Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor

Akira Imai
ICOT†

Evan Tick
University of Oregon

CIS-TR-93-09
April 1993

Abstract

A parallel copying garbage collection algorithm for symbolic languages executing on shared-memory multiprocessors is proposed. The algorithm is an extension of Baker's sequential algorithm with a novel method of heap allocation to prevent fragmentation and facilitate load distribution during garbage collection. An implementation of the algorithm within a concurrent logic programming system, VPIM, has been evaluated and the results, for a wide selection of benchmarks, are analyzed here. We show (1) how much the algorithm reduces the contention for critical sections during garbage collection, (2) how well the load-balancing strategy works and its expected overheads, and (3) the expected speedup achieved by the algorithm.

This article appears in the *IEEE Transactions on Parallel and Distributed Computing*, 1993.

†Institute for New Generation Computer Technology, 1-4-28 Mita, Minato-ku Tokyo 108, Japan.

Department of Computer and Information Science
University of Oregon

1 Introduction

Efficient methods of garbage collection are especially crucial for the performance of so-called "fifth generation" multiprocessors. These machines are designed to execute high-level, symbolic languages such as parallel Lisp and concurrent logic programming. Because these languages are based on dynamic structure creation, garbage collection is critical for reclaiming storage during computation. Moreover, logic programming languages are applicative, so that destructive update cannot effectively reduce garbage production. On top of that, the family of concurrent logic programming languages do not backtrack, and as a result cannot automatically recover storage due to failed proofs.

This article introduces a new parallel garbage collection (GC) algorithm for these symbolic languages. Based on Baker's sequential stop-and-copy algorithm [2], our method is also invoked when one half of available memory has been exhausted. The processors (PEs) suspend execution and cooperate performing GC by copying active data objects to the other half of available memory. The innovative ideas in our algorithm are the methods with which to reduce contention and distribute work among the PEs during cooperative GC. We concentrate on shared-memory multiprocessors because they constitute the clusters around which several high-performance machines have been designed (e.g., PIM [6] and DASH [8]).

We have chosen KL1 [18], a concurrent logic programming language based on flat GHC [17], as a testbed on which to experiment with this GC algorithm. A naive implementation of KL1 on a multiprocessor consumes memory area rapidly since KL1 has neither destructive update nor backtracking. For example, an entire array must be copied when only a single element is updated. As a result, GC occurs so frequently that system performance is seriously jeopardized.

There have been several schemes proposed to deal with this excessive memory consumption problem. Chikayama and Kimura [3, 11] invented an incremental GC method based on a single reference bit. Saraswat *et al.* [14] proposed a logic language constrained to have single-consumer streams, thus allowing local reuse of memory. Ueda and Morita [19] outlined a mode-analysis technique that uncovers single-consumer streams, also allowing reuse. Abstract interpretation is another method that can be used, e.g., [9]. These schemes are all promising, yet in general they do not preclude the necessity of periodically doing a general garbage collection. Unless the scheme can guarantee to recover 100% of all garbage on-the-fly, global GC is needed.

The purpose of this article is to introduce a parallel, global garbage collection algorithm based on Baker's semi-space algorithm. A complete, detailed description of the algorithm is given. The performance characteristics of the algorithm executing with KL1 benchmark programs is also given. These results indicate that the algorithm

effectively avoids critical sections, achieving parallel efficiency of 51% to 97% on eight PEs. The article is organized as follows. Section 2 reviews Baker's sequential GC algorithm. Section 3 explains our parallel algorithm. Section 4 analyzes the empirical performance characteristics of the algorithm. Conclusions are summarized in Section 5.

2 Baker's Sequential Algorithm

In this section, Baker's sequential GC algorithm is reviewed. Two heaps are allocated although only one heap is actively used during program execution. When one heap is exhausted, all of its active data objects are copied to the other heap during GC. The advantages of this algorithm are that it is simple and fast because only *active* objects are accessed.

Figure 1 illustrates the algorithm, in C-like pseudo-code. Arrays newheap and oldheap represent the heaps as linear storage areas. In this and subsequent algorithms, heaps grow from low to high addresses, described as growing "downwards." Pointer S represents the *scanning point* and B points to the *bottom* of the new heap. Copying is initiated from *root pointer(s)* into the old heap that are guaranteed to reference, through some number of indirections, every active object in the program state. The active structures referenced by the root(s) are traversed and copied to B. A tag per heap cell can be used to indicate if an old-heap cell has been copied into the new heap, as described below.

As shown in Figure 1, the scanning function, defined in main(), examines the next new-heap cell addressed by S. If the cell addressed by S holds a pointer to the old heap, the referenced structure is copied to the new heap at B. Each cell[1] in old heap that is copied is overwritten with a *forwarding address*, i.e., a pointer to the new heap cell where the copy is made. This *mark* operation prevents any cell from being copied more than once. This procedure continues until S meets B, at which time all cells in old elements have been traversed in a breadth-first manner and copied to the new heap.

3 Parallel Extensions to Baker's Algorithm

There is potential parallelism inherent in Baker's algorithm in the copying and scanning actions, i.e., accessing S and B. We limit our view to shared-memory execution models

[1] In reality, and as specified in the algorithm, one need *not* overwrite forwarding addresses for arguments of a complex structure. For example, when $f(A, B)$ is copied, a forwarding address is written over functor f, but need not be written over the two arguments of f. Note of course that if A and B point to nested structures, overwriting will be done when they are copied.

```
copy(P) {
      — P is ptr into old heap —
      Temp := oldheap[P];
      if (Temp is forwarding address)
            newheap[S] := Temp;
      else {
            newheap[S] := B;
            Arity := arity(Temp);
            — mark old heap with forwarding address —
            oldheap[P] := B;
            for (i = 1; i ≤ Arity ; i++)
                  — copy contents of old element to new heap —
                  newheap[B++] := oldheap[P++];
      }
}

main() {
      S := B := base of new heap;
      for (i := 1; i ≤ # roots; i++ )
            copy( root(i) );
      while (S < B) do {
            P := newheap[S];
            if (P points into old heap)
                  copy(P);
            S := S+1;
      }
}
```

Figure 1: Sketch of Baker's Sequential Algorithm

in this article. In this section, we develop a parallel algorithm by stepwise refinement. This clarifies both the motivations and mechanisms of the scheme.

3.1 How to Exploit Parallelism

A naive method of exploiting this parallelism is to allow multiple processors (PEs) to scan successive cells at S, and copy into B. Such a scheme is bottlenecked by the PEs vying to atomically read and increment S by one cell and atomically write B by many cells. The contention would be unacceptable.

One way to alleviate this bottleneck is to create multiple heaps corresponding to the multiple PEs. For example, this is the structure of both the Concert Multilisp [7] and JAM Parlog [4] garbage collectors. Consider a model wherein each PE(i) is allocated private sections of the new heap, managed with private S_i and B_i pointers. Copying

from the old space could proceed in parallel with each PE copying into its private new sections. As long as the *mark* operation in the old space is atomic, there will be no erroneous duplication of cells. When copying is complete, all private sections of the new heap are treated as a single shared heap, and the old and new heaps are exchanged.

