AN EFFICIENT PARALLEL GARBAGE COLLECTION SYSTEM AND ITS CORRECTNESS PROOF

H. T. Kung and S. W. Song

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

1. Introduction

In this paper we propose an efficient system to perform garbage collection in parallel with list operations, and prove the correctness of the system.

The system consists of two independent processes sharing a common memory. One process is performed by the list processor (LP) for list processing and the other process is performed by the garbage collector (GC) for marking active nodes and collecting garbage nodes. The system is designed from both the correctness and efficiency points of view (see Appendix I). Assuming that memory references are indivisible (see Section 4.1), the system satisfies the following properties:

P1. No critical sections are needed in the entire system.

P2. The time to perform the marking phase by the GC is independent of the size of memory, but depends only on the number of active nodes.

P3. Nodes on the free list need not be marked during the marking phase by the GC.

P4. Minimum overheads are introduced to the LP.

P5. Only two extra bits for encoding four colors are needed for each node.

Properties P1, P2, P3, and P4 are important for the performance of the system. But none of the previous systems satisfy all the four properties. (The system of Steele[1975] does not satisfy P1 and the system proposed by Dijkstra[1976] does not satisfy P2 and P3.)

We have analyzed the performance of the parallel garbage collection system proposed in this paper. The results show that the parallel system is usually significantly more efficient in terms of storage and time than the sequential stack algorithm (see Appendix II). The complete performance results are reported in another paper (Kung and Song[1977]). In this paper we shall be mainly concerned with the correctness of the system. We must be sure that a system is correct before studying its performance.

The correctness proof in this paper is not intended to be formal or completely rigorous. Given the complexity of the system, we feel that it is more important to have a proof which is readable and convincing.

This research is supported in part by the National Science Foundation under Grant MCS 75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422. The second author is supported by Fundacao de Amparo a Pesquisa do Estado de Sao Paulo under Grant 76/517 and the Institute of Mathematics and Statistics of the University of Sao Paulo, Brazil.

We give a summary of this paper. In Section 2, we define the data structure shared by the LP and the GC. In Section 3, the parallel system is proposed, i.e., the garbage collector's algorithm and the list processor's operations are defined. Basic assumptions and correctness criteria are given in Section 4. The main theorem for proving the correctness of the system is also stated in this section. In Section 5, an upper bound on the execution time of a marking phase is derived. Section 6 contains the proof of the main theorem for an auxiliary parallel system. In Section 7, by transforming the proof and results for the auxiliary system we establish the main theorem for the parallel system defined in Section 3. Some concluding remarks are stated in Section 8. In Appendix I, we show how the parallel system is derived, by using correctness and efficiency arguments. A summary of results on the performance of the parallel system is given in Appendix II.

2. The Data Structure

The data structure shared by the two processes consists of a directed graph and an output-restricted deque.

2.1 The Directed Graph

Let the nodes of the graph be labeled by integers 1,...,M, and the node labeled by n be in memory location n for all n=1,...,M. Node n (or simply n) is used to refer to either the node labeled by n or the pointer to it depending upon the context. For the purpose of this paper, we assume that each node contains three fields: a left pointer field, a right pointer field and a color field. A pointer field contains the pointer to a node, which is one of the integers 1,...,M, or the null pointer "Λ"; but a left pointer field may sometimes contain a special value "f" which is not Λ or any of the integers 1,...,M. The color field contains one of the following four colors: white, off-white, gray and black.

The pointer contained in the left or right pointer field of node n is called the left or right pointer of n and is denoted by n.left or n.right, respectively. The color contained in the color field of node n is called the color of n and is denoted by n.color.

The topology of the graph is determined by the pointers of the nodes in the graph. Let m and n be any two nodes. If m.left (or m.right) = n, we say that there exists a left (or right) outgoing edge from m to n, and n is the left (or right) son of m. An existing edge from m to n is often denoted by (m,n), if there is no need to indicate explicitly whether it is a left or right outgoing edge from m to n. In this case, we may also simply say that n is pointed to by m. If m.left (or m.right) = Λ, then we say that there does not exist a left (or right) outgoing edge from m to any node and m does not have a left (or right) son.

The graph is changed as the pointers of its nodes are altered by the processes. We assume that the left (or right) outgoing edge from m to n is created at the instant of completion of writing n on the left (or right) pointer field of m. Hence at any moment an edge either exists or does not exist, though writing on a pointer field takes a finite amount of time.

The first R nodes, 1,..., R, are called roots and node R is also called NEW. Node R+1 is called FREE. FREE.left is the pointer to the first node of a list, called the free list (cf. Fig. 1), which is a sequence of nodes, n_1, n_2,..., n_k, satisfying the following properties:

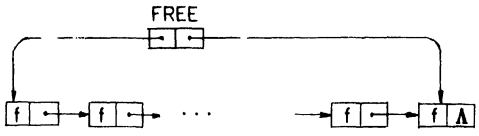


Fig. 1

F1. FREE.left = n_1, n_k.right = Λ

F2. for $1 \le i < k$, n_i.right = n_{i+1}.

F3. for $1 \le i \le k$, n_i.left = f.

F4. for $1 \le i \le k$, n_i.color = off-white.

F5. FREE.right is the pointer to the last node or the node before the last node of the free list. (The latter case occurs temporarily each time when a node has just been appended to the free list, but FREE.right yet remains to be updated.)

Based on the graph, we give the following definitions:

Definitions

Node n is said to be reachable from node m, if m=n or if there exists a path on the directed graph from m to n. (In this paper, a path always refers to a directed path with distinct nodes.)

A node is said to be free if it is reachable from FREE.

A node is said to be active if it is reachable from a root.

A node is said to be a garbage node if it is neither active nor free.

2.2 An Output-Restricted Deque

Both the GC and the LP use an output-restricted deque which is implemented outside the memory space containing the directed graph. The deque contains pointers to nodes in the directed graph. The GC inserts and removes pointers to nodes from one end of the deque, called the GC-end of the deque, and the LP only inserts pointers to nodes at the other end, called the LP-end of the deque.

Before removing a pointer from the deque, the GC tests the emptiness of the deque. If the deque is empty and the LP is inserting a pointer into the deque but has not concluded the insertion, then the test result for emptiness will be true. The pointer will be removed from the deque only if the GC finds that the deque is nonempty.

3. The Parallel Garbage Collection System

In this section we give a parallel garbage collection system, which consists of two concurrent processes sharing the data structure defined in Section 2. One process is executed by the LP and the other one by the GC.

We shall assume that the following initial conditions hold before any of the two processes starts:

I1. All the roots are black and have no sons.

I2. The color of FREE is off-white.

I3. Nodes on the free list are nodes R+2, R+3, ..., M.

I4. The free list satisfies properties F1, ..., F4 of Section 2.1 with FREE.right being the pointer to the last node of the free list.

Note that I3 and I4 imply that at the beginning of the computation, free nodes are FREE and nodes on the free list.

