A Fine-Grained Parallel Memory Compaction Algorithm

PATRICK WEEMEEUW* AND BART DEMOEN†

Department Of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200 A, B-3001 Leuven, Belgium

Memory compaction is a technique for reclaiming cells containing garbage that are scattered over the memory space. More specifically, the memory cells are rearranged, so that all usable cells appear in one compact mass at one end of the area, and the remaining space at the other end can be recycled by the program. During this process, references are updated to point to the new locations. This process happens in place; no extra memory is needed. In addition to this, a sliding compaction algorithm preserves the relative order of the useful memory calls. Morris's algorithm is such a well known sequential algorithm. In this paper, we present a program scheme that is a generalization of Morris's algorithm, and that allows for parallel execution. Our scheme can be fine tuned for a particular implementation, taking into account the characteristics of the memory area and the hardware. The algorithm is space efficient, and runs in a time linear to the size of the memory area (the actual parameters are determined by the properties of the data to be compacted). As the algorithm is very fine grained, it has a nearly linear speedup, except in some uncommon cases. The memory area need not be contiguous, but must be globally accessible. © 1994 Academic Press, Inc.

1. INTRODUCTION

High level programming languages often take the burden of memory management away from the programmer. Data structures are dynamically created when necessary, and automatically reclaimed when no longer needed. The general term "Garbage Collection" (GC) refers to several kinds of techniques for recycling the memory space occupied by data structures that have become inaccessible. Conceptually, the Garbage Collection process then consists of two subtasks: identifying all reachable objects, and reclaiming the unreachable ones.

We distinguish between three large classes of garbage collectors.¹

Reference Counting Schemes. Each dynamically created object contains a count field with the number of references point towards it. This count is maintained by all operations manipulating these references. When the count drops to 0, the object can be deallocated by including it in a free list. While this approach is easy to imple-

ment (and to parallelize), it causes a rather large time and space overhead, and general cyclic structures cannot be reclaimed. Also, the storage becomes fragmented, with a poor locality of references as a result. The technique is attractive for interactive programs, since the GC time is spread over the normal processing time. Several optimizations have been proposed [12, 4, 11].

Copying GC Techniques. In this approach [2, 8], the available memory is divided into two areas, called "semi-spaces." New objects are all allocated in one area. On GC, all reachable cells (starting from a known set of "roots") in that area are moved to the other space in a compacted form, leaving behind a temporary reference to their new location. Each moved cell is checked for references toward other cells yet to be moved, thus covering all the reachable objects. After GC, the roles of the semi-spaces are swapped.

This method is fast (it has a run time proportional to the number of useful cells) and reclaims all garbage. The main drawback is that twice as much memory is needed.

Compacting GC Techniques. In this approach, the GC is done *in situ*: the cells are rearranged so that all useful objects appear at one end of the area, and space occupied by the unused cells can be reused at the other side. Garbage collectors of this kind have two clearly separated phases: a marking phase which identifies (marks) all reachable cells, and a compaction phase. If the relative order of the cells is preserved, the algorithm is of the sliding type.

Such techniques are in general less efficient than the copying GC techniques, as they have a run time proportional to the total number of cells instead of to the number of marked cells. However, the sliding techniques are important for applications which may deallocate large chunks of memory in a stackwise manner, since reclaiming memory by deallocation is done at practically no cost. An example of such an application is WAM based Prolog [20, 1, 18], which discards large parts of the stacks on backtracking. We investigate one particular algorithm of this class in detail.

This paper presents a parallel sliding compaction algorithm, based on the algorithm of Morris [17]. This is an efficient sequential algorithm, requiring two passes over

* E-mail: patrick@cs.kuleuven.ac.be.

† E-mail: bimbart@cs.kuleuven.ac.be.

¹ For a general survey of GC techniques, see [7].

the memory space. In this paper, we present a generalization of this algorithm, allowing for fine grained parallel execution, while still running in a linear time. In contrast to [22, 23], which give a formal presentation and derivation (with correctness proof) of the algorithm, the approach taken in this paper is an informal one. This allows one to comprehend quickly the essential elements of the algorithm, without the burden of having to indulge into the formal machinery.

The application we have in mind is garbage collection for the Aurora system [16]. Aurora is an OR-parallel Prolog system for shared memory multiprocessors, based on the SRI model [21], which is an OR-parallel extension of the WAM. Each processor administers its own data space, but parts of it are shared by other processors. Global garbage collectors, such as described by Crammond [9], which allow each processor to collect its own memory space in parallel, taking into account pointers from one area to another, are not appropriate here. Since the shared data structures may be very unevenly spread across the different memory areas, such an algorithm would cause an unbalanced load. Furthermore, since the data areas are operated upon as stacks, we may wish to restrict the garbage collection to the memory space that actually runs out of space, keeping the possibility of reclaiming the other memory areas cheaply by backtracking. Because the shared data are temporarily inconsistent during this process, the other processors have to stop normal execution anyway, and they can better get involved in the garbage collection process.