Managing private heaps during copying presents some significant design problems:

- Allocating multiple heaps within the fixed space must be done with minimal loss, i.e., fragmentation. For example, if for n PEs, each of n heaps is allocated to $1/n$ of the total space, and the heaps grow nonuniformly, then some heaps will exceed their allocation whereas other heaps will not. Thus a mechanism for dynamically reallocating new heap space during GC is necessary.

- If a PE finishes scanning the cells in its private heap, i.e., $S = B$, then the PE becomes idle. There must be a mechanism to distribute the work among the PEs throughout the GC.

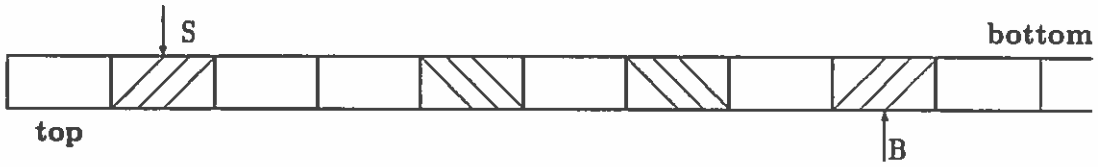
To efficiently allocate the heaps, two criteria must be met. First, the dynamic allocation must be invoked as infrequently as possible, because it is overhead that does not contribute anything to the computation. Second, the allocation must not leave unusable fragments or create a situation wherein no single fragment can hold the next structure to be copied. Given a shared-memory model, a scheme that achieves a balanced trade-off between these criteria, is to incrementally grow each heap in *chunks*.

> *Definition*: A chunk is defined to be a unit of contiguous space, a constant HEU cells in size (HEU ≡ Heap Extension Unit). □

We first consider a simple model, wherein each PE operates on its own heap, which it manages by its own pair of S and B pointers. Initially, a single empty chunk is allocated for each heap, with the S and B pairs pointing into the *top* (empty) element of the initial chunk. The B_{global} pointer is a state variable pointing to the global bottom of the new allocated space. Initially, B_{global} points to the entry one below the bottom-most initial chunk. Allocation of new chunks is always performed at B_{global}.

When a chunk has been completely copied into, the B pointer reaches the top of the next chunk (possibly not its own!). At this point a new chunk must be allocated to allow copying to continue. As mentioned, new chunks are always allocated at B_{global}, i.e., there is only one locale for heap growth.

There are two cases of B overflowing: either it overflowed from the same chunk as S, or it overflowed from a discontiguous chunk. In both cases, a new chunk is allocated. In the former case, nothing more need be done because S points into B's previous chunk, permitting its full scan. However, in the latter case, B's previous chunk will be lost if separated from S's by extraneous chunks (of other PEs for instance).



The shaded portions of the heap are owned by a PE(i) which manages S and B in the picture. Other portions are owned by any PE(j) where $j \neq i$. The two chunks shaded as '/' are referenced by PE(i) via S and B. The other chunks belonging to PE(i), shaded as '\', are *not* referenced. To avoid losing these, they are registered in the global pool.

Figure 2: Chunk Management in Simple Heap Model

The problem of how to "link" the discontiguous areas, to allow S to freely scan the heap, is finessed in the following manner. In fact, the discontiguous areas are not linked at all. When a new chunk is allocated, the B's previous chunk is simply added to a *global pool.* This pool holds chunks for load distribution, to balance the garbage collection among the PEs. Unscanned chunks in the pool are scanned by idle PEs which resume work (see Figure 2).

We now extend the previous simple model into a more sophisticated scheme that reduces the fragmentation caused by dividing the heap into chunks of uniform size. Data objects (also called "structures") come in various sizes, large and small (Section 5.4 gives a full characterization for KL1). Imprudent packing of objects into chunks might cause fragmentation, leaving useless area in the bottom of chunks. To avoid this problem, each object is allocated the closest quantum of 2^n cells (for integer $n <$ log(HEU)) that will contain it. Otherwise, larger objects are allocated the smallest multiple of HEU chunks that can contain them. When copying objects, with sizes less than HEU, into the new heap, the following rule is observed: "All objects in a chunk are always uniform in size." If HEU is an integral power of two, then no portion of any chunk is wasted. When allocating heap space for objects of size greater than one HEU, contiguous chunks are used.

In this refined model, chunks are categorized by the size of the objects they contain. To effectively manage this added complexity, a PE manipulates multiple $\{S, B\}$ pairs (called $\{S_1, B_1\}$, $\{S_2, B_2\}$, $\{S_4, B_4\}$, ..., and $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$). Initially, each PE allocates multiple chunks[2] with S_i and B_i set to the top of each chunk. Objects with size HEU and greater are managed by $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$. Since it is impossible to initially

[2]Since $HEU = 2^n$, n chunks are allocated.

allocate contiguous chunks for these large structures (because their sizes are unknown before GC), they are allocated on demand. Effectively, the pointer pairs reference subheaps. We refer to the $\{S_{\mathrm{HEU}}, B_{\mathrm{HEU}}\}$ pair, which serves much the same function as $\{S, B\}$ in Baker's algorithm, as an *overflow heap.*

Note that allocating structures in this manner trades off unused space within structures vs. fragmenting space during GC. A degenerate case of allocation can waste up to half of available memory; however, this does not happen in practice since most structures are small. To further reduce unused space, the algorithms described here can be modified to include $\{S_3, B_3\}$, $\{S_5, B_5\}$, etc. heaps for small structure sizes. There are various implementation advantages of "knowing where your wasted space is located," such as facilitating the free-list management required by incremental GC schemes, e.g., MRB.

Referring back to Figure 2, recall that shaded chunks of the heap are owned by PE(i) and non-shaded chunks are owned by other PEs. The chunks shaded as '/', in the extended model, contain objects of some fixed size k, and are managed with pointer pair $\{S_k, B_k\}$. Chunks shaded as '\' are either directly referenced by other pointer pairs of PE(i) (if they hold objects of size $m \neq k$), or are kept in the global pool.

3.2 Parallel Algorithm

The parallel algorithm is summarized in detail in Figures 3, 4, and 5. The pseudo-code represents the program executed by a single PE during garbage collection (all PEs follow this routine). In the code, *P(var)* and *V(var)* represent mutual exclusion primitives ensuring that only one process can modify *var* within that region at any one time.

Initially, PE(j) copies objects, referenced by its own GC root pointer(s), to some heap at B_k^j for appropriate size k.[3] PE(j) then scans the allocated unit of size k (in our design, k is a power of two). Figure 3 specifies the top-level scan-all() procedure that continually scans each private heap owned by PE(j). For each heap of size i that has as-yet-unscanned cells, scan(i) is invoked. If PE(j) becomes idle, under the condition that $\forall k(S_k^j = B_k^j)$, then it grabs work from the global pool, and begins scanning again. Thus the load is distributed between PEs which have many cells to be scanned and PEs which do not. The algorithm terminates when $\forall j, k(S_k^j = B_k^j)$, *and* the global pool is empty.

Procedure scan(n) differs from that in Baker's algorithm, primarily because the

[3] We do not explicitly indicate the PE superscript j in the pseudo-code presented. It should be understood that the code given is executed on a single PE, which references its own private heaps, via its own set of heap pointers. Physically, all the heaps are intertwined in shared memory.

scanning heap and the copying heap may differ. The cell scanned at S_n may point into the old heap to an object with a size other than n. Procedure copy(P,S_n) copies the old structure pointed to by P, leaving a forwarding pointer at the new heap location indicated by S_n, and returns the size (up to HEU) of the object copied. At this point, both the B and S pointers involved must be checked for chunk overflow. If no copying was necessary (when the size m is either zero or HEU), then no B pointer can overflow. Otherwise, B_m and S_n are both checked for overflow (described in detail below).

