Linked Allocation for Parallel Data Structures

Marinus Veldhorst

RUU-CS-87-18
October 1987

Linked Allocation for Parallel Data Structures

Marinus Veldhorst

Technical Report RUU-CS-87-18
October 1987

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands

LINKED ALLOCATION FOR PARALLEL DATA STRUCTURES

Marinus Veldhorst

Department of Computer Science, University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht, The Netherlands.

ABSTRACT. It will be shown that by using linked allocation a P-RAM of K processors can maintain J stacks of total size S in space $O(JK+S)$ while each simultaneous update of the J stacks takes $O(\log K + \log \max_j |S_j|)$ time where $|S_j|$ is the size of the j^{th} stack. Thus, the use of linked allocation in PRAM algorithms is not necessarily more expensive than the use of array allocation.

1. Introduction. In the area of the design of efficient sequential algorithms, the use of linked allocation is essential. One reason for its success is that it allows for the postponement of the decision for which purpose a memory cell will be used: this decision can be taken on the very moment the memory cell is needed. This may lead to a lower upper bound of the memory used, compared with array allocation. For example, suppose J stacks S_j $(1 \le j \le J)$ are to be maintained; let it be given (by theoretical arguments) that the length of stack S_j is bounded by B_j and that the total length of all J stacks is bounded by a number B such that $B < \sum_{j=1}^{J} B_j$. In case of array allocation stack S_j would be maintained in an array of size B_j and each stack operation requires constant time. Thus, array allocation requires $O(\sum_{j=1}^{J} B_j)$ space. On the other hand, with linked allocation stack S_j can be maintained in $O(|S_j|)$ space, without an increase in the order of the time bound for a stack operation. Thus, with linked allocation a space bound $O(B)$ is obtained. This is especially important when B is an order of magnitude smaller than $\sum_{j=1}^{J} B_j$. Clearly we assume that arrays cannot be enlarged unless their contents is copied into a larger part of memory.

As for parallel algorithms, array allocation dominates in the literature, even for data structures like stacks (cf. [5], [3]). Using array allocation in the example mentioned above, stack S_j can be maintained in $O(K+B_j)$ space while each simultaneous update of J stacks by at most K processors takes $O(\log K)$ time (cf. [5]). The cooperation of processors leads to a term K in the space bound and a term $\log K$ in the time bound (compared with the case of sequential algorithms). Thus the total memory bound is $O(JK + \sum_{j=1}^{J} B_j)$.