2. THE ALGORITHM OF MORRIS

Let us first restate the problem a bit more specifically. The number and positions of the marked cells are known from the preceding marking phase. Then, given an area of storage $M[\text{bottom} \dots \text{top}]$ containing N scattered marked cells, rearrange them into a compact mass at one end of the area $M[\text{bottom} \dots \text{bottom} + N - 1]$ (see Fig. 1). External references (i.e., pointers originating from a location outside the actual area pointing into the area) as well as

internal references (i.e., pointers from one cell to another cell in the storage area) have to be updated. The order of the cells has to be preserved.

Regarding the references, we define some useful notions. The *relocation set* of a cell i is the set of cells containing (originally) a reference toward cell i . During compaction, it is necessary to identify all cells in the relocation set of a cell, since these cells have to be updated with a reference to the new location of the cell. Therefore, the relocation set is represented by a data structure, called a *relocation collection*. The relocation collection of a cell is said to be *complete* when it contains all cells in the relocation set. The *head* of the relocation collection of cell i is cell i itself. The *members* of the relocation collection of cell i are the cells in the relocation set of cell i currently included in the relocation collection. Hence, the members of a relocation collection pointed originally toward the head.

There are two basic operations on relocation collections: inserting a cell into the relocation collection of the cell it points to, and updating the members of a complete relocation collection with a reference to the new location of the head. After the update of a relocation collection, the head contains its original contents again, and still has to be moved to its new location.

In Fig. 2, we illustrate how a relocation collection is represented in the algorithm of Morris at a cost of only one bit per cell (besides the mark bit). Suppose the relocation set of cell a is $\{b, c, d\}$ (Fig. 2, left). The head of the relocation collection then contains a reference to the first member of the relocation collection; each member except the last one contains a reference to the next member (Fig. 2, middle). The last cell contains the original contents of the head cell, and is therefore called the *value cell* of the relocation collection. All cells, except the last one, are marked with the F-bit (the *forward* bit), such that the end of the chain can be detected. Figure 2, right, shows how each member of the relocation collection is updated with a reference to the new location of the head and that the original value of the head is restored by the update operation. It is easily seen that relatively simple

FIG. 1. Effect of a sliding composition algorithm.

FIG. 2. Relocation collections.

pointer operations suffice to insert a cell in a relocation collection, or to update the members of a relocation collection, once the new location of the head is known.

This simple data representation does not allow relocation collections to be nested one into another—a cell member of a relocation collection cannot be the head of another relocation collection at the same time. To avoid this complication, Morris's algorithm guarantees that the relocation collection of a cell has been updated before the cell itself is possibly included in the relocation collection of another cell. However, a relocation collection must not be updated before it is complete. Therefore, Morris' algorithm consists of two phases: an upward and a downward one (see Fig. 2). The upward phase visits each cell from *top* to *bottom* and updates all upward directed pointers. The downward phase updates downward directed references and moves the cells to their final position, while scanning the area in the opposite direction. The relocation set is partitioned accordingly into two subsets: an upward and a downward one. The upward (downward) relocation set of a cell consists of all cells containing originally an upward (downward) directed reference towards the cell. Cells containing a reference toward themselves are considered later.

When cell *i* is handled during the upward phase, all cells between position *top* and *i* + 1 containing an upward reference have been inserted in the appropriate relocation collections. Therefore, the relocation collection of cell *i* is complete with respect to the upward direction, since all other references toward it must be downward ones. The new location of cell *i*, *i'*, is known at this moment: it is the *bottom* + *N* minus the number of marked cells visited already during the upward phase. Hence, the relocation collection of cell *i* can be updated. After its update, cell *i* contains its original contents and

can be included in the relocation collection of another cell if necessary.

A similar reasoning holds for the downward phase. Additionally, the contents of the cells are moved to their final position, and the mark bits are reset. Cells referring to themselves are copied with as well during this copying operation.

We can now present the algorithm of Morris in some more detail. We use some literate programming notations introduced by Knuth [14, 3] to clarify the most essential aspects of the algorithm in a top down manner. It is assumed that all external references have already been included in the relocation collections: sometimes this is done while marking, otherwise an extra *sweep* phase may be necessary.

```

⟨Morris⟩ ≡
count := N; { number of marked cells yet to visit }
for i := top to bottom { upward phase }
  if ⟨cell i is marked⟩ then
    begin
      count := count - 1;
      ⟨update the relocation collection of cell i, if any⟩
      { cell i contains now its original contents }
      if ⟨cell i contains an upward reference⟩ then
        ⟨insert i in the relocation collection of the cell
        it points to⟩
    end
  { count is now the number of marked cells already
  visited during the downward phase }
for i := bottom to top { downward phase }
  if ⟨cell i is marked⟩ then
    begin
      ⟨update the relocation collection of cell i, if
      any⟩
    end
  end

```

```

if <cell i contains a reference to itself> then
  <put a reference to i' in cell i>
else
  <copy contents of cell i to cell i'>
  <reset mark bit of cell i'>
  if <cell i' contains a downward reference> then
    <insert i' in the relocation collection of the
    cell it points to>
    count := count + 1;
  end

```

A cell c has a contents bit ($c.cts$) and two annotation bits ($c.F$ and $c.M$). i and j are references to cells. We get the subfields of the cell pointed to, with the $\rightarrow$ operator.

