          $B[l_j] \leftarrow \textit{next-space} + y_j - 1$, $C[\textit{next-space} + y_j - 1] \leftarrow l_j$
        $A[l_j] \leftarrow v_j$
3.    $\textit{next-space} \leftarrow \textit{next-space} + y_n$.
```

It is easily checked from the pseudo-code that the invariant holds after *read* or *write* provided that it held before this step. Clearly step two of *read* and steps two and three of *write* complete in constant time on an n processor EREW PRAM.

It remains to provide more details for *write-check*. The values y_1 through y_n can be computed from x_1 through x_n in $O(\log n)$ time on an EREW PRAM using Prefix Sums [1]. Therefore we need only to determine how to compute $x_1, \dots, x_n$ (that is $P_1, \dots, P_n$). Even though all l_j's are distinct, processors might read nondistinct positions in array C. Also, each processor needs to access *next-space*. Some care is needed to ensure exclusive access in the simulation. First all processors cooperate to lexicographically sort the pairs $(B[l_1], 1), \dots, (B[l_n], n)$, yielding $D_1, \dots, D_n$ where $D_j = (B[l_k], k)$, for some $k \in \{i, \dots, n\}$. Let $D_{j,1}$ (respectively $D_{j,2}$) denote the first (respectively second) coordinate of D_j. The processors then determine all maximal intervals $[i \dots k]$ such that $D_{i,1} = D_{k,1}$. For each such interval $[i \dots k]$, processor i reads $C[D_{i,1}]$ and initiates a broadcast of the result to the remaining processors in its interval. Next a broadcast is used to distribute the value of *next-space* to each processor. Each processor j can now determine whether $D_{j,1} \in \{1, \dots, \textit{next-space} - 1\}$ and whether $C[D_{j,1}] = l_{D_{j,2}}$ without conflict. If either test fails, then $x_{D_{j,2}}$ is set to `false`, otherwise $x_{D_{j,2}}$ is set to `true`. Broadcast is a basic technique that can be achieved on an n processor EREW PRAM in $O(\log n)$ time [6]. Cole's parallel merge sort [7] can be used to sort in $O(\log n)$ time. All other operations requires just constant time.

Theorem 2 *Let α be an algorithm for an n processor EREW PRAM with initialized memory taking $t_\alpha(n)$ time and using $s_\alpha(n)$ space. Then $S(\alpha)$ simulates α on an n processor EREW PRAM with uninitialized memory in time* $O(t_\alpha(n) \log n)$ *and space* $3s_\alpha(n) + 3n + 1$.

3.2. CREW PRAM simulation

In the EREW PRAM simulation, the function *write-check* uses $O(\log n)$ time in order to avoid read collisions of the variable *next-space* and (possibly) of locations in array C. If concurrent reads are permitted then determining P_1 through P_n can be achieved in constant time with no change in data structures. In step two of procedure *write* however, each processor writing to a new location needs to be allocated a new distinct record from the single array C and in step three the pointer *next-space* must be incremented by the number of such processors. Determining this number requires more than constant time even with concurrent reads. To circumvent this problem, we replace array C and variable *next-space* with a separate array C_j and variable *next-space*$_j$ for each $j \in \{1, \dots, n\}$. This ensures that in each step, at most one record will be allocated from any one array C_j and that each *next-space*$_j$ is incremented by at most 1.

Specifically, on the n processor CREW PRAM with uninitialized memory, create n extra integer variables, *next-space*$_j$ for $j \in \{1,\ldots,n\}$, and $3n$ cells of working space, and partition the remaining memory into $n+2$ arrays, A, B and C_j for $j \in \{1,\ldots,n\}$ by interleaving. Array A is used to simulate the memory, say M, of the n processor CREW PRAM with initialized memory. Entries in B are now a pair (p,i) where p is an integer between 1 and n and i is an index into array C_p.

Simulation R maintains the following two part invariant for every location l in M: (1) $M[l]$ has been written by β if and only if $B[l] = (p,i)$ where $p \in \{1,\ldots,n\}$ and $i \in \{1,\ldots,\mathit{next\text{-}space}_p - 1\}$ and $C_p[i] = l$; and (2) if $M[l]$ has been written by β then $A[l]$ contains the last value written to $M[l]$.