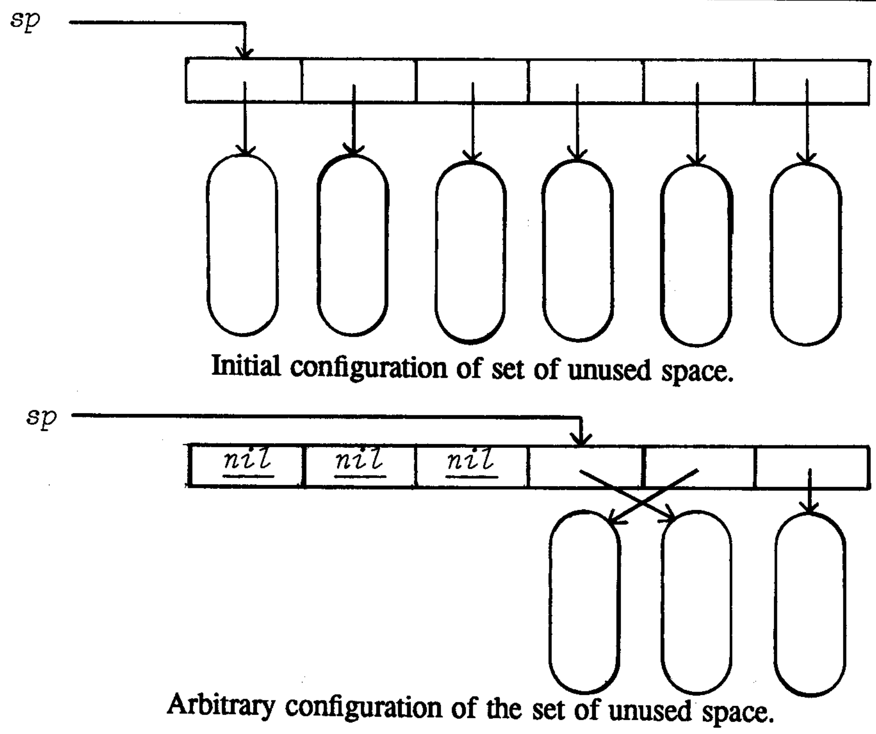
In this paper we will prove that a lower space bound of $O(K + |S_j|)$ for each stack S_j is possible while a simultaneous update of J stacks by at most K processors takes $O(\log K + \log \max_j |S_j|)$ time. This is a slight increase in the time bound (compared with array allocation), but the order of magnitude is not changed provided that the size of each stack is bounded by a polynomial in the number of processors.
Our main result can be applied in the parallel maximum flow algorithm of Shiloach and Vishkin (cf. [5]). Their use of array allocation for n stacks leads to an $O(mn)$ space bound (for an n-node graph with m edges) while theoretically only $O(n^2)$ of it is actually used. In [7] the space bound has been reduced to $O(n^2)$ by a rather ad-hoc method, still using array allocation. Incorporating our main result in the algorithm of Shiloach and Vishkin gives an $O(n^2)$ space bound and no increase in the order of magnitude of the time bound, provided that the number of processors is bounded from below by a power of the number of vertices of the graph under consideration.
The data structure we use for our main result is a generalization of the tree data structure that Leiserson and Maggs (cf. [2]) use for list contraction. We can use it for linked allocation of stacks, queues and double ended queues.

2. Machine model and maintenance of unused space.

The machine model used in this paper is the P-RAM in which processors may be enabled and disabled. At each moment all enabled processors execute the same instruction (possibly on different data). All processors have access to a shared memory, but a simultaneous write to the same location by several processors is not allowed. The algorithm under consideration is incorrect in case a simultaneous write occurs. The amount of time to access one memory location is assumed to be bounded by a constant.

With the use of linked allocation it is necessary to provide tools for the maintenance of unused space. Processors may simultaneously ask for or return unused space. We do not assume that one processor is in charge of unused space; thus processors have to cooperate to establish a good maintenance of unused space. Actually, unused space can be considered as a stack, but because of its special character we will implement it by a combination of array and linked allocation: it consists initially of an array of pointers to objects that can serve as nodes for user defined stacks (see fig. 1). With the array is associated a stackpointer sp, which indicates the left most array element pointing to an object.

As for the request or return of unused space, enabled processors must be scheduled in order to prevent that different processors receive the same object, respectively, objects are lost. For this purpose we use a partial sums tree as introduced by Shiloach and Vishkin (cf. [5]).



Initial configuration of set of unused space.

Arbitrary configuration of the set of unused space.

Figure 1. Data structure for unused space.

DEFINITION 1. An access tree for K processors is a complete binary tree of $2^{\lceil \log K \rceil}$ leaves; the K leftmost leaves are called active and leaf i is associated with processor PE_i. Each internal node x is the root of a complete subtree T_x and contains the sum of the leaves of T_x (we call this the partial sums property).

Because K is fixed, an access tree can be implemented in an array of size $2K-1$. We use the following operations (precise descriptions of the first three of them can be found in [5]):

$CLEAR(i)$	sets leaf i to value 0 and restores the partial sums property;
$UPDATE(i, a_i)$	sets leaf i to value a_i and restores the partial sums property;
$SUM(i)$	returns the sum of the values of the i leftmost leaves;
$RANK(i)$	returns the size of the set $\{j<i$: leaf j has value 1 $\}$.
$NEXTRANK(i)$	returns the smallest integer j such that $SUM(j)=RANK(i)+2$.