<insert i in the relocation collection of the cell it points to> =

```

begin
  j := i → cts; { i points to cell j }
  i → cts := j → cts;
  i → F := j → F;
  j → F := true;
  j → cts := i;
end

```

<update the relocation collection of cell i, if any> =

```

while (i → F)
  begin
    j := i → cts;
    i → cts := j → cts;
    i → F := j → F;
    j → F := false;
    j → cts := bottom + count;
  end

```

3. NESTING THE RELOCATION COLLECTIONS

The “flatness” of the relocation collections in the algorithm of Morris inhibits parallel execution. Indeed, for each cell, the relocation collection has to be updated before the cell itself can be inserted, and the relocation collection cannot be updated before the previous cells all have been inserted (otherwise, its relocation collection may be incomplete.) This imposes a total ordering between the operations; hence no parallelism is possible. We introduce nested relocation collections to remove this sequential dependency.

The essence of the algorithm relies on the observation that every tree can be represented by an annotated linear list [13]. When we take references between memory cells as child-parent pointers, the references form a tree (Fig. 3, left)—we discuss circular reference chains and cells containing a reference to themselves later. Figure 3, right, represents the same information. There, all cells of the tree are linked together in a linear list, starting from the top of the tree. The last cell contains the original

FIG. 3. Equivalent representations.

contents of the topmost cell. If a cell has any children, it contains a reference to one of its children; otherwise, if it has a “nearest sibling,” it contains a reference to it; otherwise, the cell is the last one in the chain. The nearest sibling is an unvisited sibling if it exists; otherwise it is an unvisited sibling of the nearest ancestor having one.

This linear list is disambiguated with appropriate annotations: C means that the cell has a child node, and S means that the cell has a sibling node somewhere further in the chain. We call the *pointer chain* of cell i the transitive closure of all cells belonging to the relocation collection of cell i , i.e., all cells originally pointing directly or indirectly to i . This is the set of cells in the linear list starting at cell i and ending with the value cell of i . The value cell then either contains the original contents of i , or a pointer to the nearest sibling of i , or is the value cell of a surrounding pointer chain. All this is schematically represented in Fig. 4. A dashed triangle encloses all cells belonging to the pointer chain of the cell at the top.

Our algorithm will transform references incrementally into and back from the alternative representation in Fig. 3. While transforming back, the references are updated to point to the new location of their parent cell.

Figure 5 illustrates how one pointer chain is inserted in another pointer chain. Cell i originally contained a reference to cell j . Now either cell i has its own pointer chain (and its original contents is stored in the value cell of its pointer chain), or it has not, and then the value cell of cell

FIG. 4. Nested pointer chains.

FIG. 5. Insertion.

i coincides with cell i itself. The contents of cell j , which are either the original contents of cell j , or a reference to the next sibling cell of cell i , are copied to the value cell of pointer chain i . Cell j has to point now to cell i (its first child) and must have the C annotation (if there is already an S annotation, this is preserved.) Finally, cell i must have the S annotation if cell j had already a C annotation.

Figure 6 shows the effect of the update operation. For each cell member of the relocation collection of cell i , its S annotation is removed and the value cell of its pointer chain is updated with a reference towards cell i' (the new location of cell i). Cell i itself now contains its original contents, or points to the next cell in the pointer chain in which it is enclosed. The C annotation is removed from cell i . This way, the pointer chain of each cell previously a member of the relocation collection of cell i is now effectively detached from the enclosing pointer chain.

In the next section, we specify the partial ordering between the basic operations (insert, update, and move) that guarantees a correct result. Any pair of operations among which no dependency holds can be executed concurrently.

4. A PARALLEL PROGRAM SCHEME

In [22, 23] we have proved the correctness of our algorithm, using the UNITY formalism². To keep a close relationship with this proof, we present the algorithm here in

² For a complete description of UNITY, see Chandi and Misra [6].

FIG. 6. Update.

a UNITY-like notation. However, since the main focus of this paper is on the presentation of the algorithm rather than the correctness proof, excessively intrinsic details will be left out. Each new concept is explained when introduced. We start with the skeleton of a UNITY program.

Program Compaction

declare

<declaration part>

initially

<initial condition of the data structures>

assign

<multiple assignment statements>

end

In UNITY, there is no notion of control flow. All statements occurring in the assignment section are repeatedly executed in a nonspecified, sequential order, starting from any state satisfying the initial condition. Hence, the computation is nondeterministic, and goes on forever.

However, in the program that we present, a stable state is reached after a finite time: whatever statement is selected for execution, the state does not change any more. This final state is determined by the initial contents of the data structures only, and is independent of the actual execution path.

In practice, the computation is terminated as soon as the final state is reached. We also execute statements concurrently, ensuring that this has the same effect as executing them sequentially in some order possible in the UNITY program.

We start our refinement by stating that we have an array M of cells, whose initial contents are equal to some constant array M^0 . M^0 is not used in the final program, and is only needed to describe the algorithm formally.