Figure 4 lists the copy(P,S) procedure. There are three main cases buried in this specification: (1) P points to a forwarding address, obviating the need for copying, (2) P points to a structure of size less than HEU, and (3) P points to a large structure of size HEU or greater. In latter case, the overflow heap is used. In both copying cases, the scanned cell in the new heap and the head of the structure in the old heap, are overwritten with a forwarding address (4). This forwarding address is the new-heap location where the structure is now copied into (5).[4] Large structures must be allocated directly at B_{global}, copied into a multiple of chunks. These chunks are immediately linked into the global pool (6) with procedure add-to-pool (described below).

Figure 5 lists the procedures for checking S and B pointers for chunk overflow, as well as managing the global pool of chunks. Four shorthand macros are used here:

$$\text{is-at-top-of-HEU(x)} \equiv ((\text{x mod HEU}) = 0)$$
$$\text{Top-of-HEU(x)} \equiv (\text{x div HEU}) \cdot \text{HEU}$$
$$\text{Bottom-of-HEU(x)} \equiv (\text{Top-of-HEU(x)} + \text{HEU} - 1)$$
$$\text{Top-of-prev-HEU(x)} \equiv (\text{Top-of-HEU(x)} - \text{HEU})$$
$$\text{Bottom-of-prev-HEU(x)} \equiv (\text{Top-of-HEU(x)} - 1)$$

When B_m points to the top of a chunk, an overflow from the previous chunk has occurred. Two actions are performed to service this overflow. As described in the overview given previously, first the previous chunk must be added to the pool if not referenced by S_m, to prevent loss (7). Next, a new chunk is allocated at the global heap bottom, and B_m is reassigned there (8). Servicing an S_n overflow is simpler: the pointer is reassigned to the top of the chunk referenced by B_n (9). This effectively skips over intervening chunks of size n, that have been explicitly added to the pool.

The global chunk pool is managed as a stack, Pool, that grows upwards from a base address, Bottom-of-Pool. The top of stack pointer is PoolPtr. In our model, chunks from *all* heaps share a single pool (this can be extended to multiple pools in obvious

[4] Each data structure has a *true* size, for example $f(a,b,c,d)$ has arity five, and a power-of-two *allocated* size, in this case, eight. In the pseudo-code, Arity is the allocated size. The cells allocated beyond the true size, which need never be initialized, also need not be copied or scanned by virtue of a simple optimization. This optimization is not described in the pseudo-code.

```
scan-all() {
    repeat {
        — scan each size heap for given PE —
        repeat {
            Scanning := False;
            for (i := HEU; i ≥ 1 ; i := i/2;)
                if ( S_i < B_i ) {
                    scan(i);
                    Scanning := True;
                }
        } until not(Scanning);

        — try to get a new chunk from pool —
        if ( get-from-pool(S_HEU, B_HEU) ) {
            — Success getting a new chunk from pool —
            — S_HEU and B_HEU is used to scan the chunk —
        } else
            declare that I'm idle;
    } until (all PEs idle);
    — GC has terminated here —
}

scan(n) {
    — scan heap size n for given PE —
    while (S_n < B_n) do {
        P := newheap[S_n];
        if ( P points into old heap ) {
            — m is the updated size —
            m := copy(P,S_n);
            if ( (m ≠ 0) and (m ≠ HEU) )
                checkB(m);
        }
        S_n := S_n + 1;
        checkS(n);
    }
}
```

Figure 3: Parallel GC Algorithm: Scanning All the Heaps

```
copy(P,S_c) {
    — P is ptr into old heap —
    — S_c is current scanning point —
    P( oldheap[P] );
    Temp := oldheap[P];
    if ( Temp is forwarding address ) {                          (1)
        V( oldheap[P] );
        newheap[S_c] := Temp;
        return(0);
    } else {
        Arity := arity(Temp);
        if ( Arity < HEU )
            m := Arity;                                          (2)
        else {
            m := HEU;                                            (3)
            P (B_global);
            from := B_HEU := B_global;
            B_global := B_global + Arity;
            V (B_global);
        }
        newheap[S_c] := B_m;                                     (4)
        — mark old heap with forwarding address —
        oldheap[P] := B_m;
        V( oldheap[P] );
        for (i := 1; i ≤ Arity; i++)
            — copy contents of old element to new heap —
            newheap[B_m++] := oldheap[P++];                      (5)
        if ( Arity ≥ HEU ) {
            — add the chunks to pool immediately —
            add-to-pool(from, B_HEU);                            (6)
        }
        return(m);
    }
}
```

Figure 4: Parallel GC Algorithm: Copying Cells

ways — we comment about single-pool utility in Section 5.3). Full chunks are added to the pool with the intention of offloading them, at a future time, to idle PEs for scanning. The scanning operation is independent of the size of the objects within the chunk, so these offloaded chunks can be added to any of the empty heaps owned by the idle PE. In procedure scan-all() (Figure 3), we arbitrarily offload the chunks to the overflow heap.

3.3 Optimization for Load Balancing

In the previous algorithm, selecting an optimal HEU, the heap extension unit, is a difficult choice. As HEU increases, B_{global} accesses become less frequent (which is desirable, since contention is reduced); however, the average distance between S and B (in units of chunks) decreases. This means that the chance of load balancing decreases with increasing HEU.

One solution to this dilemma is to introduce an independent, constant size unit for load balancing.

> *Definition*: The load distribution unit (LDU) is this predefined constant, distinct from HEU, enabling more frequent load balancing during GC. We assume HEU = kLDU, for integer $k > 0$. □

In general, the optimized algorithm incorporates a new rule in procedure checkS, wherein if $(B_k^j - S_k^j >$ LDU), then the region between the two pointers (i.e., the region to be scanned later) is pushed onto the global pool. The new procedure is summarized in Figure 6.

The optimized procedure checkS is split into three conditions concerning the object size n of the chunk referenced by the scan pointer: (1) n is not greater than LDU, (2) if n is between LDU and HEU, and (3) n not less than HEU. The latter two cases comprise the same function as does the unoptimized version of checkS (Figure 5).[5] The first condition is where the optimization is occurs, for two cases when S_n and B_n either share the same chunk, or reference different chunks (see Figure 7). In the former case, all the integral LDUs between S_n and B_n can be added to the pool. In the latter case, all the LDUs remaining in S_n's chunk, and the integral LDUs preceding B_n, can be added to the pool.