3.1 Garbage Collector's Algorithm

The GC executes repeatedly the following cycle, which is composed of three phases, the root insertion phase, the marking phase, and the collecting phase. In the algorithm, MARKING is a Boolean variable initialized to false. We say that a marking phase starts at the time when MARKING is set to true, and ends at the time when MARKING is set to false.

```
GC1. [Root insertion phase]
for i ← 1 until R-1
    do insert node i into the GC-end of the deque od;

GC2. [Marking phase]
MARKING ← true;
s ← NEW.left;
if s ≠ Λ then
  if s.left ≠ f and s nonblack then
     blacken s;
     insert s into the GC-end of the deque
  fi
fi;
while the deque is not empty
     do
       n ← the node at the GC-end of the deque;
       blacken n;
       remove n from the deque;
       s ← n.left;
       if s ≠ Λ and s is nonblack then
          blacken s;
          insert s into the GC-end of the deque
       fi;
       s ← n.right;
       if s ≠ Λ and s is nonblack then
          blacken s;
          insert s into the GC-end of the deque
       fi;
     od;
MARKING ← false;

GC3. [Collecting phase]
for i ← R+2 until M
     do
       if node i is white
          then APPEND(i)
       else
          if i.left ≠ f then color node i white fi
       fi
     od
```

The procedure APPEND(n) is defined as follows:

```
n.color ← off-white;
n.left ← f;
n.right ← Λ;
[FREE.right].right ← n;
FREE.right ← n
```

3.2 List Processor's Operations

The LP may perform operations only on active nodes and may perform the operation LP_C defined below.

The LP can perform many kinds of operations, such as traversing a certain list structure through its pointers, testing if the left or right pointer of a node is Λ, etc. But for the purpose of this paper it suffices to consider only those operations which change the data structure. The process executed by the LP is controlled by any program in which operations of the latter kind are defined as follows:

Operation LP_A: Add a left (or right) outgoing edge from an active node m to an active node n.

1. set the left (or right) pointer of m to n;
2. if MARKING and n is white or off-white then
   gray n;
   insert n into the LP-end of the deque
   fi

Operation LP_Λ: Delete a left (or right) outgoing edge from an active node m.

set the left (or right) pointer of m to Λ.

Operation LP_C: Make a free node active and pointed to by NEW.

1. CREATE;
2. n ← NEW.left;
   if MARKING then
   blacken n;
   insert n into the LP-end of the deque
   fi

The procedure CREATE is defined as follows:

```
while FREE.left = FREE.right
    do nothing od;
NEW.left ← FREE.left;
FREE.left ← [FREE.left].right;
[NEW.left].right ← Λ;
[NEW.left].left  ← Λ
```

To simplify the correctness proof, we assume that the right pointer field of NEW always contains Λ, and the left pointer field of NEW can be altered only through an operation LP_C or LP_Λ. (Note that this assumption is not a restriction for a real list processing system.)

3.3 Remarks on the Parallel System

1. The parallel system is evolved from the well known sequential garbage collection system which uses a stack for marking nodes. In Appendix I of this paper, we show how both the correctness and efficiency arguments were used to guide the derivation of the parallel system, starting from the sequential system. In particular, we will show why it is necessary to color nodes with four colors, and argue that the proposed system is essentially the only parallel system which satisfies all the five properties stated in Section 1 of this paper. It is instructive to note that the ordering of the operations appearing in the parallel system is in general crucial to the correctness and efficiency of the system. For example, if the ordering of step 1 and step 2 in LP_A, or the coloring and insertion operations in step 2 of LP_A, is interchanged, then examples can be found to show that the resulting system would be incorrect.

2. In the sequential system, the stack is accessed only by the GC. In the parallel system, a deque is used instead of the stack to avoid possible access conflicts of the stack, since both the GC and the LP may manipulate it at the same time.

3. Step 2 of operation LP_C is included only for efficiency reasons, which we explain as follows. It is usually the case that after performing an operation LP_C, the LP will perform an operation LP_A to make some node point to the newly created node. Thus, during the marking phase, it is better to blacken the newly created node (and insert it into the deque) once for all, so that there is little chance that the node will be colored and inserted into the deque by both the LP and the GC. For similar efficieny reasons, node s is blackened in GC2 before it is inserted into the deque; this blackening would not be necessary if only the correctness of the system were concerned.

4. The Correctness of the Parallel Garbage Collection System

4.1 Assumptions

A1. The LP and the GC can read and write on individual fields of a node, and the following operations are indivisible:

"Read or write a field" of a node by the LP or the GC.
"Gray or blacken a node" by the LP.
"Blacken, whiten or off-whiten a node" by the GC.

A2. The initial conditions I1, I2, I3 and I4 stated in Section 3 are satisfied at the beginning of the computation.

A3. ("The procedures operated on the free list are correct:") If the free list, i.e., the list pointed to by the left outgoing edge of FREE, is updated only by the procedures APPEND and CREATE, then the properties F1, F2, F3, F4 and F5 of Section 2.1 are preserved all the time. (We choose to assume A3 rather than to prove it, for it is similar to the traditional producer/consumer problem.)

A4. ("The deque will not overflow and operations on it are correct:") There is always some extra temporary space available for storing the deque and the GC does not find the deque empty until all the nodes which were inserted into the deque have been removed from it. (An upper bound on the number of elements the deque may have is derived in Section 5.)

4.2 Definition of Correctness

We say that the parallel garbage collection system is correct, if the following conditions are all satisfied:

C1. Only garbage nodes are appended to the free list by the GC.

C2. The GC never changes pointers of active nodes.

C3. A garbage node will always be appended to the free list within a certain time, which can be estimated a priori.

Conditions C1 and C2 guarantee that the GC does not interfere with the LP operations. Condition C3 ensures that the GC indeed collects garbage effectively.

By C1 and the fact that no active nodes are on the free list at the beginning of the computation, we see that the free list can be modified only by the procedures APPEND and CREATE. Hence by assumption A3, we know that the free list is manipulated "correctly", i.e., the free list always satisfies all the properties F1 to F5.

4.3 The Correctness Proof and Statement of the Main Theorem

In Sections 6 and 7, we shall prove the following theorem:

Main Theorem:

For the parallel system defined in Section 3, the following properties hold:

(i) **During a marking phase at each time when the GC checks the emptiness of the deque, a white active node is always reachable from some node in the deque.**

(ii) **A free node is always off-white.**

Hence during a marking phase, if the GC finds the deque to be empty, then by (i) there is no white active node. This implies that when a collecting phase starts all white nodes are not active and by (ii) they are garbage. Note that the LP never colors a node white and, during a collecting phase, the GC examines each node only once. Hence we have shown that the system satisfies condition C1.

Because the algorithm satisfies condition C1, it also satisfies condition C2, since the GC only changes pointers of free nodes or garbage nodes (through procedure APPEND).

It is not difficult to see that a garbage node can always be appended to the free list within time 2T, where T is an upper bound on the time taken by one garbage collection cycle. It is clear that the execution times of the root insertion phase and collecting phase can be estimated a priori. In the next section, we shall give an upper bound on the time taken by the marking phase of any garbage collection cycle. These imply that the system satisfies condition C3.

5. An Upper Bound on the Marking Phase Time

Consider the marking phase of any garbage collection cycle. Let

A = number of active nodes, besides the roots, at the beginning of the marking phase,

$1/k$ = time to insert a node into or remove a node from the deque,

r = rate that new active nodes are created (i.e., removed from the free list) in the sequential system, when the LP is running,

T_M = time taken by the marking phase.