<declaration part> $\equiv$

i : integer;

M : array[bottom..top] of Cell;

<initial condition of the data structures> $\equiv$

$\langle \langle i : \text{bottom} \leq i \leq \text{top} :: M[i] = M^0[i] \rangle \rangle$

The latter construct between angle brackets is a quantified equation, and stands for $M[\text{bottom}] = M^0[\text{bottom}] \parallel M[\text{bottom} + 1] = M^0[\text{bottom} + 1] \parallel \dots \parallel M[\text{top}] = M^0[\text{top}]$. In this context, the $\parallel$ -operator indicates that each of the component equations holds at the same time.

Essentially, the algorithm consists of three operations that have to be done once for each cell. Each cell containing a marked reference towards some other cell has to be inserted in the relocation collection of that cell. When the relocation collection (if any) of a cell is complete, it has to be updated. And finally, the updated contents of each marked cell have to be moved to their new location.

```

<multiple assignment statements> =
  <□ i : bottom ≤ i ≤ top ::
    <insert cell i once>
  >□
  <□ i : bottom ≤ i ≤ top ::
    <update cell i once when complete>
  >□
  <□ i : bottom ≤ i ≤ top ::
    <move cell i once, when appropriate>
  >

```

$\langle \square i : \text{bottom} \leq i \leq \text{top} :: \dots \rangle$ stands for a quantified assignment statement, which expands to one assignment statement for each i in the range, separated by the $\square$ -operator. Such assignment statements are executed in an arbitrary order, infinitely often. The statements are further refined to guarantee that each operation has no effect unless all its preconditions are satisfied.

This is illustrated by the refinement of the “once” in the current level of description. “||” specifies that the component assignments are executed concurrently as one atomic action.

```

<insert cell i once> =
  <<insert cell i in the relocation collection of the cell it
  points to>
  || <mark cell i as inserted>
  > if <cell i has not yet been inserted>

```

We introduce boolean arrays *inserted*, *updated*, and *moved*, whose contents are true iff the corresponding action has been done on the cell. In the actual implementation, no such arrays will have to be maintained if we choose an execution order that guarantees that the conditions are satisfied. Our template algorithm however, specifies explicitly the minimal requirements for each operation.

```

<declaration part> + =
  inserted, updated, moved: array[bottom..top] of
  Boolean;

```

```

<initial condition of the data structures> + =
  << i : bottom ≤ i ≤ top ::
    inserted[i] = updated[i] = moved[i] = false>
  <mark cell i as inserted> =
    inserted[i] := true
  <cell i has not yet been inserted> =
    not inserted[i]

```

We refine the second and third part analogously. *complete(i)* stands for a boolean expression evaluating to true, when all cells originally containing a reference to cell i have been inserted.

```

<update cell i once when complete> =
  <<update the RC of cell i, if any> || updated[i] := true>
  if complete(i) and not updated[i]

```

Now we have to specify when it is “appropriate” to move (to copy the contents of) a cell i to its new location i' . First, cell i must not have a relocation collection itself; i.e., cells originally pointing to i already point to i' (*updated[i]*). Second, it must already have been inserted in a relocation collection (*inserted[i]*), because after the move operation, the contents of cell i are undefined. Moreover, if cell i originally contained a reference to some cell j , cell j must have been updated, so that cell i now contains a reference to cell j' (*not contained in a relocation collection any more*)—this can be written as a boolean expression involving M^0 . Finally, cell i' must be “empty”; this means that either cell i' has already been moved (and thus its contents may be overwritten), or $i = i'$, or the cell i' contained garbage from the beginning.

```

<move cell i once, when appropriate> =
  <<move the contents of cell i to their new location> ||
  moved[i] := true>
  if not moved[i] and updated[i] and inserted[i]
    and <not contained in a relocation collection any
    more> and empty(i)

```

We now specify the insert operation more formally (see Fig. 5). A cell has four subfields: its contents (*cts*), and the *M*, *C*, and *S* annotation bits. *end(i)* is a function returning the index of the value cell of the pointer chain of cell i . *ref(i)* returns the address of $M[i]$; and *index(r)* transforms a reference to a cell to the corresponding index.

```

<insert cell i in the relocation collection of the cell it
points to> =
  < < M[end(i)].cts := M[j].cts
    || M[j].cts := ref(i)
    || M[j].C := true
    || <add an S annotation to cell i if cell j has a C
    annotation>
  > where j = index(M[end(i)].cts)
  > if M[i].M and <M[end(i)] contains a reference>

```

The update operation is refined as follows (see Fig. 6).

```

<update the RC of cell i, if any> =
  < M[i].C := false
    || M[i].cts := M[end(i)].cts
    || <for each j in the relocation collection of cell i:
      <M[j].S := false || M[end(j)].cts=ref(i')>
  > if M[i].C

```

The move operation is specified below. Clearing the annotation of cell i indicates that its contents may be overwritten, and ensures that after the compaction all mark bits are reset. Clearing the *M* bit is sufficient, as the *C* and *S* bits are false already given the enclosing conditions.

```

<move the contents of cell i to their new location> =
  <<M[i'].cts := M[i].cts || M[i].M := false> if M[i].M

```

FIG. 7. A cyclic data structure.

5. DISCUSSION

The programming scheme above specifies the ordering constraints between the basic operations to guarantee a correct result. Several programs can be derived from it, tuned toward particular applications. But before we take a closer look at a few examples, we discuss the general characteristics.