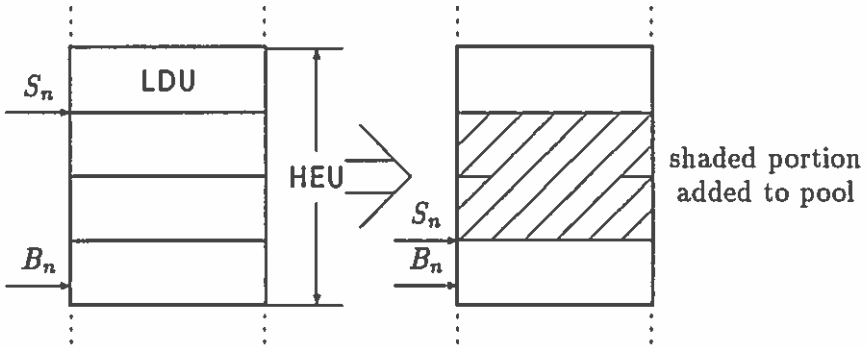
[5] The condition of (LDU $< n <$ HEU) prevents exploitation of the load balancing optimization because of a conflicting optimization we are using to avoid extra memory references (Footnote 4). Essentially, for a structure with an allocated size larger than its true size, we must prevent the distribution of LDUs filled with only garbage cells.

```
checkB(m) {
      — check if B_m has overflowed chunk —
      if ( is-at-top-of-HEU(B_m) ) {
            if ( Top-of-HEU(S_m) ≠ Top-of-prev-HEU(B_m) )
                  add-to-pool(Top-of-prev-HEU(B_m),                    (7)
                              Bottom-of-prev-HEU(B_m));
            — allocate a new chunk at bottom of heap —
            P (B_global);
            B_m := B_global;                                           (8)
            B_global := B_global + HEU;
            V (B_global);
}     }

checkS(n) {
      — check if S_n has overflowed chunk —
      if ( n ≠ HEU )
            if ( is-at-top-of-HEU(S_n) )
                  S_n := Top-of-HEU(B_n);                              (9)
}

add-to-pool(from, to) {
      — add unscanned chunk to pool (stack) —
      P ( PoolPtr );
      Pool[PoolPtr] := from;
      Pool[PoolPtr+1] := to;
      PoolPtr := PoolPtr + 2;
      V ( PoolPtr );
}

get-from-pool(from, to) {
      — get unscanned chunk from pool (stack) —
      P ( PoolPtr );
      if ( PoolPtr > Bottom-of-Pool ) {
            to := Pool[PoolPtr − 1];
            from := Pool[PoolPtr − 2];
            PoolPtr := PoolPtr − 2;
            V ( PoolPtr );
            return(True);
      } else { — pool is empty, fail in getting a chunk —
            V ( PoolPtr );
            return(False);
}     }
```

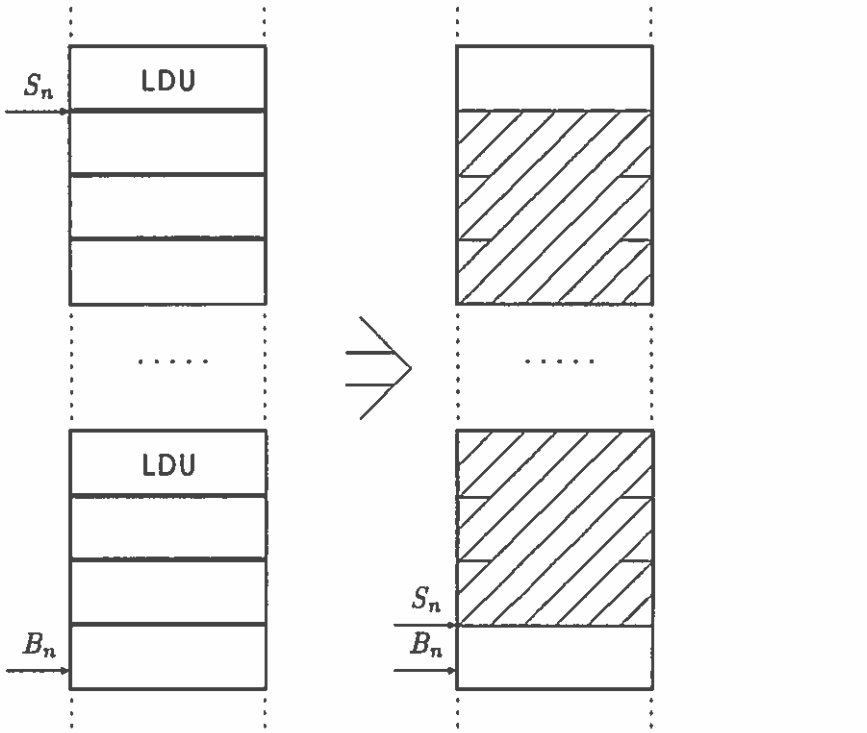
Figure 5: Parallel GC Algorithm: Load Distribution

```
Top-of-LDU(x) ≡ (x div LDU) · LDU
Bottom-of-prev-LDU(x) ≡ (Top-of-LDU(x) − 1)
is-at-top-of-LDU(x) ≡ ((x mod LDU) = 0)

checkS(n) {
      — check if S_n has overflowed subchunk —
      if ( is-at-top-of-LDU(S_n) ) {
            if ( n ≤ LDU ) {
                  switch (distance(S_n ,B_n)) {
                        case (within same chunk): {
                              add-ldu-to-pool(S_n,Bottom-of-prev-LDU(B_n));
                        }
                        case (in different chunks): {
                              add-ldu-to-pool(S_n,Bottom-of-HEU(S_n));
                              add-ldu-to-pool(Top-of-HEU(B_n),
                                          Bottom-of-prev-LDU(B_n));
                        }
                  }
                  S_n := Top-of-LDU(B_n);
            } else
                  if ( LDU < n < HEU )
                        — see Footnote 5 —
                        if ( is-at-top-of-HEU(S_n) )
                              S_n := Top-of-HEU(B_n);
                  — else (HEU ≥ n) do nothing —
      }
}

add-ldu-to-pool(from, to) {
      if ( from ≠ to )
            add-to-pool(from, to);
}
```

Figure 6: Optimized GC Algorithm: Load Distribution When S Overflows



Case 1: $\{S_n, B_n\}$ within same chunk. A portion between S_n and Bottom-of-prev-LDU(B_n) is pushed into global pool.



Case 2: $\{S_n, B_n\}$ in different chunks. Two portions between S_n and Bottom-of-HEU(S_n), and between Bottom-of-HEU(B_n) and Top-of-prev-LDU(B_n) are pushed into global pool.

Figure 7: Chunk Management in Optimized Heap Model

4 Relationship to Previously-Published Algorithms

As memory size increases, we desire that the ratio of GC time to total time will decrease. If the ratio increases, future memory sizes would cause intolerable GC overheads. With a copying algorithm, GC time depends on the number of active objects and does not depend on the size of memory, so the ratio decreases. With a compaction algorithm (e.g., mark and sweep), GC time depends on the size of memory and is independent of the number of active objects, thus the ratio remains constant. Reference-counting algorithms avoid this problem by incremental collection and thus retain higher locality of reference than copying algorithms. In summary, the main drawbacks of the copying algorithm proposed is 1) that twice as much heap memory is needed than in the alternative methods; 2) reference locality is lower than in free-list management-based algorithms, such as reference counting, and 3) collection is not done in real time. However, we use copying because we believe that for non-real time environments, its efficiency advantages outweigh these disadvantages.

We now compare our parallel algorithm to three previously-published garbage collectors for Concert Multilisp [7] and JAM Parlog [4], and for shared-memory multiprocessors [1].