With respect to a given computer, we assume that quantities A, k and r can be estimated from a given list processing program. In the following we derive an upper bound on T_M, under the reasonable assumption that $k > r$.

During the marking phase, the GC is busy inserting nodes into and deleting nodes from the deque, and also doing some minor operations (such as blackening a node or testing the color of a node). Assuming that these latter operations are incorporated in the insertion and deletion operations, we can write the following:

$$T_M = T_I + T_D$$

where

T_I = total time in the marking phase during which the GC is making insertions, and

T_D = total time in the marking phase during which the GC is making deletions.

Note that nodes inserted into the deque by the GC are among those nodes which are active at the beginning of the marking phase. We have the following inequality:

$$T_I \leq A / k.$$

Let D be the total number of nodes deleted from the deque during the marking phase. Then

$$T_D = D / k.$$

Note that nodes inserted into the deque must be either free or active at the beginning of the marking phase. Nodes in the first category, after being removed from the free list, are blackened and inserted into the deque once for all by the LP. The number of such nodes is $\leq rT_M$, since r certainly is an upper bound on the rate that new active nodes are created in the parallel system for which there are overheads for the LP. Consider now those nodes which are active at the beginning of the marking phase. Each of them, except the roots, can be inserted into the deque at most three times, since the black color of a node may be overwritten by the gray color at most once and the gray color by the black color at most once (cf. the counter-example given in Section 7.1). But a root can be inserted into the deque at most once. Since the number of nodes deleted from the deque is no greater than that inserted into the deque, we have therefore shown that

$$D \leq rT_M + 3A + R.$$

Note that the above inequality might be inexact to a small number of nodes owing to the fact that no synchronization is assumed in the manipulation of the Boolean variable MARKING. (For instance, there could be a large gap between the time when the LP finds MARKING to be true and the time when the LP inserts the corresponding node into the deque.) Here we ignore this possible unimportant discrepancy. Thus,

$$T_D \leq (rT_M + 3A + R) / k$$

and we have established the following theorem:

Theorem 1:

$$T_M \leq (4A + R) / (k - r).$$

The theorem gives upper bound on T_M which is proportional to A and independent of the size M of the memory space. This property turns out to be extremely crucial to the performance of a parallel garbage collection system, but is not satisfied by any previous system which does not use critical sections. Note that the time for the GC to execute a root insertion phase or a collecting phase is more or less a constant. Hence to minimize the time to execute each garbage collection cycle, it is necessary to minimize the time to execute each marking phase. This is an intuitive explanation on why we want to minimize T_M.

An obvious upper bound on D is M. Theorem 1 and the fact that $D \leq rT_M + 3A + R$ give another upper bound on D:

$$D \leq \frac{r}{k - r}(4A + R) + 3A + R$$

$$= \frac{3k + r}{k - r} A + O(1).$$

6. The Proof of the Main Theorem for an Auxiliary Parallel System

In this section we introduce an auxiliary parallel garbage collection system and prove a stronger version of the main theorem for this auxiliary system. In the next section, with a small effort we will be able to prove the main theorem for the parallel system in Section 3 by transforming the proofs and the results obtained for the auxiliary system in this section.

We now define the auxiliary system. The garbage collector's algorithm is defined as follows:

```
GC1. [Root insertion phase]
for i ← 1 until R-1
  do insert node i into the GC-end of the deque od;
insert NEW into the GC-end of the deque;

GC2. [Marking phase]
MARKING ← true;
s ← NEW.left;
if s ≠ Λ then
   if s.left ≠ f and s nonblack then
      insert s into the GC-end of the deque;
      blacken s
   fi
fi;
remove NEW from the deque;
while the deque is not empty
  do
    n ← the node at the GC-end of the deque;
    blacken n;
    s ← n.left;
    if s ≠ Λ and s is nonblack then
       insert s into the GC-end of the deque;
       blacken s
    fi;
    s ← n.right;
    if s ≠ Λ and s is nonblack then
       insert s into the GC-end of the deque;
       blacken s
    fi;
    remove n from the deque
  od;
MARKING ← false;

GC3. [Collecting phase]
for i ← R+2 until M
  do
     if node i is white
        then APPEND(i)
     else
        if i.left ≠ f then color node i white fi
     fi
  od
```

The list processor operations used in the auxiliary system are the same as those used in the original parallel system in Section 3, except that the "gray n" operation in LP_A is now replaced by "shade n". The operation "shade" makes a white or off-white node into gray and leaves a black or gray node unchanged, and is assumed to be indivisible. (As a matter of fact, in the auxiliary system it turns out that the operation "shade" will never have to be performed on a gray node.) Under the assumptions stated in Section 4.1, the following theorem can be proven:

Theorem 2:

For the auxiliary parellel system defined in Section 6, the following properties hold:

(i) **During a marking phase, a white active node is always reachable from some node in the deque.**

(ii) **The left pointer field of a free node (except FREE) always contains the value f, and a node whose left pointer field contains the value f is always off-white.**

We shall prove the theorem by induction on successive garbage collection cycles. Note that if Theorem 2 holds through the end of the marking phase of the ith garbage collection cycle then (ii) holds through the end of the marking phase of the (i+1)st cycle. This follows from the following argument at the end of the marking phase of the (i+1)st cycle: Since at the beginning the free list contains no active nodes and since only garbage nodes have been appended to the free list (cf. the proof of C1 in Section 4.3), free nodes have been accessed only by the procedures APPEND and CREATE and, consequently, by assumption A3 the properties in (ii) are preserved.

Since there are no white nodes before the collecting phase of the first garbage collection cycle starts, (i) holds automatically for the first cycle. This together with the fact that the free list contains no active nodes at the beginning of the computation imply that free nodes can only be accessed by the procedures APPEND and CREATE during the first cycle. Hence by assumption A3 (ii) also holds for the first cycle. **In the rest of Section 6 we assume that Theorem 2 holds for the ith cycle and want to prove that it holds for the (i+1)st cycle.** As noted in last paragraph, in the proof we may use the fact that (ii) holds through the end of the marking phase of the (i+1)st cycle.

6.1 Notation

In order to present our correctness proof for the auxiliary parallel system more easily, we introduce some "ghost operations" in the list processor operations LP_A and LP_C. The new definitions of LP_A and LP_C are given in the following, where ghost operations are indicated between square brackets in steps G1 and G2. Note that these ghost operations are not intended to be part of the real algorithm, but serve merely for proof purposes. We assume that step G1 or G2 is executed at the instant of completion of step 1 or 2 of LP_A, respectively, (or of step 2 or 6 of LP_C), and the execution takes no time.