The basic operations, as given, cannot handle cyclic reference chains in general. The reason for this is that it is not possible to represent a general graph structure as a linear annotated list. However, it should be noted that cyclic data structures can be made, without cyclic reference chains on the cell level (e.g., see Fig. 7: the cycle is at the record level).

There are two types of cyclic structures that need our attention. Cells containing a reference to themselves can be coped with by a careful extension of the insert and the update operations. The insert operation should leave such cells alone, and the update operation should check whether the updated value of the head is a reference to itself, and substitute it with a reference to the new location if appropriate. On the other hand, cyclic reference chains of length more than 1 can be handled by dividing the algorithm into an upward and a downward phase. All the operations used can be partitioned correspondingly in two components: an upward and a downward one. By combining these two extensions, all kinds of structures can be handled.

Although we used three separate annotation bits in our presentation, not all combinations do occur. For example, the *C* or *S* bits imply the *M* bit. This reduces our set of annotations to the following combinations: {*none*, *M*, *MC*, *MS*, *MCS*}. For parallel execution, we need also a locked state, indicating that the contents of the cell are temporarily invalid. These six cases can be represented in three bits, which must be atomically changed. To avoid deadlock, a particular order can be imposed on the acquiring of locks on cells: first according to the nesting level (top down), and second according to the order of the cells in the relocation collection (left to right). Locking can be greatly simplified while external references are inserted.

The main factor determining the execution time of the algorithm is the average length of the pointer chains of the cells: to find the value cell of a relocation collection, the pointer chain has to be traversed. However, the length of the pointer chains is not determined by the size of the data space, but by the characteristics of the data

representation. Also, updating the relocation set as soon as possible helps to reduce the average length of the pointer chains (see further).

Since we use in total only six annotations out of the eight possible with three bits, we may introduce some extra annotations to improve efficiency. For example, the *MC* annotation can be split into an *MC* and an *MC'* annotation, depending on whether the value cell of the pointer chain contains a reference to another cell or not. This would eliminate the need to follow the pointer chain if it does not contain a reference to another cell.

Although the algorithm as presented assumes contiguous memory, this is not really necessary. Of course, the computation of the new-function may then be more complicated, but this is not specific to the parallel algorithm. Only when the compaction is split into an upward and a downward phase, there is the extra requirement of a total ordering relation over the address space, which is easily satisfied. However, the memory must be globally accessible, and the algorithm is therefore best suited for shared memory architectures.

We now present two clear cut scheduling orders for the basic operations, and discuss how these influence the implementation of those operations. As already mentioned before, it is the intention to arrange the basic operations in such an order that the precondition for their execution is true, and thus need not be checked. Of course, there has to remain sufficient parallelism.

The In Mass Approach

In this approach, the whole compaction space is first converted into the intermediate format by the insert operations in a parallel for loop. Then, after synchronization, each cell of the compaction area is updated, and finally, again after synchronization, each cell is moved to its final destination. This yields the following Pascal-like program (*pfor* denotes a parallel for loop).

```

pfor i := bottom to top
  <insert cell i in the relocation collection of the cell it
  points to>
pfor i := bottom to top
  <update the RC of cell i, if any>
pfor i := bottom to top
begin
  <wait till empty(i')>
  <move the contents of cell i to their new location>
end

```

Comparing this program with the general scheme presented before, almost all the explicit preconditions have disappeared. The relocation collections are always complete before an update operation is done. Only the condition *empty(i')* has still to be checked.

The main drawback of this approach is that the nesting level of the relocation collection is clearly maximal, but

the severity of this depends largely on the characteristics of the data representation. On the other hand, the implementation of the basic operations can be simplified, because there is only one type of operation executed at a time. This means that, e.g., the insert operation should only be prepared for the interference of other insert operations occurring at the same time. This simplifies the mutual exclusion problem and reduces the overhead necessary for locking.

If there are no cyclic structures, there is also no need to have separate upward and downward phases.

The Chasing Approach

In this approach, we try to minimize the nesting level of the relocation collections. Therefore, the compaction area is partitioned in a number of segments, between which a total order must be defined. The number of marked cells must be known for each segment. There are an upward and a downward phase, separated by a synchronization point. Processors can dynamically switch between three states: inserting, updating, or moving a segment (the latter only in the second phase).

Each processor initially takes a segment at one end of the compaction area, and performs the insert operation on each cell of its segment with respect to the current direction. A segment can be updated as soon as it is completely inserted and all preceding segments have been inserted, because all the cells in the segment then have complete relocation collections. A similar condition holds for moving a segment during the second phase.

This approach has the advantage that pointer chains are kept short: the insert operation possibly increases the nesting level, but at the same time the update operation reduces the nesting level. Also, the algorithm has been split into an upward and a downward phase, and is therefore able to handle cyclic reference chains.

On the other hand, possible interference between concurrent operations is more complex now. Also, one or two extra passes are needed for doing the insert and update operations in two phases. One extra pass may be sufficient if the move operation can be integrated with the update operation in the second phase. Anyway, one should take an extra cost of 20% to 50% into consideration, which has to be compensated by the more efficient insert and update operations.