Appel *et al.* [1] is unique in that it operates in real time with no customized hardware. For a Lisp benchmark executing on a five processor DEC Firefly, the GC system latency was 80 times shorter than a sequential stop & copy algorithm. GC speedup (as defined in the next section) is not relevant, whereas overlap of GC with user execution is: they achieved 62% overlap for the benchmark.

Both Halstead's and Crammond's garbage collectors [7, 4] statically divide the entire heap area by the number of PEs, and each PE copies active objects onto its private (new heap) area. The main difference between the two algorithms is in their handling of *remote* objects, which are objects not in the private area of the old heap.

In Halstead's algorithm, remote objects are copied by the PE which first encounters the object, which means that GC changes the "ownership" of an object. On the other hand, in Crammond's algorithm, remote objects remain remote, even after GC. This is implemented by *asking* the owner to copy objects. For this purpose, an *indirect pointer stack* area is statically allocated, and a global remote object counter is maintained.

In our algorithm, we do not distinguish between remote and local cells because automatic load distribution among PEs is implemented by the mechanisms previously described. Therefore the distinction between remote and local cells serves no purpose. Thus we must lock any cell when writing a forwarding address to that cell.[6] Halstead's

[6] However, if the MRB of a list is off, we copy the list without writing a forwarding address.

algorithm also requires this, but interestingly, Crammond's algorithm need lock only "asked" cells. Locking, however, is cheap on the shared-memory machines we are targeting, such as PIM. On a NUMA (Non Uniform Memory Access) shared-memory multiprocessor, we believe Crammond's scheme may work better.

We do not attempt to improve spatial locality with our algorithm. The global pool balances the load, and may indirectly improve locality (compared to Baker's algorithm) because S and B pointers stay closer together, but we have not examined this. In Section 5.3 we discuss a variant of the algorithm that may improve locality further.

The advantages of our algorithm, over these alternatives, are clarified by the following points.

Load Distribution: No dynamic load distribution mechanisms among PEs is given in Halstead's or Crammond's algorithms. We believe that load distribution must be done not only during normal execution but also during GC. Otherwise, the performance of GC largely depends upon the performance of the standard scheduler (for user-program execution), which determines the initial load distribution of GC.

Private Heap Overflow: Halstead comments that Concert Multilisp does exhibit instances when a single PE exhausts its allocated heap space, requiring reallocation of space from either a global pool or another PE. He claims this is "only a minor extension of the basic garbage collection algorithm," however, we tend to disagree. Our philosophy is to exploit the shared-memory model, incrementally growing all heaps by chunks, during GC. This guarantees that fragmentation cannot occur during GC, obviating the need to devise a fair heap reallocation policy among PEs.

Size of Extra Spaces: Since GC is invoked when memory is full, the extra memory space required by the algorithms should be as small as possible. In Crammond's algorithm, whenever a PE scans a cell holding a remote pointer, a pointer to this cell (equivalent to the value of our S pointer) is pushed onto the *indirect-pointer stack* of the owner of the remote object. It is difficult to statically determine the necessary size of the indirect-pointer stack area. Crammond concluded (for small benchmarks) that the space required for this stack is relatively small, less than 1% of the heap space. However, considering the worst case when all data objects consist of all remote reference pointers, the indirect-pointer stack requires the same size as the heap to guarantee that GC will terminate. In our algorithm, to guarantee the termination of GC, the global-pool stack needs only $2\times$ (HeapSize / LDU) words.

Benchmark	Heap (Kw)	# GCs	#Red. x1000	#Susp. x1000	comments
BestPath	192	6	394	57	shortest-path problem (30x30 nodes)
Boyer	128	4	529	18	toy theorem prover
Cube	128	5	291	6	logical constraints (7 cubes) †
Life	128	4	353	236	life game simulation (38x38 nodes)
MasterMind	128	8	1525	5	game-playing program †
MaxFlow	128	3	80	35	max. flow in network (80 nodes,123 links)
Pascal	64	13	285	1	Pascal's triangle (row 250)
Pentomino	64	7	188	9	2-D puzzle packing (5x5 square,6 pieces) †
Puzzle	128	19	1254	145	3-D puzzle packing (7 pieces) †
SemiGroup	448	6	732	12	calculation of Brandt semigroup (5 tuples)
TP	64	23	564	47	theorem prover (4-Cook's wff)
Turtles	320	1	1178	62	logical constraints (12 cards) †
Waltz	128	6	1207	19	3-D drawing constraints (38 nodes) †
Zebra	320	9	405	2	logical constraints (extended version) †

† all solutions search

Table 1: Summary of the Benchmarks

5 Evaluation

The parallel GC algorithm was evaluated for a large set of medium-sized benchmark programs (from [16, 15] and other sources), executing on VPIM, a parallel KL1 emulator. The measurements presented in this article were collected on a Sequent Symmetry with 16 processors, although we used at most eight. Because of the parallel execution, slight scheduling differences affect the number of GCs, reductions, suspensions, etc. Statistics in the tables were measured on eight PEs with HEU=256 words and LDU=32 words, unless specified otherwise.

The evaluation of all benchmarks was done with the MRB (Multiple Reference Bit [3]) optimization enabled, facilitated by support from the VPIM system. MRB, a method of incremental garbage collection, gives us a realistic characterization of the garbage produced by the programs. Other methods of local memory reuse, as mentioned in Section 1, differ mainly in their execution overheads, and we believe that the results presented here are informative with respect to those schemes as well.

The benchmarks are summarized in Table 1, where "heap size" is the statically allocated, maximum size of the old heap (which equals the size of the new heap). Note that the measurements presented here represent a *single* execution of each benchmark. Averages are calculated among the multiple GCs within a benchmark. In the next sections we analyze, with respect to varying LDU and HEU sizes, various algorithm

characteristics: load balancing effectiveness and overhead, speedup, global-heap-bottom access frequency, global-pool access frequency, and active data cell distribution by type.

VPIM is an appropriate system upon which to test garbage collection because its storage management is not naive. For a group of nine programs similar to our benchmarks, the number of words generated per reduction was measured. Without MRB, this metric ranged from 1.4–21, with six of the programs generating less than 1.8 words/reduction. With MRB, the reduced range was 0.26–12. The benefit of MRB ranged from none to 1100% reduction in words generated. For in-depth analysis of MRB performance, see Nishida *et al.* [11].

5.1 Load Balancing and Speedup

To evaluate load balancing during GC, we define the *workload* of a PE, and the *speedup* of a system, as follows:

$$\begin{aligned} \text{workload(PE)} &= \text{number of cells copied} + \text{number of cells scanned} \\ \text{speedup} &= \frac{\sum \text{workloads}}{max(\text{workload of PEs})} \end{aligned}$$

The workload value approximates the GC time, which cannot be accurately measured because it is affected by DYNIX scheduling on Symmetry [12]. Workload is measured in units of *cells referenced.*[7] Speedup is calculated assuming that the PE with the *maximum* workload determines the *total* GC time. Note that speedup represents only how well load balancing is performed, and does not take into account any extra overheads of load balancing (which are tackled separately in Section 5.3). We also define the *ideal speedup* of a system:

$$\text{ideal speedup} = min\left(\frac{\sum \text{workloads}}{max(\text{workload for one object})}\ ,\ \#\text{PEs}\right)$$

Ideal speedup is meant to be an approximate measure of the fastest that n PEs can perform GC. Given a perfect load distribution where $1/n$ of the sum of the workloads is performed on each PE, the ideal speedup is n. This perfect distribution is rarely achievable in practice. There is an obvious case when in fact an ideal speedup of n cannot be achieved: when a single data object is so large that its workload is greater than $1/n$ of the sum of the workloads. In this case, GC can complete only after the workload for this object has completed. These intuitions are formulated in the above definition.

