TR-512

Generation Type Garbage Collection for Parallel Logic Languages

by
T. Ozawa, A. Hosoi & A. Hattori (Fujitsu)

October, 1989

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Generation Type Garbage Collection for Parallel Logic Languages

Toshihiro Ozawa, Akira Hosoi, and Akira Hattori

Fujitsu Limited
1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Abstract

This paper presents a garbage collection (GC) method for parallel logic programming languages. Parallel logic languages require large amounts of data since logic variables can have only one value. Efficient memory management is important for an efficient language processor.

In the parallel logic language Flat Guarded Horn Clauses (FGHC), the amount of live data is always small compared to the total amount of data allocated. These are two kinds of data: short-term and long-term. We concluded that garbage collection using only 2 generations best suits this kind of language.

We call our garbage collection method "2-generation garbage collection". Short-term data is garbage collected back into the 1st generation garbage collection and long-term data is collected into the 2nd generation garbage collection. This method is efficient independent of the ratio of the amount of live data to heap size. When this ratio is high, our method is especially good, reducing the amount of data copied by a factor of 10, compared to simple copying garbage collection.

1. Introduction

This paper presents a garbage collection method for parallel programming languages. We paid special attention to logic languages[1][2][3] because of their semantic clarity and the simplicity of their execution model. We have developed a language system for the parallel logic language FGHC on a shared memory multiprocessor as part of the Japanese Fifth Generation Computer Systems project. This language system is being used for designing a parallel inference machine, PIM/p, whose aim is to achieve the high speed execution of a parallel logic language.

In parallel logic languages, many processes are produced which communicate with each other using shared variables. Many variables must be allocated since each variable can have only one value. Because the order of execution changes dynamically, the lifetime of data can't be

determined beforehand. Efficient, automatic memory reclamation is needed. Using this system, we evaluated the performance and the memory consumption of FGHC and designed a garbage collection (GC) system we call the 2-generation garbage collection. It is a variation of generation type garbage collection optimized for this kind of language. It is efficient at any ratio of the amount of live data to heap size.

2. The Parallel logic programming language FGHC

FGHC is a subset of the parallel logic programming language GHC. An FGHC program is a finite set of guarded Horn clauses consisting of a head, guard goals, a commit operator (|), and body goals. In FGHC, guard goals are restricted to built-in goals in order to achieve higher efficiency.

```
Head :- Guard1,Guard2,... | Body1,Body2,... .
        passive part        active part
```

The combination of the head and guard goals represent the conditions under which the body goals can be executed. During execution of these tests, no assignments to the caller goal arguments can be made. The head and guard goals also play a part in passing arguments. If there are clauses whose conditions are satisfied, the execution of the caller goal is finished, one of them is chosen randomly, and the body goals are executed in parallel. During execution of body goals, assignment to caller goal arguments is allowed. If there is no clause that satisfies its conditions because some arguments of the caller goal are not yet instantiated, execution of the caller goal is suspended until the conditions are satisfied by the execution of other goals. Execution of one goal is called a reduction.

3. An FGHC processor on a shared memory multiprocessor

3.1. Structure

Figure 1 shows the structure of the FGHC processor we developed. The shared memory is divided into three parts. The first part is used for storing the clause definitions and is called the code area. The second part is the heap area, used for allocation of data such as variables, lists,

vectors, atoms, etc. The last is a free area pointed to by a heap allocation pointer, which is incremented when allocating new data. Data is represented by a cell that consists of a value field, a tag field for data typing, a garbage collection field, and a lock field for exclusive access. Data cells are allocated dynamically during execution and this area is reclaimed by garbage collection .

Because the order of execution of FGHC goals is nondeterministic, a goal can't be managed by the stack mechanism normally used in sequential languages. A goal is represented by a goal record[4][5]. Arguments and corresponding clause definitions are referenced by the goal record. These records are stored in the goal area. Many goals are created during execution[6][7], but when execution of a goal is completed, the goal record can be reclaimed. This is done with a free goal list, which is simply a linked list of currently unused goal records.

The free goal list and the heap are common resources. To avoid contention for access to the free goal list and heap allocation pointer, we assign a free goal list and a heap allocation pointer for each processor. That is, individual goal record areas and heap areas are distributed to each processor prior to execution. These common resources are redistributed if they become exhausted in a processor.