LEMMA 1. The operations *CLEAR*, *UPDATE*, *SUM*, *RANK* and *NEXTRANK* use $O(\log K)$ time, both in the sequential as well as in the parallel model.

The data structure for unused space consists of three parts:

(1) an array *unused* of pointers to objects that can serve as nodes for user defined stacks;

(2) a stackpointer *sp* which indicates the left most *unused* entree pointing to an object;

(3) an access tree *SPACCESS*.

Requesting and returning space can be done as follows:

```
OPERATION RETURN for enabled processors $PE_i$:
    (* $PE_i$ has a pointer $p_i$ to the object to be returned *)
    All enabled processors $PE_i$ execute the following algorithm:
    begin
    (1)  $UPDATE(i,1)$ in $SPACCESS$
    (2)  $r[i] := RANK(i)$ in $SPACCESS$
    (3)  $unused[sp+1+r[i]] := p_i$
    (4)  if $r[i]=0$ then $sp := sp+$ value in root of $SPACCESS$ endif
    (5)  $CLEAR(i)$ in $SPACCESS$
    end
```

```
OPERATION REQUEST for enabled processors $PE_i$:
    (* $PE_i$ has a pointer $p_i$ to which must be assigned an unused object *)
    All enabled processors $PE_i$ execute the following algorithm:
    begin
    (1)  $UPDATE(i,1)$ in $SPACCESS$
    (2)  $r[i] := RANK(i)$ in $SPACCESS$
    (3)  $p_i := unused[sp-r[i]]$;   $unused[sp-r[i]] :=$ nil;
    (4)  if $r[i]=0$ then $sp := sp-$ value in root of $SPACCESS$ endif
    (5)  $CLEAR(i)$ in $SPACCESS$
    end
```

In lines (1), PE_i states that it wants to access unused space; in line (2) it computes how many PEs with smaller index wants to access also; in line (3) the actual access occurs; in line (4) the stack pointer of unused space is updated and in line (5) the access tree *SPACCESS* is cleared for correct use in the future. We assume that at the start of any call of *REQUEST* or *RETURN*, all leaves of SPACCESS have value 0.

LEMMA 2. If, with the above mentioned organization of unused space, at most K processors call simultaneously $REQUEST$, respectively, $RETURN$, then they are finished in time $O(\log K)$.
Proof. Follows immediately from the algorithms and the previous lemma. Q.E.D.

Observe that we do not state that the P-RAM consists of at most K processors. It might have more processors but only K of them will ever call REQUEST and RETURN.

3. Implementation of parallel stacks with linked allocation.

Suppose there are K processors $PE_0, \ldots, PE_{K-1}$ and J stacks $S_1, \ldots, S_J$. At time t each processor PE_i has a number $s(i)$ $(0 \le i < K,\ 1 \le s(i) \le J)$ and either each active PE_i wants to push an element a_i on stack $S_{s(i)}$ or each active PE_i wants to pop a number from stack $S_{s(i)}$ into memory location a_i. The effect of a (simultaneous) pop or push is considered to be correct if it could be obtained by sequentializing the processors involved (this is in accordance with the sequentializing principle as explained in [4]). Thus, if $PE_{i_1}, PE_{i_2}, \ldots, PE_{i_k}$ $(k \le K)$ want to push onto stack S_1, then the numbers $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ must be added to S_1 in some order but the precise order is immaterial.

Obviously, a pushing operation is done by first allocating space, assigning to it the numbers to be pushed and finally incorporating this space in the data structure of the appropriate stack. Similarly, a pop operation runs in reverse order and returns the space that is not needed anymore.

DEFINITION 2. A 2-3 tree T is called enriched if with T is associated a spare node $spare(T)$ and with each internal node u of T with 3 sons is also associated a unique spare node $spare(u)$.

Observe that $spare(T)$ must be different from $spare(root(T))$ if the latter exists. Moreover, a proper subtree of an enriched 2-3 tree is not enriched.