Operation LP_A: Add a left (or right) outgoing edge from an active node m to an active node n.

```
1. set the left (or right) pointer of m to n;
G1. [mark the edge (m,n) created at step 1;]
2. if MARKING and n is white or off-white then
      shade n;
      insert n into the LP-end of the deque
   fi;
G2. [unmark the edge (m,n) marked at step G1]
```

Operation LP_C: Make a free node active and pointed to by NEW.

```
1. while FREE.left = FREE.right
     do nothing od;
2. NEW.left ← FREE.left;
G1. [mark the left outgoing edge from NEW;]
3. FREE.left ← [FREE.left].right;
4. [NEW.left].right ← Λ;
5. [NEW.left].left ← Λ;
6. n ← NEW.left;
   if MARKING then
      blacken n;
      insert n into the LP-end of the deque
   fi;
G2. [unmark the edge marked at step G1]
```

We say an edge is marked if it has been marked by the LP at step G1, but step G2 which unmarks the edge has not been executed by the LP. Hence an edge is marked if and only if i) the operation LP_A which created the edge has finished its step 1 but not step 2 or ii) the operation LP_C which created the edge has finished its step 2 but not step 6. Since there is only one list processor, at any time there is at most one marked edge. If an edge is marked at time t, it is called the marked edge at time t.

A path is called a marked path, if the marked edge is on the path. A path is called an unmarked path, if no edge on the path is marked.

We shall assume that we are at some time t during the marking phase of the (i+1)st garbage collection cycle, and that t_M is the starting time of the marking phase. As noted earlier, statement (ii) of Theorem 2 holds through the end of the marking phase of the (i+1)st cycle, and hence through time t.

6.2 Preliminary Lemmas

Lemma 1:

If at time t a black node m has a son, then m was in the deque at some time in $[t_M,t]$.

Proof:

If m is a root then the lemma is obvious, since roots are all in the deque at time t_M. Suppose that m is not a root. We first show that m was white or off-white at some time during the collecting phase of the previous cycle. During that phase the color of m was tested by the GC. The test outcome was either white or nonwhite. In the latter case, if m.left = f then m was off-white and on the other hand, if m.left ≠ f then m was colored white afterwards by the GC.

Since m is black at time t, it was blackened either by the GC at some time in $[t_M,t]$, or by the LP through an operation LP_C. If m was blackened by the GC, then at the time when m was blackened m was in the deque (the GC only blackens a node which is already in the deque). If m was blackened by an operation LP_C, then since m has a son at time t the operation LP_C must have been completed and consequently, m was inserted into the deque before time t. We conclude that m was in the deque at some time in $[t_M,t]$.

Lemma 2:

At time t, if edge (m,n) is unmarked with m black and n nonblack, then at least one of the two nodes m and n is in the deque.

Proof:

Note first that n must be nonblack throughout the interval $[t_M,t]$, since a black node remains black during the marking phase.

Let t_{LP} be the time instant when edge (m,n) was created, with $t_{LP} \leq t$. (t_{LP} is the instant of completion of step 1 of LP_A or step 2 of LP_C.)

a) $t_{LP} \leq t_M$: Edge (m,n) has been existing since time t_M. By Lemma 1, m was in the deque at some time in $[t_M,t]$. Suppose that m is not in the deque at time t. If m ≠ NEW, then before m was removed from the deque, n would have been blackened. This is a contradiction. If m = NEW, then before NEW was removed from the deque, the GC examined the left pointer field of n. If it did not contain f, then n would be blackened, which is a contradiction. If it contained f, then at the time when the GC was examining n.left, step 6 of the operation LP_C which created (m,n) had not started yet. Clearly from that time through time t, the Boolean variable MARKING was true. Since (m,n) is unmarked at time t, the operation LP_C has been completed by time t, and thus n would have been blackened by the LP. This again is a contradiction.

b) $t_M < t_{LP}$: Since $t_M < t_{LP} < t$, edge (m,n) was created during the interval(t_M,t). Suppose that the operation which created (m,n) was an operation LP_C. Since (m,n) is unmarked at time t, the operation LP_C has been completed by time t and thus n would have been blackened. This contradiction shows that the operation which created (m,n) must be an operation LP_A. Since (m,n) is unmarked at time t, the operation LP_A has been completed by time t. During the time interval (t_{LP},t), the LP tested the color of n. The outcome of the test must have been white, off-white or gray.

i) The test outcome was white or off-white. Then n was inserted into the deque by the LF at some time in $(t_{LP},t]$

ii) The test outcome was gray. Note that as shown in the proof of Lemma 1, node n was white or off-white at some time t' during the collecting phase of the previous garbage collection cycle. The operation LP_A which colored n gray must have inserted n into the deque at some time in [t',t). This implies that n was in the deque at some time in $[t_M,t]$, since no nodes were removed from the deque during $[t',t_M]$.

Both cases imply that n is in the deque at time t, because if n were removed from the deque, it would have been blackened by the GC.

6.3 Proof of Theorem 2

Suppose that w is a white active node at time t. We shall prove that w is reachable from some node in the deque. Since w is white and active, w is not a root and is reachable from some root through at least one path. There are two cases.

Case 1: w is reachable from some root through an unmarked path.

Let m be the first black node that is encountered on the path by traversing backwards from w to the root. Clearly such a black node exists, since the root is black. By Lemma 2, m or its son on the path is in the deque.

Case 2: The marked edge at time t is on every path from a root to w.

Let the marked edge be (m,n). Since nodes on a path are all distinct, we must have m ≠ n. Without loss of generality, we assume that (m,n) is the left outgoing edge from m to n. Consider any one of the paths from roots to w, and call it path P. Let b be the first black node that is encountered on path P by traversing backwards from w to the corresponding root. Suppose that b is a descendent or ancestor of m with respect to path P (see Fig. 2 and Fig. 3). Then the outgoing edge from b on path P is unmarked. By Lemma 2, we conclude that b or its son on path P is in the deque.

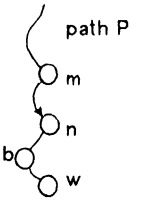


Fig. 2

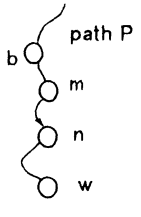


Fig. 3

In the following we assume that b = m. By Lemma 1, m was in the deque at some time in $[t_M,t]$. Suppose that m is not in the deque at time t. We shall show that w is reachable from some node in the deque through a path, called the "m^*-w path" below. Let t_{GC} be the time instant when the GC started reading the left pointer field of m before m was removed from the deque for the last time. Let t_{LP} be the time instant when edge (m,n) was created. We have the following two cases:

A) $t_{LP} \leq t_{GC}$. If (m,n) was created by an operation LP_A, then m ≠ NEW and consequently node n would have been blackened before m was removed from the deque. This is a contradiction. Suppose that (m,n) was created by an operation LP_C. Then m = NEW. Moreover, since (m,n) is marked at time t, the descendents of n are all off-white and, consequently, n = w. Before NEW was removed from the deque, the GC examined the left pointer field of w. If it did not contain f, then w would be blackened, which is a contradiction. If it contained f, then at that time w was off-white. This implies that w would not be white at time t, since the white color is set only during a collecting phase. Again we have a contradiction.

B) $t_{GC} < t_{LP}$. (See Fig. 4). Since $t_{GC} < t_{LP} \leq t$, edge (m, n) was created during the interval $(t_M,t]$. Suppose that the operation which created (m,n) is an operation LP_C. Then, as in A, m = NEW, n = w, and hence at time t_{LP} w was off-white. This is a contradiction since w is white at time t. Therefore the operation which created (m,n) is an operation LP_A. Choose t^* so that $t_{GC} < t^* < t_{LP}$ and that step 1 of the operation LP_A started before time t^*. (Recall that t_{LP} is the time instant of completion of step 1.) Clearly at time t^* there was no marked edge.