6. RESULTS

We implemented the in-mass approach in C on a Sequent Symmetry S-27 with eight processors, and compared the run-time characteristics with those of the sequential algorithm of Morris. We used three bits for the annotations in the parallel algorithm (using four bits would have made the manipulation of the annotations

FIG. 8. Test data structure.

slightly more efficient). Both algorithms were applied in turn on exactly the same copy of the compaction area—the sequential one using only two of the three available bits. No special optimizations were applied. All results mentioned are *averages* over 10 runs. In order to avoid the influence of paging, the size of the compaction space was fixed at 1 megabyte.³

Each set of runs was further characterized by the occupancy o of the compaction space (the relative number of marked cells) and the properties of the data structures represented by the marked cells. The cells are randomly selected, paying no attention to the locality of the references—in practical circumstances, our implementation would benefit from locality of the references, as memory is (dynamically) divided in chunks to operate upon with one processor. The data structures in the compaction area were characterized by w (the width, or the number of cells pointing directly to one cell) and d (the depth, or the length of the pointer) chains), as illustrated in Fig. 8. All this allows us to assess the relative efficiency of our algorithm as a function of the characteristics of the data representation.

The expression we put forward for the sequential algorithm of Morris is $t = (a + o \cdot f(w, d)) \text{Size}$, where t is the time, Size the size of the compaction space, and o the occupancy. a may then be interpreted as the time needed to inspect a cell, and $f(w, d)$ as the additional time to handle a marked cell. $f(w, d)$ is dependent on w and d , since w and d determine the relative number of pointers which have to be updated.

For the parallel algorithm, we propose the following expression for the total execution time (not taking into account possible locking conflicts): $t = (a' + o \cdot f'(w, d)) \text{Size}$. $f'(w, d)$ is the sum of the average times taken to move a cell, to update a pointer, and to follow a pointer chain. By taking o equal to 0, we can determine $\tau_0 = a/a'$, i.e., the relative time taken by the parallel algorithm with respect to the sequential one, for handling an unmarked cell. We find $\tau_0 = 1.57$, which corresponds roughly to the three passes needed for the parallel algorithm, versus the

³ Although paging effects are an important factor in the total garbage collection time, there should be no difference between the sequential and the parallel algorithm regarding this aspect.

TABLE I
 τ , Relative Overhead to Handle a Marked Cell

Depth	Width					
	0	1	2	3	4	5
0	1.20					
1		1.87	1.87	1.87	1.87	1.87
2		2.20	2.29	2.32	2.34	2.35
3		2.45	2.63	2.69	2.72	2.73
4		2.65	2.93	3.00	3.02	3.04
5		2.82	3.19	3.26	3.27	3.29

two passes needed for Morris's algorithm. $\tau(w, d)$ is defined as $(a' + f'(w, d))/(a + f(w, d))$. This is to be interpreted as the average time needed per cell to compact a data structure as represented in Fig. 8, relative to the sequential algorithm. The figures for $\tau(w, d)$ are given in Table I. By taking the weighted sum of τ_0 and $\tau(w, d)$, according to the properties of the data structures, one can estimate the total execution time for a particular application. Note that the width is much less important than the depth of the pointer structures. The main reason for this is that we used a recursive procedure call for each nesting level of the pointer chains. This aspect could be optimized a lot by inline expansion of the recursive invocation, or by running the program on RISC processors, on which a procedure call is very cheap.

In Table II, the average speedup of our algorithm is shown for up to seven processors—we reserved one processor for the operating system activities. The figures for seven processors include the standard deviation. The algorithm has an almost linear speedup. The sublinear effect was caused by:

- The sequential code for a processor to acquire a new chunk of work. This is negligible.
- The global synchronization points between the insert, update, and move phases. This effect can be re-

duced by dividing the work into successively smaller chunks.

- The move operation that behaves initially sequentially when o is too high, because before a processor can move a cell, another processor must have freed the destination cell. This is illustrated by the speedups for $w = 0$, $d = 0$, and $o = 0.1, 0.5$, and 0.9 , respectively. For larger w and d , this effect becomes less important, since the time needed for insert and update increases, while the time taken by the move operation is independent of w and d . Moving the cells initially in small chunks per processor helps to reduce this. Note that this problem only occurs when all cells have to be rearranged into one contiguous block.

- The mutual exclusion when concurrently accessing large linked structures. This effect can be seen in Table II for occupancy 0.5 with increasing w and d values. Since the parallelism is very fine grained, this factor grows slowly: we still have an average speedup of more than 6 when each pointer structure occupies 21% of the total number of cells (for $w = 6$ and $d = 6$, each pointer structure occupies 223 Kb; with three such structures and 1 Mb of memory, we have $o = 0.64$).

This illustrates that the contention between processors is very low, except in some uncommon cases.

7. APPLICATION TO THE AURORA SYSTEM

The suitability of our algorithm depends highly on the characteristics of the data structures. It may be argued that often the layout of the data structures can be adapted to the benefit of our algorithm without loss of efficiency for normal execution. However, in Prolog, pointer chains are very short in general [19], and can be collapsed [5]. For Aurora, the pointer chains are still shorter, because they may be broken by a conditional binding. A conditional binding goes through the Binding Arrays, and can be treated therefore as an external reference.

