ICOT Technical Report: TR-653

TR-653

A Shared-Memory Multiprocessor Garbage Collector and its Evaluation for Committed-Choice Logic Programs

by
A. Imai & E. Tick

June, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Shared-Memory Multiprocessor Garbage Collector and its Evaluation for Committed-Choice Logic Programs

Akira Imai
Institute for New Generation Computer Technology (ICOT)*

Evan Tick
Dept. of Computer Science
University of Oregon†

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.

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

*imai@icot.or.jp, 21F Mita Kokusai Bldg, 1-4-28 Mita, Minato-ku Tokyo 108, Japan

†please contact: tick@cs.uoregon.edu, Eugene OR 97403, USA (this paper is 4705 words long).

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 paper introduces a new parallel garbage collection (GC) algorithm for these symbolic languages. Based on Baker's sequential stop-and-copy algorithm [1], 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, specifically the Parallel Inference Machine, PIM [5].

We have chosen KL1 [12], a concurrent logic programming language based on flat GHC [11], 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. Some examples are: incremental garbage collection, language constraints that force all streams to have a single consumer, static mode analysis and abstract interpretation to uncover and reuse single-consumer streams. 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 paper 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 paper 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 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 (scan pointer) and B (copy pointer). We limit our view to shared-memory execution models in this paper. In this section, we develop a parallel algorithm by stepwise refinement. This clarifies both the motivations and mechanisms of the scheme.

2.1 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 [6] and JAM Parlog [3] 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 criterion 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*. A chunk is defined to be a unit of contiguous space, a constant HEU cells in size (HEU ≡ Heap Extension Unit).

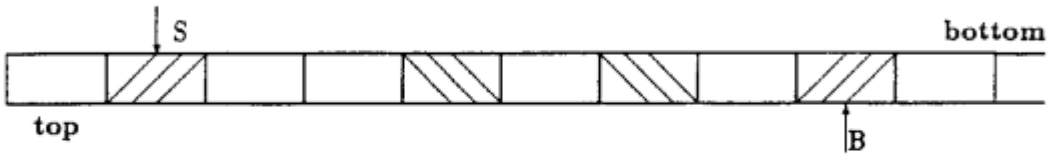
We first consider a simple model, wherein each PE operates on a single heap, managed by a single pair of S and B pointers. Initially, a single empty chunk is allocated for each heap, with the S and B pairs pointing into the *top* (empty) element of the initial chunk. The B_{global} pointer is a state 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 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 1).

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 4.4 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



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 1: Chunk Management in Simple Heap Model

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[1] 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 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. 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.

Referring back to Figure 1, 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.

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

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. The load distribution unit (LDU) is this predefined constant, distinct from HEU,[2] enabling more frequent load balancing during GC. In general, the optimized algorithm incorporates a new rule 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. This action is illustrated in Figure 2.

3 Relationship to Previously-Published Algorithms

We now compare our parallel algorithm to two previously-published garbage collectors for Concert Multilisp [6] and JAM Parlog [3]. Both of these garbage collectors 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.

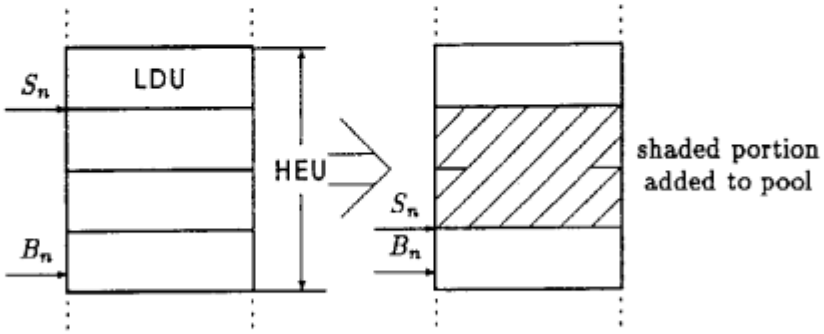
In Halstead's algorithm, remote objects are copied by the PE which first encounters the object, which means that GC changes the "ownership" of an object. On the other hand, in Crammond's algorithm, remote objects remain remote, even after GC. This is implemented by *asking* the owner to copy objects. 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.