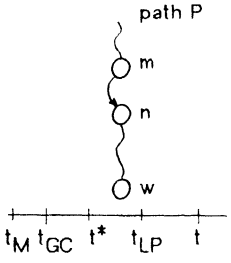


Fig. 4

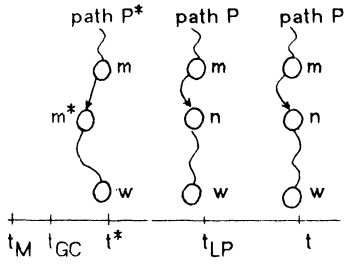


Fig. 5

Node w was active at time t^*, since it is active at time t and no new active nodes were created during the interval $[t^*,t]$. Let path P* be any one of the paths from roots to w at time t^*. (See Fig. 5.) Let m^* be the son of m on path P* at time t^*. Consider the path from m^* to w on path P* at time t^*, and call it the "m^*-w path". Note that the m^*-w path was not affected by the change of the left pointer of m at time t_{LP} and remains unchanged throughout the time interval $[t^*,t]$. Also note that the m^*-w path at time t is unmarked. Hence by Lemma 2 we have the following result:

If there is a black node on the m^*-w path at time t, then some node on the m^*-w path is in the deque at time t.

Thus if m^* is black at time t, then our proof is complete. In the following we assume that m^* is nonblack throughout the time interval $[t_M,t]$. Note that m was black at time t_{GC} and hence at time t^*. By Lemma 2, at least one of the two nodes m and m^* is in the deque at time t^*.

a) m^* was in the deque at time t^*. Then m^* is in the deque at time t, for otherwise m^* would have been blackened by the GC before it was removed from the deque.

b) m was in the deque but m^* was not at time t^*. Let t_{LP}^* be the time instant when edge (m,m^*) was created. Then from the proof of Lemma 2, it is easy to see that we must have the case $t_{LP}^* \leq t_M$. Hence the GC found m.left to be m^* at time t_{GC}. This implies that m^* was blackened by the GC at some time in (t_{GC},t) before m was removed from the deque. This is a contradiction.

We have shown that statement (i) of Theorem 2 holds for the (i+1)st garbage collection cycle. This together with the fact that statement (ii) holds through the end of the marking phase of the (i+1)st cycle imply that only garbage nodes have been appended to the free list through the end of the (i+1)st cycle. Therefore through the end of the (i+1)st cycle free nodes can be accessed only by the procedures APPEND and CREATE and, consequently, by assumption A3 statement (ii) holds. The proof of Theorem 2 by induction is complete.

7. The Proof of the Main Theorem for the Parallel System

In this section, the auxiliary system introduced in the preceding section is transformed to the parallel system proposed in Section 3. We will examine how the transformation will affect the proofs and results in the preceding section. The transformation is done in two steps.

7.1 Transformation 1

This transformation replaces the "shade n" operation in LP_A by the simpler indivisible operation "gray n".

After this transformation, Lemma 2 is no longer valid, as the following counter-example shows: Assume that (m,n) is an edge such that m is a black node at the GC-end of the deque and n is white. Consider the while loop of GC2 in which m is removed from the deque. Suppose that after the GC finds the color of n to be white and before the GC inserts n into the deque, the LP performs an operation LP_A to make some node (≠ m) point to n and also finds node n to be white, and then the LP pauses for a while. Now the GC inserts node n into the deque and blackens n. Suppose that after m and n are both removed from the deque by the GC, the LP resumes its previous operation LP_A and grays n. We have an unmarked edge (m,n) with m black and n gray, and neither of them is in the deque!

However, the results in the preceding section are still valid with respect to the following interpretation. We say that a node is once-black at time t during a marking phase if it is black at some time in the interval $[t_M,t]$, where t_M is the starting time of the marking phase. Then one can see that the lemmas, Theorem 2 and their proofs are still correct if we substitute all the occurences of the word "black" by "once-black" and "nonblack" by "non-once-black". (The substitution should be done only in the statements and proofs of the lemmas and Theorem 2. The substitution does not affect the parallel system and is used only for proof purposes.) Therefore, Theorem 2 holds for the auxiliary system after Transformation 1.

7.2 Transformation 2

The second transformation optimizes the garbage collector's algorithm. The root insertion and marking phases of the garbage collector's algorithm are redefined as follows (where operations inside square brackets are ghost operations used merely for proof purposes).

```
GC1. [Root insertion phase]
for i ← 1 until R-1
    do
    [insert node i into the GC-end of the ghost-deque;]
    insert node i into the GC-end of the deque
    od;
[insert NEW into the GC-end of the ghost-deque;]

GC2. [Marking phase]
MARKING ← true;
s ← NEW.left;
if s ≠ Λ then
   if s.left ≠ f and s nonblack then
```

```
        [insert s into the GC-end of the ghost-deque;]
        blacken s;
        insert s into the GC-end of the deque
      fi
    fi;
    [remove NEW from the ghost-deque;]
    while the deque is not empty
      do
        n ← the node at the GC-end of the deque;
        blacken n;
        remove n from the deque;
        s ← n.left;
        if s ≠ Λ and s is nonblack then
          [insert s into the GC-end of the ghost-deque;]
          blacken s;
          insert s into the GC-end of the deque
        fi;
        s ← n.right;
        if s ≠ Λ and s is nonblack then
          [insert s into the GC-end of the ghost-deque;]
          blacken s;
          insert s into the GC-end of the deque
        fi;
        [remove n from the ghost-deque]
      od;
    MARKING ← false;
```

We also redefine the operations LP_A and LP_C by adding the ghost operation

[insert n into the LP-end of the ghost-deque;]

before the insertion of node n into the deque. Observe that the updates of the ghost-deque in the transformed system occur in the same positions as those of the deque in the auxiliary system. Hence our proofs in Section 6 apply to the ghost-deque in the transformed system. Therefore Theorem 2 holds for the transformed system with deque replaced by ghost-deque. Observe also that at each time when the GC checks the emptiness of the deque, the deque and the ghost-deque contain the same set of elements. Hence statement (i) of Theorem 2 holds for the transformed system at each time when the GC checks the emptiness of the deque.

Since after the two transformations the auxiliary system becomes the parallel system defined in Section 3, we have shown that the Main Theorem holds for the parallel system.

8. Concluding Remarks

The idea of performing garbage collection in parallel with list operations has been around for some time. (Knuth[1968, exercise 2.3.5-12] credits this idea to M. Minsky.) Though it is an appealing idea for real time list processing applications, no papers on parallel systems were published until two years ago. Steele[1975] is probably the first one who investigated such a system. Because of the necessity of performing semaphore-type operations so frequently, his system is not efficient on standard, general purpose computers. Both our system and the one proposed by Dijkstra, et al., called system D below, do not use any semaphore-type operations. However, there are some essential differences between system D and our system:

(i) During the marking phase in system D, free nodes are marked by the GC; this is not required in our system.

(ii) During the marking phase, system D may step through the whole memory, i.e. M nodes, as many as N times, where N is the number of nodes to be marked. Our system uses a deque and the system is so designed that the marking phase has the execution time proportional to the number of active nodes and independent of the size of the memory. On the other hand, system D has smaller list processor overheads (for example, its only overhead in the operation LP_A is the "shading" of the target node). Also, system D does not require any extra space.