The data may be very irregularly spread among the stacks of the processors. Therefore, our algorithm may prove to be more efficient in this situation than other approaches that rely on data partitioning among the processors (each processor collecting its own stacks). It is also possible to determine after the marking phase which approach is best suited for the compaction.

We also want to preserve the possibility of compacting only the stack that actually runs out of space. The stacks of the other processors then remain uncompact, until they can be reclaimed by backtracking, or until they overflow themselves. Our algorithm is well suited for such situations.

We now estimate the efficiency of our algorithm executed with one processor, relative to the algorithm of Morris for an Aurora global stack. As the figures in Table

TABLE II
Speedups

o	w	d	Number of processors			
			2	4	6	7
0.10	0	0	2.00	3.99	5.96	6.95(± 0.01)
0.50	0	0	2.00	3.98	5.94	6.91(± 0.02)
0.50	1	1	1.99	3.96	5.89	6.85(± 0.03)
0.50	2	2	1.99	3.94	5.86	6.81(± 0.01)
0.50	3	3	1.99	3.94	5.86	6.80(± 0.02)
0.50	4	4	1.99	3.93	5.84	6.78(± 0.01)
0.50	5	5	1.99	3.92	5.79	6.72(± 0.05)
0.64	6	6	1.95	3.72	5.35	6.07(± 0.59)
0.90	0	0	1.75	3.27	4.56	5.25(± 0.07)

I are rather independent of w , especially for low values of d , we ignore the fact that in Aurora cells are likely to be referenced by more cells (due to the sharing of parts of the data structures) than in sequential Prolog. We estimate, based on data from Tick [19], that 2% of the non garbage cells form data structures of depth 2, 65% of depth 1, and 33% of the cells contain no references nor are referred to by any other cell. If we suppose that 50% of the heap contains garbage, we find $\tau = 0.5\tau_0 + 0.5(0.02\tau(-, 2) + 0.65\tau(-, 1) + 0.33\tau(0, 0)) \approx 1.6$. One level of in line code expansion of the recursive procedure calls reduces this to about 1.4. Given the small grain size of the parallelism, and hence the excellent speedup figures, this seems reasonable.

8. RELATED WORK

In the literature, several parallel garbage collector algorithms have been proposed, among which the mutator-collector algorithm of Dijkstra *et al.* ([10]) is perhaps most widely known. However, as far as the authors know, only two other parallel versions of Morris' sliding algorithm have been published.

In [15], Küchlin presents a parallel garbage collection algorithm, based on the algorithm of Morris, using the divide and conquer technique. We restrict our attention to his bottom up version—the top down one is similar.

Given p processors, the memory area is divided into p segments. During the first phase, each processor updates (locally) all intrasegment references in its segment using the algorithm of Morris, but without actually moving the cells. Intersegment pointers are left alone.

In the next phase, segments and processors are taken together in pairs. All intersegment references between the two segments in the same pair are updated by the corresponding pair of processors. Therefore, the two segments are scanned twice: once inserting and updating upward references, and once for the downward ones. References that have already been updated in one of the previous phases are protected with an extra protection bit. To divide the work among the two processors, each processor scans all cells, but the former updates only the cells containing an even reference, while the latter updates the cells containing an odd reference. This ensures mutual exclusion, while avoiding locking on the cell level.

For each next phase, segments are taken together in groups (called clusters). Each cluster consists of the segments of two clusters in the previous phase. Then twice as many processors update in parallel intercluster references between two clusters of segments. Now, each processor i of the k processors, cooperating to combine the k segments, updates all the cells containing references to a cell with address equal to i modulo k .

This process goes on until the whole area has been covered, after which the cells can be moved to their final location. Global synchronization is necessary between two successive phases.

In the paper, no implementation results are presented. However, we believe that the author underestimates the cost of scanning several times the contents of each cell (the number of inspections increases with the number of processors), while overestimating the cost of locking on the cell level.

In [9], Crammond proposed a parallel version of the algorithm of Morris. In this approach, each processor compacts its own local data structures (stacks) with a slightly extended sequential algorithm. Since no locking of cells is necessary, this can be done efficiently.

However, consistency of the interstack references has to be maintained. Therefore, during the marking phase all such references are redirected toward the indirect pointer stack (IPS) that corresponds to the stack they point to. The IPS then contains a reference toward the cell originally pointed to. During the first phase of Morris's algorithm, the references in the IPSs are relocated. During the second phase, all indirections are eliminated by updating each encountered reference that points toward an IPS cell.

The main advantage of this algorithm is that it retains much of the efficiency of the sequential algorithm, by avoiding to lock each cell. However, extra memory, proportional to the number of interstack references, is needed for the IPS's. This can be limited to some fixed size, at the cost of severely degraded performance if the amount turns out to be too small. Also, the algorithm is very sensitive to unbalanced stacks, as each processor takes a time proportional to the number of cells on its own stack. As a consequence, the sequential efficiency is outdone by the poor speedups when the number of processors increases. By contrast with this, our approach offsets the constant locking overhead with excellent speedups.