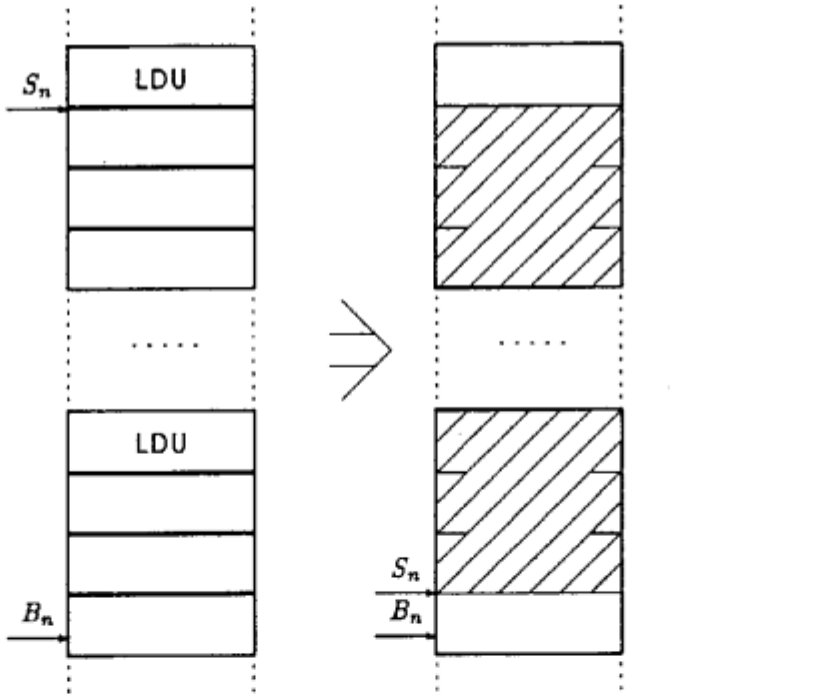
The advantages of our algorithm, over these alternatives, are clarified by the following points.

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

[2] We assume (HEU mod LDU) = 0.



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 2: Chunk Management in Optimized Heap Model

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. 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× (HeapSize / LDU) words.

4 Evaluation

The parallel GC algorithm was evaluated for a large set of medium-sized benchmark programs (from [10] and other sources), executing on VPIM, a parallel KL1 emulator. The measurements presented in this paper 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 where 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 [2]) optimization enabled, facilitated by support from the VPIM system. MRB, a method of incremental garbage collection, gives us a realistic characterization of the

Benchmark	Heap (Kw)	# GCs	#Red. x1000	#Susp. x1000	#Work. x1000	comments
BestPath	192	6	394	57	165	shortest-path problem (30x30 nodes)
Boyer	128	4	529	18	47	toy theorem prover
Cube	128	5	291	6	139	logical constraints (7 cubes) †
Life	128	4	353	236	101	life game simulation (38x38 nodes)
MasterMind	128	8	1525	5	4	game-playing program †
MaxFlow	128	3	80	35	95	max. flow in network (80 nodes,123 links)
Pascal	64	13	285	1	5	Pascal's triangle (row 250)
Pentomino	64	7	188	9	3	2-D puzzle packing (5x5 square,6 pieces) †
Puzzle	128	19	1254	145	17	3-D puzzle packing (7 pieces) †
SemiGroup	448	6	732	12	496	calculation of Brandt semigroup (5 tuples)
TP	64	23	564	47	17	theorem prover (4-Cook's wff)
Turtles	320	1	1178	62	203	logical constraints (12 cards) †
Waltz	128	6	1207	19	32	3-D drawing constraints (38 nodes) †
Zebra	320	9	405	2	167	logical constraints (extended version) †

† all solutions search

Table 1: Summary of the Benchmarks

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), and "work" is the average workload in thousands of cells referenced (discussed in the next section). 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.

4.1 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:

$$\text{workload(PE)} = \text{number of cells copied} + \text{number of cells scanned}$$

$$\text{speedup} = \frac{\sum \text{workloads}}{max(\text{workload of PEs})}$$

The workload value approximates the GC time, which cannot be accurately measured because it is affected by DYNIX scheduling on Symmetry [8]. Workload is measured in units of *cells referenced*.[3] 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 4.3). 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.

Figure 3 summarizes the speedup metrics for the benchmarks. In general, 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.

[3] 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.