(iii) System D assumes an indivisible "shade" operation. No special indivisible operations are assumed in our system.

We understand that Dijkstra [1976], Gries [1976] and Lamport[1976] (the latter two papers also consider system D or similar ones) deal mainly with the correctness issue and regard efficiency as a separate issue. But it is precisely for efficiency reason that we wanted to consider parallel garbage collection systems in the first place. The point of this research was to handle these two important issues at the same time.

For efficiency reasons, we propose using a deque for the marking phase. The inclusion of the deque in our system has significantly increased the complexity of proving the correctness of the system. (For example, step 2 of operation LP_A would be an indivisible action in system D.) In spite of this, we believe that we have given a correctness proof of our system which is still relatively short and readable. We achieve this mainly by making the "right" assertions for the system through the use of so called "ghost variables". Our "stepwise refinement" proof technique is also crucial. We first introduce an auxiliary system and prove its correctness. The proof and results are then transformed step by step as the auxiliary system is transformed to the parallel system for which we want to prove the correctness. If one does not use this stepwise refinement technique and attempts to prove directly the correctness of the final system, one would almost surely end up with an unreadable and complicated proof. (It is unlikely that one would come up with, say, assertions involving concepts such as "once-black" directly from the final system.) Note that we are not proposing a methodology for the correctness proof of such parallel systems. Our main concern was to make the proof clear and convincing.

The really restrictive assumption is A4 in which we assumed that there is always some extra temporary space available for storing the deque. Although there are methods which use reversed pointers in the nodes themselves as a stack, these methods are not suitable for the parallel system since pointer fields are modified by the garbage collector.

There are a number of possible extensions which can be made based on the system described in this paper. Our intention here was to describe the basic ideas of the system rather than explore several variations.

References

Dijkstra[1976] Dijkstra, E. W. et. al. "On-the-fly Garbage Collection: An Exercise in Cooperation", in Language Hierarchies and Interfaces, edited by F. L. Bauer and K. Samalson, Springer-Verlag, New York, 43-56.

Gries[1976] Gries, D. "An Exercise in Proving Parallel Programs Correct", in Language Hierarchies and Interfaces, edited by F. L. Bauer and K. Samalson, Springer-Verlag, New York, 57-81.

Gries[1977] Gries, D. "On Believing Programs To Be Correct", Comm. ACM 20, 49-50.

Kung and Song[1977] Kung, H. T. and Song, S. W., "Performance Analysis of a Parallel Garbage Collection System", Department of Computer Science Report, Carnegie-Mellon University, August 1977.

Knuth[1968] Knuth, D. E. The Art of Computer Programming,

vol.1: Fundamental Algorithms. Addison-Wesley, Reading, Mass..

Lamport[1976] Lamport, L. "Garbage Collection with Multiple Processes: an Exercise in Parallelism", in Proc. 1976 International Conference on Parallel Processing, edited by P. H. Enslow Jr., IEEE Computer Society, Long Beach, California, 50-54.

Steele[1975] Steele, G. L. Jr. "Multiprocessing Compactifying Garbage Collection", Comm. ACM 18, 495-508.

Wadler[1976] Wadler, P. L. "Analysis of an Algorithm for Real Time Garbage Collection", Comm. ACM 19, 491-500.

Appendix I

The Derivation of the Parallel Garbage Collection System

The parallel system proposed in this paper is evolved from the well known sequential garbage collection system which uses a stack for marking nodes, assuming that extra space is available for storing the stack. In this Appendix I we show informally how both the correctness and efficiency arguments were used to guide the derivation of the parallel system, starting from the sequential system. For briefness in the examples below, we write "insert n" and "remove n" for "insert n into the stack (or deque)" and "remove n from the stack (or deque)", respectively.

In the following we first present the sequential system and then transform it into the parallel system in four major steps, A, B, C and D. The sequential garbage collection algorithm blackens all active nodes by using a stack, and then appends all white nodes to the free list and turns all black nodes into white ones. It is assumed that all the R roots are initially black and have no sons, and that all other nodes are initially white and on the free list (see Fig. 6). Node R is called NEW and node R+1 is called FREE.

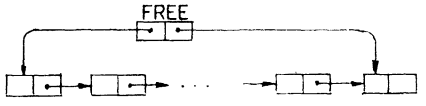


Fig. 6

The garbage collection algorithm is given as follows:

```
GC1. [Root insertion phase]
for i ← 1 until R
  do push node i onto the stack od;

GC2. [Marking phase]
while the stack is not empty
  do
    n ← the node at the top of the stack;
    remove n from the stack;
    s ← n.left;
    if s ≠ Λ and s is white then
      blacken s;
      push s onto the stack
    fi;
    s ← n.right;
    if s ≠ Λ and s is white then
      blacken s;
      push s onto the stack
    fi
  od;

GC3. [Collecting phase]
for i ← R + 2 until M
do
  if node i is white then
    APPEND(i)
  else
    color node i white
  fi
od
```

The procedure APPEND(n) is defined as:

```
[FREE.right].right ← n;
FREE.right ← n
```

The list processor's operations which change pointer fields of nodes are given as follows:

Operation LP_A: Add a left (or right) outgoing edge from an active node m to an active node n.

```
set the left (or right) pointer of m to n.
```

Operation LP_Λ: Delete a left (or right) outgoing edge from an active node m.

```
set the left (or right) pointer of m to Λ.
```

Operation LP_C: Make a free node active and pointed to by NEW.

```
if FREE.left = FREE.right then
  perform garbage collection fi;
NEW.left ← FREE.left;
FREE.left ← [FREE.left].right
```

A. The sequential system must be modified.

Suppose that the sequential system is not modified. Consider Example 1.

	GC	LP
1.	remove m	
2.	read m.left	
3.	read m.right	
4.		create(m,n)
5.		delete all edges to n but (m,n)

Example 1

Assume that the following initial conditions hold before step 1: m is a black active node on the stack and has no sons; n is a white active node not on the stack. Then after step 5 is executed, n is a white active node and will not be blackened by the GC before the next collecting phase starts since m, the only immediate predecessor of n, is black and not on the stack. The system is therefore incorrect.

Example 1 suggests that n should be pushed onto the stack by the LP before step 5. To avoid access conflicts, we use a deque instead of a stack, so that the GC can access one end of the deque and the LP the other end. An example similar to Example 1 was first given by Dijkstra[1976] for illustrating that "no overhead for the LP" is unattainable in a parallel system.

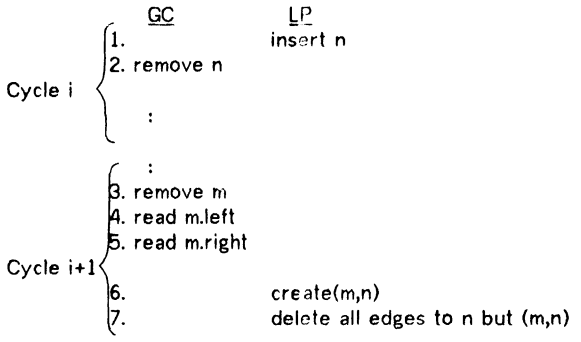
B. The operation LP_A must be modified.