Table 2 summarizes the average workload and speedup metrics for the benchmarks.

[7] We roughly estimate two memory accesses per cell referenced. A scan operation requires one read (if the object is atomic) or one read and one lock-and-read otherwise. A copy operation requires one read and one write, per cell. Additionally, one write and one write-and-unlock is required per object.

Benchmark	average workload x1000	speedup size of LDU 32w	64w	128w	256w	ideal
BestPath	165	7.15	7.06	6.46	6.36	8.00
Boyer	47	5.67	5.83	4.38	4.12	8.00
Cube	139	7.74	7.67	7.35	6.83	8.00
Life	101	7.10	6.86	6.31	6.29	8.00
MasterMind	4	2.50	2.48	2.58	2.48	2.87
MaxFlow	95	4.06	3.84	3.70	2.86	8.00
Pascal	5	2.67	2.91	3.45	2.77	7.25
Pentomino	3	4.34	3.34	3.67	4.21	8.00
Puzzle	17	2.63	2.84	2.58	2.61	2.92
SemiGroup	496	7.75	7.28	7.49	7.02	8.00
TP	17	2.49	2.39	2.43	2.33	2.79
Turtles	203	7.79	7.44	7.20	7.22	8.00
Waltz	32	4.38	2.92	2.31	1.64	8.00
Zebra	167	6.27	6.04	6.42	6.28	8.00

Table 2: Average Workload and Speedup (on eight Symmetry PEs, HEU=256 words)

Workload is listed as thousands of cells referenced. Averages are arithmetic means calculated over the GCs executed. The table shows that benchmarks with larger workloads display higher speedups. For instance, benchmarks with workloads over 100,000 cells referenced, achieved speedups greater than six, for any size LDU. This illustrates that the algorithm is quite practical.

In some benchmarks, such as MasterMind, Puzzle and TP, ideal speedup is limited (2–3). As explained above, this limitation is due to inability of cooperation among the PEs in accessing a single large structure. The biggest structure in each of the benchmark programs is the *program module*. A program module is actually a first-class structure and therefore subject to garbage collection (necessary for a "self-contained" KL1 system, including a debugger and incremental compiler). In practice, application programs consist of many modules, opposed to the benchmarks measured here, with only a single module per program. Thus the limitation of ideal speedup in MasterMind and Puzzle is peculiar to these toy programs.

In benchmarks such as Pascal and Waltz, the achieved speedup is significantly less than the ideal speedup. These programs create many long, flat lists. When copying such lists, S and B are incremented at the same rate. The proposed load distribution mechanism does not work well in this degenerate case. Our method works best for deeper structures, so that B is incremented at a faster rate than S (especially in the early stage of GC). In this case, ample work is uncovered and added to the global pool

Benchmark	Size of LDU (words)					
	32w			64w	128w	256w
	Naive	Smart	N/S	N/S	N/S	N/S
BestPath	124,569	2,305	54	60	56	57
Boyer	22,779	587	39	44	31	38
Cube	158,923	1,686	94	90	83	93
Life	68,687	1,326	52	52	52	52
MasterMind	3,427	522	7	6	6	7
MaxFlow	55,639	699	80	73	72	62
Pascal	14,437	917	16	15	16	16
Pentomino	9,480	305	31	32	29	30
Puzzle	38,831	1,486	26	29	27	27
SemiGroup	708,183	6,229	114	114	114	115
TP	40,455	1,738	23	23	23	22
Turtles	28,209	566	50	49	49	55
Waltz	39,476	715	55	62	56	60
Zebra	126,761	3,885	33	32	33	32

Table 3: Total Number of Updates of B_{global} over all GCs (8 PEs, HEU=256 words). "Naive" assumes all objects copied to B_{global}, N/S is ratio of naive updates to updates made by chunked algorithm.

for distribution.

5.2 Reducing Contention at the Global Heap Bottom

In this section we analyze the frequency with which the global heap-bottom pointer, B_{global}, is updated (for allocation of new chunks). This action is important because B_{global} is shared by all the PEs, which must lock each other out of critical sections that manage the pointer. We show that the algorithm described significantly reduces contention for these critical sections.

The update frequency of B_{global} depends on the value of the heap extension unit (HEU) and the average size of active objects, but is not affected by the size of LDU. For instance, in Zebra (given HEU = 256 words and LDU = 32 words), B_{global} is updated 3,885 times within all GCs. If B_{global} were updated whenever a single object was copied to the new heap, the value would be updated 126,761 times. Thus update frequency is reduced by over 32 times compared to this naive update scheme.

General results for all the benchmarks are summarized in Table 3. For each LDU size measured, we show the ratio of the naive updates to the (smart) updates made by our algorithm. Note that MasterMind achieved the *least* reduction in update frequency

Benchmark	avg # pool access/GC				avg workload/pool access (x1000)			
	Size of LDU (words)				Size of LDU (words)			
	32w	64w	128w	256w	32w	64w	128w	256w
BestPath	421.0	139.6	84.4	45.8	0.4	1.2	2.0	3.6
Boyer	208.8	131.3	24.3	12.8	0.2	0.4	1.9	3.7
Cube	609.4	241.6	96.3	55.5	0.2	0.6	1.4	2.5
Life	145.8	66.5	29.8	14.8	0.7	1.5	3.4	6.9
MasterMind	3.9	1.5	1.1	1.0	1.1	2.8	3.7	4.2
MaxFlow	211.3	75.0	37.0	10.0	0.4	1.3	2.6	9.5
Pascal	1.6	1.0	1.0	1.0	3.5	5.6	5.6	5.6
Pentomino	134.3	65.3	21.0	7.5	0.2	0.5	1.5	4.1
Puzzle	51.6	30.6	10.5	4.9	0.3	0.6	1.7	3.6
SemiGroup	1700.7	910.8	439.3	29.6	0.3	0.5	1.1	16.8
TP	44.4	19.8	8.8	4.6	0.3	0.8	1.9	3.7
Turtles	1427.0	640.0	314.0	136.0	0.1	0.3	0.6	1.5
Waltz	76.0	36.0	11.5	1.4	0.4	0.9	2.8	22.7
Zebra	2127.9	920.2	467.7	222.4	0.1	0.2	0.4	0.8

Table 4: Access Characteristics of the Global Pool (8 PEs, HEU=256 words). Left side shows pool accessed most frequently when LDU is small. Right side shows work delivered per access is largest when LDU is large.

— only a factor of seven, significantly below that of the other benchmarks. This can be explained by the small workloads involved. As described in section 3.2, log(HEU) initial chunks are allocated per PE before GC starts. Thus MasterMind initially allocates 8 (PEs) × 8 (chunks/PE) × 8 (GCs) = 512 chunks, or 98% of all chunks allocated. In other words, the program is doing the minimum required allocation and so reduction in updates is limited. Excluding this benchmark, the ratios of the other programs range from 15–114.