ACKNOWLEDGMENTS

This research was supported by ESPRIT Project 2471 (PEPMA) and the RFO-AI-02 project, sponsored by DPWB. The authors thank Maurice Bruynooghe for careful reading previous versions of this paper.

REFERENCES

1. Appleby, K., Carlsson, M., Haridi, S., and Sahlin, D. Garbage collection for Prolog based on WAM. *Comm. ACM* **31**, 6 (June 1988), 719–741.
2. Baker, H. G., Jr. List processing in real time on a serial computer. *Comm. ACM* **21**, 4 (Apr. 1978), 280–294.
3. Bentley, J. L. Programming pearls. *Comm. ACM* **29**, 5 and 6 (1986), 364–368 and 471–483.

4. Bevan, D. J. Distributed garbage collection using reference counting. *Proceedings PARLE*, Lecture Notes in Computer Science, Vol. 259. Springer-Verlag, Berlin/New York, 1987, pp. 176–187.
5. Carlsson, M., and Sahlin, D. Variable shunting for the WAM. *2nd NACLP Workshop on Logic Programming Architectures and Implementations*. Nov. 1990.
6. Chandi, K. M., and Misra, J. *Parallel Program Design—A Foundation*. Addison-Wesley, Reading, MA, 1988.
7. Cohen, J. Garbage collection of linked data structures. *Comput. Surv.* **13**, 3 (Sept. 1981), 341–367.
8. Courts, R. Improving locality of reference in a garbage collecting memory management system. *Comm. ACM* **31**, 9 (Sept. 1988), 1128–1138.
9. Crammond, J. A garbage collection algorithm for shared memory parallel processors. *Int. J. Parallel Programming* **17**, 6 (1988), 497–522.
10. Dijkstra, E. W., Lamport, L., Martin, A. J., et al. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, **21**, 11 (Nov. 1978), 966–975.
11. Glaser, H., Reeve, M., and Wright, S. An analysis of reference count garbage collection schemes for declarative languages. *Proceedings International Computer Science Conference*. Dec. 1988, pp. 641–648.
12. Goto, A., Kimura, Y., Nakagawa, T., and Chikayama, T. Lazy reference counting: An incremental garbage collection method for parallel inference machines. *Proceedings of the 5th International Conference and Symposium on Logic Programming*. 1988, pp. 1241–1256.
13. Knuth, D. E. *Fundamental Algorithms*, Vol. 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1968.
14. Knuth, D. E. Literature programming. *Comput. J.* **27**, 2 (1984), 97–111.
15. Küchlin, W. A space-efficient parallel garbage compaction algorithm. *Proc. ACM-SIGARCH, Fifth International Conference on Supercomputing*. Cologne, June 1991.
16. Lusk, E., Butler, E. R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D. H. D., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A., and Hausman, B. The Aurora or-parallel Prolog system. *Proceedings of the International Conference on Fifth Generation Computer Systems*. 1988, pp. 819–830.
17. Morris, F. L. A time- and space-efficient garbage compaction algorithm. *Comm. ACM* **8** (1978), 662–665.
18. Pittomvils, E., Bruynooghe, M., and Willems, Y. Towards a real time garbage collector for Prolog. *Proceedings of the Second Symposium on Logic Programming*. 1985.
19. Tick, E. *Memory Performance of Prolog Architectures*. Kluwer Academic, 1988.
20. Warren, D. H. D. An abstract Prolog instruction set. Technical Report, SRI International, Artificial Intelligence Center, Aug. 1983.
21. Warren, D. H. D. The SRI model for or parallel execution of Prolog—Abstract design and implementation. *Proceedings of the Symposium on Logic Programming*. 1987, pp. 92–102.
22. Weemeeuw, P., and Demoen, B. A la recherche de la mémoire perdue, or: Memory compaction for shared memory multiprocessors—design and specification. In Debray, S., and Hermenegildo, M. (Eds.). *Logic Programming: Proceedings of the 1990 North American Conference*. 1990, pp. 306–320.
23. Weemeeuw, P., and Demoen, B. A la recherche de la mémoire perdue, or: Memory compaction for shared memory multiprocessors: Technical Report, Katholieke Universiteit Leuven, Department of Computer Science, Nov. 1990. CW-Report 112.

PATRICK WEEMEEUW obtained the engineering degree in Computer Science in 1985 from the K.U. Leuven. Since then he is research assistant at the department of Computer Science. He has been involved in several research projects concerning logic programming. His current research interests are focused on garbage collection for parallel Prolog systems.

BART DEMOEN obtained a degree in Mathematics from the K.U. Leuven in 1975, a PhD in Theoretical Physics in 1979 and an additional degree in Computer Science in 1983, all at the K.U. Leuven. He then became the project leader of Prolog by BIM, a commercial Prolog implementation and is still involved in its design and implementation. In 1987, he became research assistant at the K.U. Leuven. He was appointed assistant professor in 1990 and teaches logic. He is involved in several national and ESPRIT projects. He is actively involved in the standardisation of Prolog. His research interests include implementation of sequential and parallel Prolog and of committed choice languages, program analysis and program transformation.