Suppose that before an edge is deleted the LP always inserts the target node into the deque if it is white. Then the correctness problem illustrated by Example 1 seems to be solved. But with this solution, every garbage node will be blackened and inserted into the deque. This is clearly undesirable from the efficiency point of view. A better solution is that when an edge is created the LP always inserts the target node into the deque if it is white. That is, the operation LP_A

should be modified. In the following, we assume that the edge created during the operation LP_A is (m,n) and n is white.

B.1 The LP should insert n into the deque after (m,n) is created.

Suppose that n is inserted into the deque before (m,n) is created. Consider Example 2

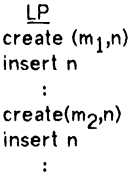


Example 2

We see that the insertion at step 1 is cancelled by the removal at step 2. Consequently, the same correctness problem illustrated in Example 1 occurs in cycle i+1. This shows that the system is incorrect.

B.2 The LP should color n.

Suppose that the LP does not color (or mark) n. Consider Example 3.

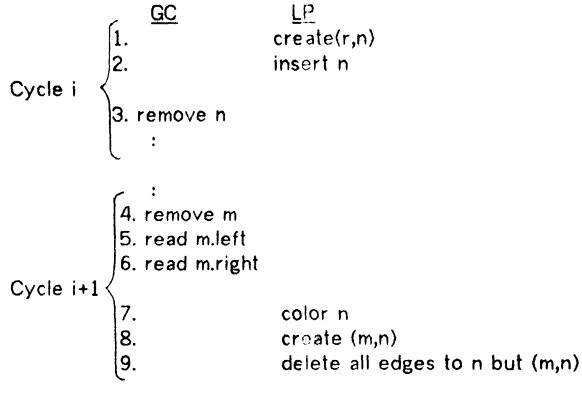


Example 3

This example shows that the same node n can be inserted into the deque arbitrary number of times. Consequently, the deque is unbounded and hence the marking phase time is also unbounded. The solution to this problem is to color n when it is inserted into the deque so that the LP will know that n need not be inserted again by testing the color of n.

B.3 The LP should color n before n is inserted into the deque.

Suppose that the LP colors n after n is inserted into the deque. Consider Example 4.



Example 4

Notice that after step 8 the LP does not insert n into the deque since n has already been colored by the LP at step 7 (cf. Section B.2). By the argument used in Example 1, any white active node which is reachable from a root only via n at that time will not be blackened by the GC during cycle i+1. This shows that the system is incorrect.

B.4 The LP should color n gray (or any color different from black).

Suppose that the LP colors n black. Consider the case in which at the very beginning of a marking phase, n is not in the deque, and the LP blackens n and then stops for the rest of the phase. Then those white active nodes which are reachable from roots only via node n will not be blackened by the GC during the marking phase since n is black at the beginning of the phase. The system is therefore incorrect.

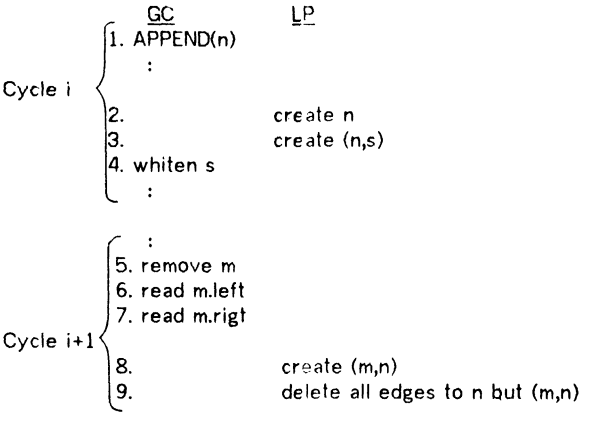
B.5 The LP should color n gray and insert n into the deque only during a marking phase.

Note that the collecting phase time is proportional to M. Hence if the LP inserts nodes during a collecting phase then the number of nodes inserted to the deque would be proportional to M and, consequently, so would the marking phase time. This violates our requirement that the marking phase time be O(A). It can also be observed that as a matter of fact there is no need for the LP to gray n or insert n during the root insertion and collecting phases for solving the correctness problems illustrated by all the previous examples.

C. Free list and operations on it should be modified.

C.1 The color of a free node should be off-white (or any color different from white, gray or black).

A newly created node must be nonwhite, for otherwise it can be regarded as a garbage node and appended to the free list incorrectly by the GC. Let n be such a newly created node. Suppose that n is gray. Note first that during a collecting phase nodes should not be inserted into the deque (cf. Section B.5). Consider Example 5, which uses the same principle as Example 4.



Example 5

After step 8 n is not inserted into the deque by the LP for n has been gray since step 2. We see that the white active node s will not be blackened in cycle i+1. This shows that the system is incorrect. Similarly, one can see that n must be nonblack.

It turns out that the best way to ensure that a newly created node is off-white is by letting all the free nodes be always off-white. This can be achieved by assuming that the procedure APPEND(n) will off-whiten n and the procedure CREATE will not change the color of nodes being created. In the operation LP_A an off-white node is treated in the same way as a white node.

C.2 Nodes on the free list should be readily identifiable.

During a collecting phase the GC should turn an off-white node into a white node. If the GC does not do so, then a released off-white node which does not happen to have an ancestor in the deque will never be appended to the free list. But the GC does not want to whiten nodes on the free list, which are supposed to be always off-white (cf. Section C.1). Hence, it is necessary to put a value "f" on the left pointer field of nodes on the free list so that they are identifiable to the GC. (Note that the left pointer field of nodes on the free list are originally not used anyway.) The procedures APPEND and CREATE should, of course, be modified accordingly for inserting and deleting f.

D. The garbage collector's algorithm must be modified.

Modifications to the collecting phase algorithm are essentially discussed in Section C.2. Appropriate modifications should also be done to the marking phase algorithm. Since the left son of NEW can temporarily lead to the free list during the middle of the execution of CREATE, caution must be taken to avoid the possibility that nodes on the free list are blackened by the GC. Hence the left son of NEW should be treated as a special case during the marking phase. Furthermore, minor changes on the marking phase algorithm can be made to improve the performance of the whole parallel system. This has been addressed in remark 3 of Section 3.3.

We see that after all the modifications indicated above have been made the sequential system has been transformed into the parallel system proposed in this paper.

Appendix II

The Performance of the Parallel System

In Appendix II, we briefly summarize part of the results in Kung and Song[1977], where the performance of the parallel system proposed in the present paper is analyzed. The analytic model we use takes account of the overheads introduced to the LP and the GC for the parallel system. We believe that the results derived from the model can be used successfully to predict the actual performance of the system.

Notation

M = Memory size.

A = Number of active nodes.

α = M / A.

r = rate nodes are released (that is, become garbage) in the sequential system, when the LP is running

u = rate nodes are used (that is, removed from the free list) in the sequential system, when the LP is running.

a = rate operations LP_A which may have a white target node are performed.