To initialize the structure, each processor j sets *next-space*$_j$ to 1 thus ensuring that the invariant holds initially. Suppose, as previously, at some time step of algorithm β, processors $1,\ldots,n$ access memory locations $M[l_1],\ldots,M[l_n]$ respectively, and suppose the invariant holds before simulating this step. Let P_j be the predicate $(B[l_j] = (p,i) \wedge p \in \{1,\ldots,n\} \wedge i \in \{1,\ldots,\mathit{next\text{-}space}_p - 1\} \wedge C_p[i] = l_j)$. Let *write-check*$'(l_1,\ldots,l_n, x_1,\ldots,x_n)$ be a procedure that sets x_j to P_j for $j \in \{1,\ldots,n\}$. Notice that since the model permits concurrent reads, *write-check*$'$ takes only constant time by having each processor j compute P_j independently.

A CREW PRAM read procedure is obtained from the procedure *read* by replacing *write-check* with *write-check*$'$. One parallel write step of the simulation R is as follows. For each $j \in \{1,\ldots,n\}$, v_j is the value to be written to $A[l_j]$, which simulates $M[l_j]$.

```
    procedure crew-write($(l_1,v_1),\ldots,(l_n,v_n)$)
1.      write-check'($l_1,\ldots,l_n, x_1,\ldots,x_n$).
2.      for $j \in \{1,\ldots,n\}$ pardo
            if $\neg x_j$, then
                $B[l_j] \leftarrow (j, \mathit{next\text{-}space}_j)$, $C_j[\mathit{next\text{-}space}_j] \leftarrow l_j$.
                $\mathit{next\text{-}space}_j \leftarrow \mathit{next\text{-}space}_j + 1$.
            $A[l_j] \leftarrow v_j$.
```

It is easily checked that the invariant holds after *crew-write* provided it held immediately before its execution. Since the model is exclusive write, all l_j are distinct and thus step two of *crew-write* takes constant time on a CREW PRAM.

Theorem 3 *Let β be an algorithm for an n processor CREW PRAM with initialized memory taking $t_\beta(n)$ time and using $s_\beta(n)$ space. Then $R(\beta)$ simulates β on an n processor CREW PRAM with uninitialized memory in time* $O(t_\beta(n))$ *and space* $(n+3)s_\beta(n) + 3n$.

4. Discussion and Open Problems

In our simulations, each instruction executed by the PRAM with initialized memory is emulated by a short sequence of instructions on the PRAM with uninitialized memory. Of course, for any memory cell not written during the run of the algorithm, the contents of the original and simulated memories may differ. To extend the simulations so that the final outputs are the same, we have to assume that the final output is actually written by the PRAM processors rather than just being declared as some arbitrary part of the contents of memory.

According to simulation R, it is justifiable to assume that memory is preinitialized when determining a problem's time complexity on a CREW PRAM. We do not know if such an assumption can be justified for an EREW PRAM. Furthermore, we do not know how to justify the assumption for a CREW PRAM without increasing the size of memory by a linear factor.

Acknowledgements

This research was supported in part by a research grant and a postgraduate scholarship, both from the Natural Sciences and Engineering Research Council of Canada.

References

1. Richard M. Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 17. Elsevier Science Publishers, Amsterdam, The Netherlands, and The MIT Press, Cambridge, Massachusetts, U.S.A., 1990.
2. W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Lecture Notes in Computer Science 143: Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 597–609. Springer Verlag, 1983.
3. Lisa Higham and Eric Schenk. Maintaining B-trees on an EREW PRAM. Technical Report 91/446/30, Department of Computer Science, University of Calgary, September 1991. Submitted for publication.
4. Eric Schenk. The parallel asynchronous recursion model. Master's thesis, Department of Computer Science, University of Calgary, Canada, 1992. Research Report No. 92/473/11.
5. K. Abrahamson. private communication.
6. Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, Great Britain, 1988.
7. Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, August 1988.