PROPOSITION 3. Let T be an enriched 2-3 tree. Then the number of internal and spare nodes equals the number of leaves.
Proof. By induction. Q.E.D.

LEMMA 4. The standard operations INSERT, DELETE, SPLIT and CONCATENATE on enriched 2-3 trees with N leaves can be done sequentially in $O(\log N)$ time with $O(1)$ additional space.

Proof. Modify the operations (cf. [1]) such that spare nodes change into internal nodes and vice versa. Q.E.D.

Without proof we state the following proposition.

COROLLARY 5. Let T_1 and T_2 be enriched 2-3 trees. Then the tree $CONCATENATE(T_1,T_2)$ contains the same number of nodes as T_1 and T_2 together.

A similar result holds for the SPLIT operation. All this means that concatenation and splits of enriched 2-3 trees can be executed without using (temporarily) any unused space.

Now we will show how processors can update simultaneously a number of stacks. With each stack S_j is associated a access tree $ACCESS(j)$ of at least K leaves. At the beginning of each update operation, the values of the leaves of $ACCESS(j)$ must all be zero. Each processor PE_i knows its index i and can access leaf i of $ACCESS(j)$ in constant time. With each leaf of $ACCESS(j)$ are associated 2 pointers (initially with value *nil*). One of them will be used to make a singly linked list of leaves with value 1, and the second pointer can be used for an enriched 2-3 tree. Each stack S_j will be implemented as an enriched 2-3 tree $T(j)$ with stack entries of S_j in the leaves of $T(j)$; each internal node x of $T(j)$ contains the number of leaves that are descendants of x.

Pushing k numbers unto stack S_j is done as follows:

(1) Make k enriched 2-3 trees, each of one leaf and assign the k numbers to these leaves; and combine them into one singly linked list.

(2) Make one enriched 2-3 tree $NT(j)$ of k leaves, using these k enriched 2-3 trees; this is done deterministically in a way similar to the construction of balanced binary trees (cf. [6]) and list contraction (cf. [2]).

(3) Concatenate $NT(j)$ with $T(j)$; this is done by one processor for each stack.

A precise description of the parallel push can be found in program A.

Popping k numbers from stack S_j is done similarly, but in reverse order:

(1) Split off from $T(j)$ an enriched 2-3 tree $NT(j)$ of k leaves; this is done by one processor for each stack S_j.

(2) Break up $NT(j)$ into a singly linked list of k enriched 2-3 trees, each of 1 leaf.

(3) Take the numbers from the leaves of these k trees, assign them to the appropriate locations and return the trees to the unused space.

THEOREM 6. k simultaneous calls of push or pop on at most J stacks by $k \le K$ processors takes $O(\log K + \log \max_j |S_j|)$ time. The total amount of space used for the maintenance of these stacks equals $O(JK + \sum_{j=1}^{J} |S_j|)$.