5.3 Global-Pool Access Behavior

Table 4 shows the average number of global-pool accesses made by the benchmarks, and the average number of cells referenced (in thousands) by the benchmarks per global-pool access. These statistics are shown with varying LDU size. The data confirms that, except for Pascal and MasterMind, the smaller LDU, the more chances to distribute unscanned regions, as we hypothesized. Grossly, the amount of distribution overhead is at least two orders of magnitude below the useful GC work, and in most cases, three orders of magnitude (this observation is made more accurate below).

Table 5 shows the difference in the average number of global-pool accesses made

Benchmark		Size of LDU				Benchmark		Size of LDU			
		32w	64w	128w	256w			32w	64w	128w	256w
BestPath	max	8.5	7.0	2.2	14.0	Pentomino	max	8.3	13.0	5.3	3.0
	min	−2.7	−4.8	−3.2	−6.5		min	−0.3	−4.0	−2.0	−2.5
Boyer	max	0.5	−1.3	1.5	1.3	Puzzle	max	1.7	0.9	1.1	0.8
	min	−4.3	1.3	−1.5	−0.5		min	−1.9	−2.6	−0.9	−0.2
Cube	max	−2.2	−1.4	−1.3	0.8	SemiGroup	max	1.8	4.8	11.3	−3.4
	min	−1.7	−0.8	−1.6	0.0		min	−4.5	−5.8	−0.2	3.4
Life	max	1.0	3.5	2.5	1.3	TP	max	1.7	1.5	1.6	1.2
	min	0.8	3.5	−3.0	−0.5		min	−1.5	−0.5	−0.5	−0.1
MasterMind	max	1.0	0.9	1.0	1.0	Turtles	max	−9.0	7.0	1.0	−17.0
	min	0.1	0.0	0.0	−0.1		min	−7.0	1.0	7.0	−1.0
maxFlow	max	4.3	2.0	4.0	3.5	Waltz	max	1.5	1.0	1.2	0.2
	min	−4.3	1.5	−2.5	−1.5		min	−1.2	−1.0	−0.2	0.0
Pascal	max	0.0	0.3	0.3	0.3	Zebra	max	46.1	44.9	43.2	47.8
	min	−0.2	−0.2	−0.3	−0.1		min	−8.7	−9.4	−5.7	−6.1

Table 5: Difference in the Average Number of Global-Pool Accesses Between the Max-Workload PE and Min-Workload PE (among 8 PEs, HEU=256 words)

by PEs with the maximum and minimum workloads. For example in Zebra, with LDU = 32, the maximum-workload PE pushed 142 chunks into the pool, and popped 96 chunks, on average (differential is 142−96 = 46). The minimum-workload PE pushed 293 chunks and popped 302 chunks, on average (differential is 293−302 = −9). The general trends in Table 5 show that the global pool functioned effectively to move work from heavily loaded PEs to lightly loaded PEs, i.e., to balance the workload.

The global pool plays two roles. One is for chunk "registration" to avoid losing unscanned regions, and the other is to enable load distribution among PEs. These two roles can be separated by the introduction of a *local pool* for registration, but not distribution (e.g., the JAM Parlog scheduler [5]). The advantage of a local pool is that it retains spatial locality. A disadvantage illustrated in our measurements is that maximum-workload PEs also get unscanned regions from the global pool. If local pools were available, the max-workload PE could conceivably fetch all work locally. However, it is difficult to optimally determine when to contribute to the local pool and when to contribute to the global pool. This is an area for further research.

To estimate the price of load balancing, consider Zebra, the benchmark that accessed the global pool most frequently. The average workload size, per PE, is 20,900 cells referenced (from Table 2). The average number of global-pool accesses, per PE, ranges from 2128/8 = 266 (LDU=32) to 222/8 = 28 (LDU=256) (from Table 4). Thus on

Benchmark	GP Access LDU = 32w/256w	Speedup LDU = 32w/256w
BestPath	9.2	1.12
Boyer	16.4	1.38
Cube	11.0	1.13
Life	9.9	1.12
MasterMind	3.9	1.01
MaxFlow	21.1	1.42
Pascal	1.6	0.96
Pentomino	17.9	1.03
Puzzle	10.6	1.01
SemiGroup	57.5	1.10
TP	9.7	1.07
Turtles	10.5	1.08
Waltz	54.3	2.67
Zebra	9.6	1.00

Table 6: Increases in Global-Pool Access Frequency and GC Speedup, for LDU=32 words *vs.* LDU=256 words (8 PEs, HEU=256 words)

average (for LDU=32), a PE pushes into (and pops from) the global pool once every 20,900/266 = 78 cells referenced. Since one cell reference requires two memory accesses on average, and one global-pool access also requires two memory accesses, this rate is acceptable overhead. Hence our previous estimation of at worst about two orders of magnitude difference is justified.

Table 6 shows the speedup improvement afforded by decreasing LDU size, with respect to the associated increase in global-pool access frequency. Although not entirely correlated, the top three speedup improvements (Waltz, MaxFlow, Boyer) correspond to high frequency increases. Benchmarks showing speedup improvements of 8–13% correspond to moderate frequency increases.[8] The anomaly in these statistics is Pentomino, which does not improve in speedup with successfully increasing pool-access frequency. This might be due merely to the smaller workloads.

5.4 Active Data Characteristics

The active data characteristics of the VPIM architecture offer insights into why the parallel garbage collection algorithm performs as it does. Table 7 shows the frequency of data type for each active cell during the execution of the benchmarks. The object

[8] With the exception of SemiGroup, which has such high (absolute) speedup even for LDU=256, that improvement is limited.

Benchmark	Object Size		Object Type (%)					
	Avg	σ^2	VR†1	LS†2	VT†3	GL†4	MD†5	MS†6
BestPath	4.13	569	11.3	6.4	15.9	44.6	14.3	7.5
Boyer	4.26	4,056	1.8	0.1	69.0	12.6	16.4	0.1
Cube	2.19	34	1.0	86.7	1.4	9.4	1.5	0.0
Life	2.94	87	14.5	32.9	0.3	50.1	2.1	0.1
MasterMind	6.50	3,488	1.3	21.5	6.2	16.8	52.3	1.9
MaxFlow	2.68	2,271	1.2	24.1	17.9	10.6	13.8	32.4
Pascal	2.86	1,326	0.8	65.3	0.8	7.6	23.8	1.7
Pentomino	5.78	4,639	3.3	12.0	17.3	22.2	33.4	11.8
Puzzle	5.60	16,786	0.9	23.2	14.6	8.9	52.0	0.4
SemiGroup	2.10	57	0.7	91.3	3.1	4.0	0.9	0.0
TP	6.60	56,583	0.8	22.3	15.7	6.9	53.9	0.4
Turtles	3.54	172	5.6	32.9	3.2	56.2	2.0	0.1
Waltz	2.56	366	1.1	72.6	1.5	11.6	12.7	0.5
Zebra	5.64	582	0.1	6.9	88.3	0.7	3.9	0.1

†1 **VaRiable** 1 word, represents unbound variable
†2 **LiSt** 2 words, represents list
†3 **VecTor** 1–N words, represents array
†4 **GoaL** 16 or 32 words, holds goal environment with arguments
†5 **MoDule** 1–N words (usually big), program code module
†6 **MiSc** 1–N words, other control, merger records

Table 7: Active Cells Distribution by Type

size given here is *true* size, as opposed to allocated size (see Footnote 4). In general, benchmarks achieving high-performance GC have structures with high average size and low variance. For example, types **GL** and **VT** are large and therefore good for the load distribution because they contain multiple pointers. However, type **MD** significantly affects the variance because the size is outstandingly large compared to other structures. Since copying of one structure is always done by a single PE, too-large structures tend to adversely affect load distribution.