β = a / u.

m = rate nodes are marked by the GC during the marking phase in the sequential system.

s = rate nodes are scanned during the collecting phase, if no nodes are appended to the free list.

r' = rate nodes are released during the marking phase of the parallel system.

u' = rate nodes are used during the marking phase of the parallel system

m' = rate nodes are marked by the GC during the marking phase of the parallel system.

Assumptions

1. The system (sequential or parallel) is in equilibrium, that is, the number of active nodes A is constant.

2. β and the release and use rates are constant. Thus from assumption 1 we must have r=u and r'=u'. We will use r or r' for both use and release rates.

3. Quantities A, m/r, s/m and β are known.

4. When the LP makes a node to point to some other node, it is equally likely that any active node can be the target node.

5. Active nodes are released by the LP and become garbage at random. Thus the probability that an active node is released is independent of its color.

6. The GC marks all nodes which are active at the beginning of a marking phase. Clearly with this assumption we overestimate the marking phase time, the number of "floating nodes" (see Wadler[1976]), etc. in our analysis. Our results thus are likely to give lower bounds on the actual performance of the system.

7. We ignore the cost of minor operations such as testing or coloring a node; but we do include in the analysis the cost of any deque or free list access.

8. Two physical processors are always available for the parallel system, one for the LP and the other for the GC.

Results

We first consider the sequential system. Suppose that during the entire computation of a list processing program, a total amount of N nodes are to be removed from the free list. Before the LP starts M - A = A (α-1) nodes are available. The LP runs until it uses up all the available nodes and then the garbage collector takes over. Let the length of the time period when the LP runs be T_L and let the length of the garbage collection time be T_G. If T_N is the total time taken by the computation, then

$$T_N = (N/(rT_L)) (T_L+T_G)$$

Therefore the average time to use a node in the sequential system is

$$(T_N/N)_{seq} = 1/r (1 + T_G/T_L).$$

Clearly, we are interested in minimizing T_N/N. It can be minimized by increasing α, that is, by increasing M and thus T_L. Clearly there is a trade-off between time and space.

We now consider the parallel system. Our goal is to derive formulas for computing the following quantities:

Q1. The minimum α such that the LP will not run out of space if M is chosen to be αA.

Q2. The average time to use a node in the parallel system, $(T_N/N)_{par}$.

The following useful result is first proved:

$$m'/r' = m/r,$$

by which we are able to compute m' and r' in terms of m, r and β. (Note that for the parallel system in general r' < r due to the overheads introduced to the LP and m' < m because a node may be inserted into the deque more than once.) The following result is obtained:

Theorem A:

If $\alpha = M/A \geq (2m/r + 3 - r/m) / (2m/r - 1 - 2m/s)$

then the LP never runs out of space.

Values of α for which the LP will not run out of space when M is chosen to be αA are plotted in Fig. 7.

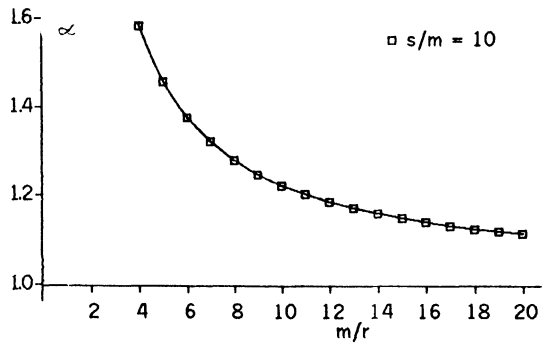


fig. 7. The minimum α such that the LP will not run out of space if M is chosen to be αA.

After M is chosen so that the LP will not run out of space, we are interested in knowing whether or not the parallel system actually executes the program faster than the sequential system, and if so, how much faster. (To answer these questions is trivial, since overheads have been introduced to the LP for the parallel system.) For this purpose, we have to know the average time to use a node. It is obvious that in any sequential or parallel system, the average time to use a node is $\geq (T_N/N)_{opt} = 1/r$. A closed formula for computing $(T_N/N)_{par}$ in terms of m/r, s/m, β and α has been obtained. In Fig. 8, we compare the values of T_N/N in the sequential and parallel systems to the optimal value of 1/r, assuming that r=1 and that the value of α is given by Fig. 7.

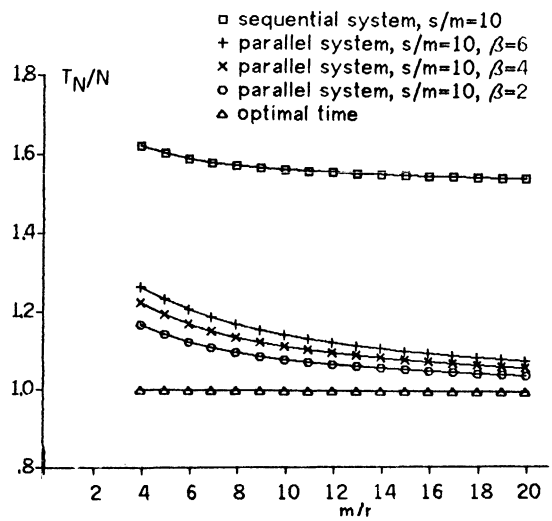


Fig. 8. The average time to use a node, assuming that r=1 and that the value of α is given by fig. 7.

A useful measure for studying the performance of a parallel system is its speed-up, which is defined as

$$\text{Speed-up} = (T_N/N)_{seq} / (T_N/N)_{par}.$$

Clearly, speed-up is always ≤ 2, since two processes are used. Another more useful upper bound can be derived as follows. Since $(T_N/N)_{par} \geq 1/r$,

$$\text{Speed-up} \leq 1 + T_G/T_L$$

for any parallel system. Corresponding to Fig. 8, the value of the speed-up of the parallel system is compared to the optimal speed-up $1 + T_G/T_L$ in Fig. 9. We see that the parallel system performs nearly optimally for large m/r.

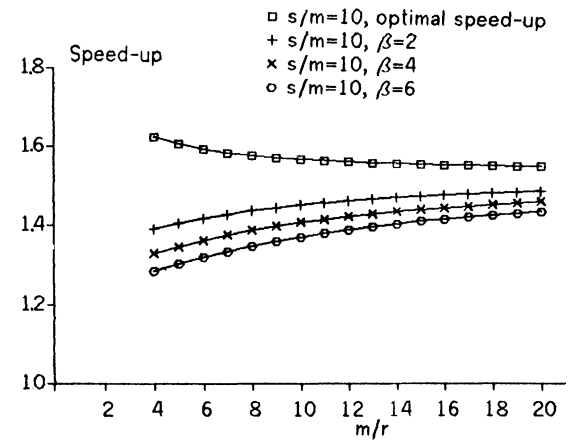


Fig. 9. Comparing the speed-up of the parallel system with the optimal speed-up, assuming that r=1 and that the value of α is given by fig. 7.

The analytic results sketched above are mainly obtained by solving differential equations describing the various transitions in the parallel system. It is assumed implicitly that all the solutions are differentiable. It is also assumed that the system is in the perfect equilibrium state, i.e., A, β, r and u are all constant in time. These restrictions might lead one to question the validity of the model. Because of this consideration, we have tested the model by simulation in which the time intervals between consecutive removals from the free list and consecutive releasing of active nodes are exponentially distributed independent random variables with the same mean. The simulation results are found to be nearly the same as the results obtained from the analytic model.