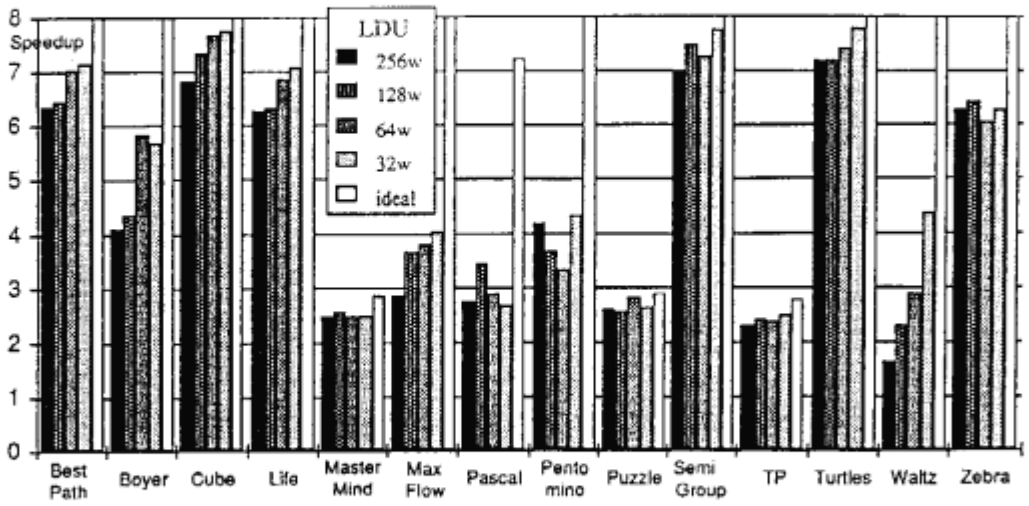


Figure 3: GC Speedup on eight Symmetry PEs, HEU=256 words

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 for distribution.

4.2 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 Figure 4. For $32 \leq$ LDU ≤ 256, we show the range of the ratio of the naive updates to the (smart) updates

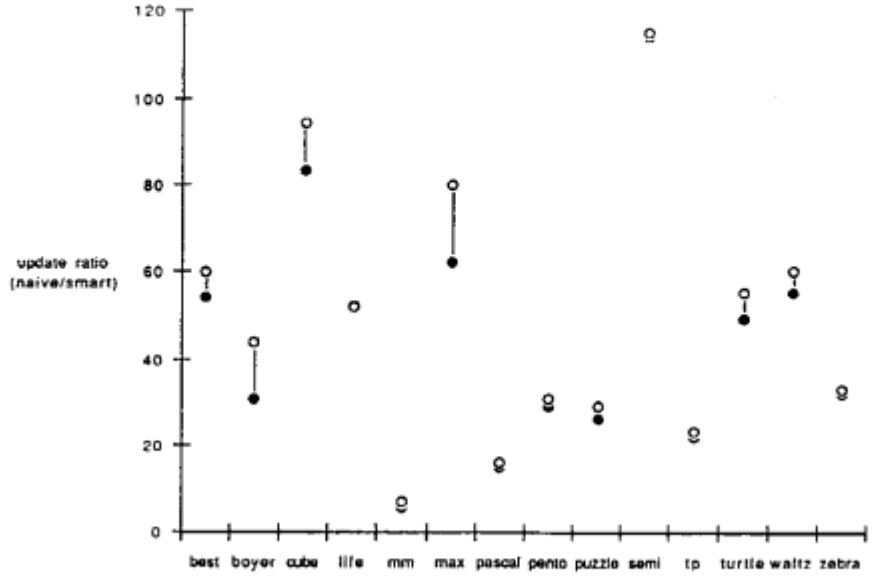


Figure 4: Average Number of Updates of B_{global} over all GCs (8 PEs, HEU=256 words)

made by our algorithm. Note that MasterMind achieved the *least* reduction in update frequency — only a factor of seven, significantly below that of the other benchmarks. This can be explained by the small workloads involved. Initially, log(HEU) 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, with little effect by LDU.

4.3 Global-Pool Access Behavior

For selected benchmarks, figure 5 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 benchmarks selected display the range of observed program behavior. 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).