To illustrate these observations, we classify the benchmarks into four groups. The boundaries of these groups are delimited at 3.0 (average) and 1000 (variance), as shown in Figure 8. For each program, the maximum and minimum speedups are listed. In general, GC speedup is influenced more by the variance in object size than by the average object size.

object size	low variance		high variance	
low average	Cube	(7.7–6.8)	Pascal	(3.5–2.7)
	Life	(7.2–6.3)	MaxFlow	(4.1–2.9)
	SemiGroup	(7.8–7.0)		
	Waltz	(4.4–1.6)	(worst group)	
high average	BestPath	(7.2–6.4)	Boyer	(5.8–4.1)
	Turtles	(7.8–7.2)	MasterMind	(2.6–2.5)
	Zebra	(6.4–6.0)	Pentomino	(4.3–3.3)
			Puzzle	(2.8–2.6)
	(best group)		TP	(2.5–2.3)

Table 8: GC Performance Groups, Categorized by Object Size (eight PEs, HEU=256 words, LDU=32 words). (max–min) speedup given for each benchmark.

5.5 Characteristics with Varying HEU

In this section we examine the relationship between the GC algorithm characteristics and the size of the Heap Extension Unit, HEU. Table 9 shows the average speedup (as defined in Section 5.1) with respect to varying HEU and LDU, for a subset of the benchmarks.

We find that speedup is affected not by the size of HEU, but by the size of LDU. If the LDU optimization were not supported, we would get the rightmost speedups in the table (emphasized). Except for Zebra, this emphasized value decreases with increasing HEU. These results imply that the LDU optimization works effectively.

Table 10 shows the average number of updates of B_{global} per GC, with varying HEU. At the bottom of the table is the number of updates needed for initial chunk allocation. In general, increasing HEU reduces the frequency of B_{global} access. Pascal displays the opposite characteristics because of its small workloads (i.e., most updates are done during initialization). These results indicate that the size of HEU should be set proportional to the average workload. Large applications will likely require tuning HEU and LDU, using the measurements presented here for smaller benchmarks as a guideline.

6 Conclusions and Future Work

This article introduced and analyzed the performance characteristics of a parallel copying garbage collector on a shared-memory multiprocessor. The system we examined is a parallel implementation of KL1, a committed-choice logic programming language. The host multiprocessor was a Sequent Symmetry, with our GC experiments limited

Benchmark	Size of HEU	Speedup				
		Size of LDU				
		32w	64w	128w	256w	512w
Boyer	128w	5.93	5.73	**4.80**	—	—
	256w	5.67	5.83	4.38	**4.12**	—
	512w	5.82	5.81	5.88	4.88	**3.90**
MaxFlow	128w	3.12	2.70	**4.23**	—	—
	256w	4.06	3.84	3.70	**2.86**	—
	512w	4.47	2.42	4.11	2.50	**2.18**
Pascal	128w	3.31	2.78	**3.19**	—	—
	256w	2.67	2.91	3.45	**2.77**	—
	512w	3.02	2.93	2.82	2.63	**2.68**
SemiGroup	128w	7.18	7.76	**7.46**	—	—
	256w	7.75	7.28	7.49	**7.02**	—
	512w	7.77	7.61	7.53	6.31	**6.88**
Zebra	128w	6.07	6.66	**6.65**	—	—
	256w	6.27	6.04	6.42	**6.28**	—
	512w	6.44	6.18	6.44	6.49	**6.49**

Table 9: Average Speedup Varying HEU (eight PEs)

Benchmark	Number of Updates of B_{global}		
	Size of HEU		
	128w	256w	512w
Boyer	293	146	121
MaxFlow	262	233	143
Pascal	69	70	73
SemiGroup	2140	1049	600
Zebra	713	428	244
(for initial allocation)	(56)	(64)	(72)

Table 10: Average Number of Updates of B_{global} Varying HEU (LDU=32 words)

to eight of the available processors.

The advantage of the proposed GC algorithm is that all memory accesses, except for marking the old heap and accessing the global pool, are performed without mutual exclusion. This avoids the necessity for costly locking when copying cells. In addition, a load-balancing mechanism is described that is shown to be quite effective in spreading the work among a limited number of PEs. Speedups ranging from 2.5 (MasterMind) to 7.8 (Cube) on eight PEs were achieved by the GC algorithm for the benchmarks studied. Accounting for limitations in ideal speedup, the parallel GC efficiency of these benchmarks ranged from 51% (MaxFlow) to 97% (Cube). The overheads of this load distribution method were shown to be low: Zebra, the program with the most load-distribution traffic, accessed the global pool on average once every 78 cells referenced, an acceptable overhead.

Future areas of research include examining the utility of local pools, and devising overall systems that can avoid copying program modules. One potential solution is to separate program modules onto their own heap, thereby reducing interference with other data types. A more general solution is to apply our algorithm to a generation-type garbage collector (e.g., [10, 13]). Generation-type GC is based on the lifetimes of data, and since program modules have long lives, copying is avoided.

Acknowledgements

A. Imai's research was kindly supported by ICOT Director, Dr. Kazuhiro Fuchi, and first research laboratory chief, Dr. Kazuo Taki. Thanks go to Dr. Atsuhiro Goto of NTT, Mr. Katsuto Nakajima of Mitsubishi Electric Corp., Dr. Keiji Hirata of ICOT, and the VPIM staffs of ICOT and cooperative companies. We also thank Mark Korsloot of the Delft University of Technology, and Dr. Ian Foster of Argonne National Laboratories for help with the benchmarks.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, June 1988. ACM Press.

[2] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[3] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of

Melbourne, MIT Press, May 1987.

[4] J. A. Crammond. A Garbage Collection Algorithm for Shared Memory Parallel Processors. *International Journal of Parallel Programming*, 17(6):497–522, 1988.

[5] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *North American Conference on Logic Programming*, pages 642–657. Austin, MIT Press, October 1990.

[6] A. Goto *et al.* Overview of the Parallel Inference Machine Architecture (PIM). In *International Conference on Fifth Generation Computer Systems*, pages 208–229, Tokyo, November 1988. ICOT.

[7] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159. Seattle, IEEE Computer Society Press, May 1990.

[9] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *International Conference on Logic Programming*, pages 747–762. Jerusalem, MIT Press, June 1990.

[10] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pages 201–204. Pisa, North Holland, May 1988.

[11] K. Nishida, Y. Kimura, A. Matsumoto, and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 83–95. Jerusalem, MIT Press, June 1990.

[12] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1989.

[13] T. Ozawa, A. Hosoi, and A. Hattori. Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pages 291–305. Austin, MIT Press, October 1990.

[14] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.

[15] S. Takagi. A Collection of KL1 Programs — Part I. Technical Memo TM-311, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, May 1987.

[16] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA., 1991.

[17] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA., 1987.

[18] K. Ueda and T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, 33(6):494–500, December 1990.

[19] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.