Each processor PE_i knows its index i, has a number $s(i)$ and a value a_i; it wants to push a_i unto stack $S_{s(i)}$. With each stack S_j is associated an access tree $ACCESS(j)$ and a memory location $total[j]$. With each leaf of $ACCESS(j)$ are associated two pointers *enrtree* and *next*. *enrtree* will be used for an enriched 2-3 tree and the pointer *next* is used to connect enriched 2-3 trees into a singly linked list. Processor PE_i is able to access leaf i of $ACCESS(j)$ in constant time.
Each enabled processor PE_i executes the following algorithm:

```
In   ACCESS(s(i))
do   UPDATE(i,1);    r[i] := RANK(i);
     if  r[i]=0   then  total[s(i)] := value in root   endif;
     next of leaf(i) := NEXTRANK(i);
     enrtree of leaf(i) := REQUEST;
       (*  enrtree consists of an enriched 2-3 tree of one leaf and associated with this tree is
           a spare node  *)
     leaf of enrtree of leaf(i) := a_i;
     while total[s(i)] ≥ 2
     do   if odd total[s(i)] and r[i] = total[s(i)]-2
          then enrtree of leaf(i) :=
                         CONCATENATE(enrtree of leaf(i), enrtree of leaf(next of leaf(i)));
               total[s(i)] := total[s(i)] - 1;
          endif;
          if even r[i] and r[i]<total[s(i)]
          then enrtree of leaf(i) :=
                         CONCATENATE(enrtree of leaf(i), enrtree of leaf(next of leaf(i)));
               next of leaf(i) := next of leaf(next of leaf(i));
               r[i] := r[i] div 2
          endif;
          if odd r[i] then  r[i] := -1  endif;
          if r[i]=0  then   total[s(i)] := total[s(i)] div 2    endif
     enddo;
     if r[i]=0  then CONCATENATE(T(s(i)), enrtree of leaf(i))    endif;
     next of leaf(i) := nil;  enrtree of leaf(i) := nil;  r[i] := -1;
     CLEAR(i)
enddo
```

Program A.   Algorithm for a simultaneous push.

Proof. Steps (1) and (2) of the push operation and steps (3) and (2) of the pop operation take $O(\log K)$ time, while step (3) of the push and step (1) of the pop operation requires $O(\log|S_{s(i)}|)$ time.
The enriched 2-3 tree $T(j)$ and $ACCESS(j)$ use $O(|S_j|)$ and $O(K)$ space, respectively. Thus, the total space bound is $O(JK + \sum_{j=1}^{J} |S_j|)$.
Q.E.D.

Obviously this results holds also for the operations on queues and double ended queues that are implemented with enriched 2-3 trees.
In Shiloach and Vishkin ([5]) also an operation *FIND* on a stack was used, dependent on the value of the numbers in the stack (that are always nonnegative):

$FIND(\alpha,k,\rho)$ Given α, *FIND* returns k and ρ satisfying $\sum_{i=1}^{k-1} a_i < \alpha \le \sum_{i=1}^{k} a_i$ and $\rho = \alpha - \sum_{i=1}^{k-1} a_i$, in which a_i is the i^{th} topmost element of the stack S under consideration.

This operation can also be implemented with an enriched 2-3 tree in $O(\log|S|)$ time. In each internal node x is stored the sum of the numbers in the leaves that are descendants of x. *FIND* is executed sequentially by one processor which walks down from the root of S to the approppriate leaf k.

4. Conclusion.

In this paper we have shown that linked allocation of data structures in parallel algorithms is feasible. This contradicts the impression one might have from many papers, that array allocation is the only efficient data structure for parallel algorithms. Thus we hope that in the future designers of parallel algorithms will use abstract data types to express the data structures they need.

Acknowledgements. The author wants to thank R. Heuvelmans for his work on the memory bound of the Shiloach and Vishkin parallel maximum flow algorithm, that lead to the more general problem of maintaining a number of parallel stacks.

REFERENCES

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[2] Leiserson, C.E. and B. Maggs, Communications efficient parallel algorithms, Proc. Intern. Conf. on Parallel Processing, IEEE, 1986, pp. 861-868.

[3] Paul, W., U. Vishkin and H. Wagener, Parallel dictionaries on 2-3 trees, Proc. 10th ICALP 1983, Springer Lecture Notes in Computer Science 154, Springer Verlag, Berlin, 1983, pp. 597-609.

[4] Schwartz, J., Ultracomputers, ACM Trans. on Programm. Lang. Syst. 2 (1980), pp. 484-521.

[5] Shiloach, Y. and U. Vishkin, An $O(n^2 \log n)$ parallel maxflow algorithm, Jrnl of Algorithms 3 (1982), pp. 128-146.

[6] Vishkin, U., Randomized speed-ups in parallel computation, Proc. 16th ACM Symposium on Theory of Computing, 1984, pp. 230-239.

[7] Vishkin, U., personel communication, 1987.