The global pool plays two roles. One is for chunk "registration" to avoid losing

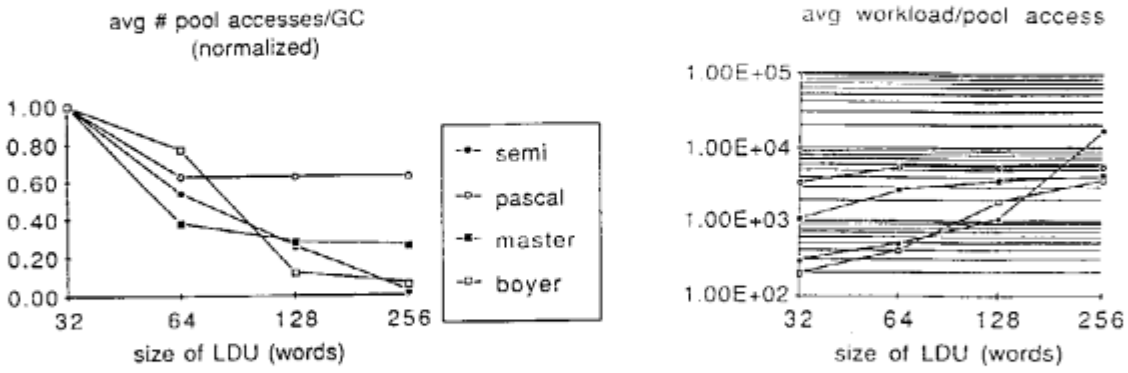


Figure 5: Accesses of the Global Pool (8 PEs, HEU=256 words)

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 [4]). 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 3). The average number of global-pool accesses, per PE, ranges from 2128/8 = 266 (LDU=32) to 222/8 = 28 (LDU=256) (from Table 5). Thus on 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 2 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% corre-

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

Table 2: Relationship Between Global-Pool Access Frequency and GC Speedup, as LDU Increases (eight PEs, HEU=256 words)

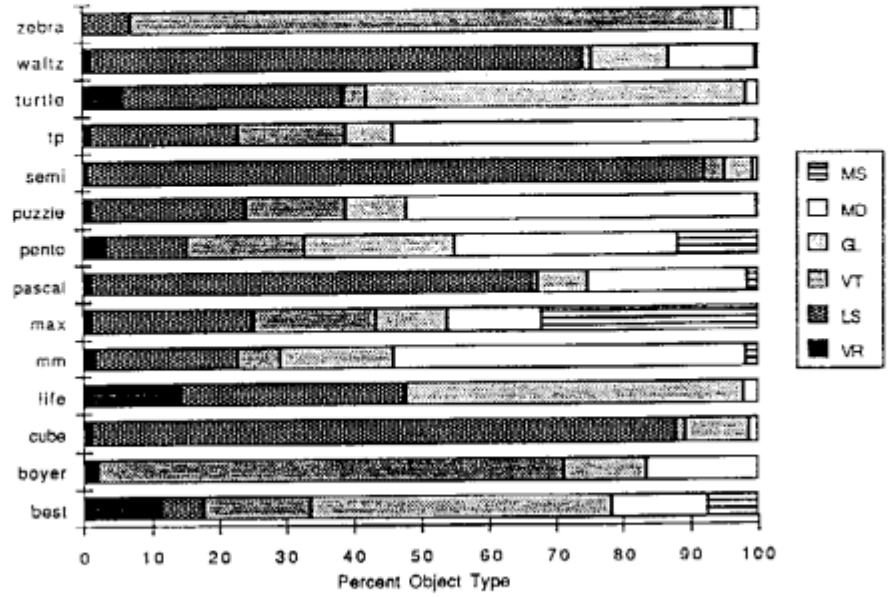
spond to moderate frequency increases.[4] The anomaly in these statistics is Pentomino, which does not improve in speedup with successfully increasing pool-access frequency. This might be due merely to the smaller workloads.

4.4 Active Data Characteristics

The active data characteristics of the VPIM architecture offer insights into why the parallel garbage collection algorithm performs as it does. Figure 6 shows the frequency of data type for each active cell during the execution of the benchmarks. The average object size ranges from 2.1–6.6 cells, with very high variance in some cases. 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 Table 3. 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.

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



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

Figure 6: Active Cells Distribution by Type

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 3: GC Performance Groups, Categorized by Object Size (eight PEs, HEU=256 words, LDU=32 words)

5 Conclusions and Future Work

This paper 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 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. An appropriate extension of this research is to apply our algorithm to a generation-type garbage collector (e.g., [7, 9]). Generation-type GC is based on the lifetimes of data, and its influence on the algorithm presented should be informative.

Acknowledgements

A. Imai's research was supported by ICOT Director, Dr. Kazuhiro Fuchi, and first research laboratory chief, Dr. Kazuo Taki. E. Tick was supported by an NSF Presidential Young Investigator award.

References

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

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

of Melbourne, MIT Press, May 1987.

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] T. Ozawa *et al.* Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pages 291–305. Austin, MIT Press, October 1990.

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

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

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