When a goal record is created, it is put into a scheduling queue. A processing element (PE) retrieves the goal record from the scheduling queue and tests the conditions of the corresponding clauses against the goal's arguments. If all tests in a clause succeed, its body goals are created

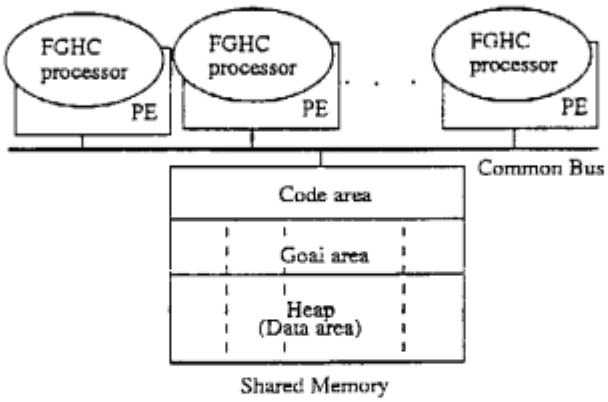


Figure 1. the FGHC Processor

and put into a scheduling queue. If the tests in all clauses fail, the goal fails. If a clause exists with tests that neither succeed nor fail because one or more of the goal's arguments are not yet instantiated, the execution of the goal is suspended. The goal is bound to the uninstantiated arguments to be able to resume when one of them is instantiated. A goal is resumed by re-inserting it into the scheduling queue.

Like free goal lists, scheduling queues are distributed to each PE in order to reduce contention for retrieving goals. Each PE has its own scheduling queue, called a local-queue, which only it can access. For dynamic load balancing, there is an extra scheduling queue, which can be accessed by all PEs. Usually, a PE retrieves from and puts goals into its local-queue. However, if the local-queue becomes too long or another PE's local-queue is empty, the PE takes goals from its local-queue and puts them into the extra queue. A PE whose local-queue becomes empty will try to retrieve a goal from the extra queue. All PEs are informed by a flag in shared memory when one of the PEs has an empty local-queue. This flag is set by a PE when the local-queue and the extra queue are empty. Each PE checks it at the beginning of every goal execution, and resets it if it puts goals into the extra queue.

3.2. Performance evaluation

We have evaluated the FGHC processor to determine the speedup per additional processor used, and to investigate its memory consumption.

We used the following benchmark programs:

1) BUP

This is a bottom up parser which searches all alternatives of a parse tree.

2) DB1

This is an OR parallel meta-interpreter for searching a data base.

3) MAXF

This program finds the maximum flow in a given network; when the flow in each link is restricted.

4) PRIME

This program generates a sequential list of numbers and searches it for primes.

5) QUEEN

This program searches for all solutions to the Eight Queens problem.

Table 1 lists the reduction and data allocation characteristics of these benchmarks.

Table 2 gives the execution time for each benchmark using only one processor and ignoring garbage collection by providing ample memory. Figure 2 shows the relationship between the number of processors and the execution speed when more than one processor is used. We can speed up the system by adding PEs. Idle time, when no goal can be found in the local-queue or extra queue, is very small (about 1% of total execution time) for this load balancing method.

Table 1. Reduction and data allocation characteristics of benchmark programs

Benchmark	BUP	DB1	MAXF	PRIME	QUEEN
Number of reductions	36K	724K	69K	33K	39K
Total allocated data (K words)	73	1700	266	32	40

Table 2. Execution time for 1 processor

Benchmark	BUP	DB1	MAXF	PRIME	QUEEN
Execution time (s)	13.7	198	69.2	10.6	13.3

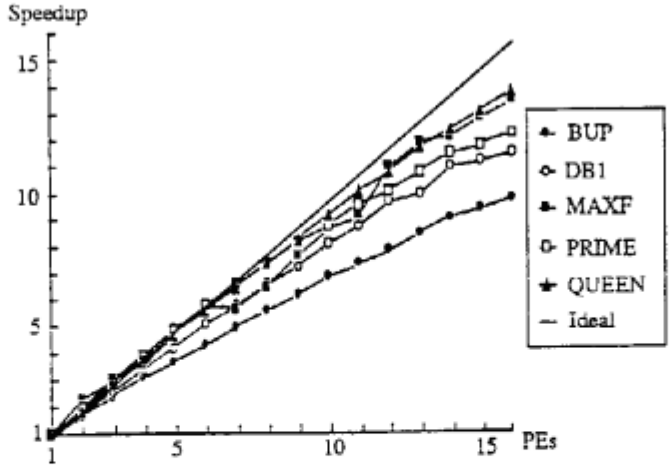


Figure 2. Execution Speed

Table 3. Suspension ratio

Benchmark		BUP	DB1	MAXF	PRIME	QUEEN
Suspension ratio (%)	1PE	1	11	42	0	0
	8PE	38	26	42	4	0
	16PE	35	26	41	12	1

Table 3 shows how the ratio of the number of suspensions to the number of reductions (suspension ratio) varies with the number of processors. In BUP and DB1, the suspension ratio increases dramatically, resulting in a smaller increase in speed when adding processors. This indicates that a significant overhead due to suspensions exists when executing FGHC in parallel. Reducing the number of suspensions is difficult to do using only dynamic load balancing. To achieve a substantial overall reduction, static program analysis is needed.

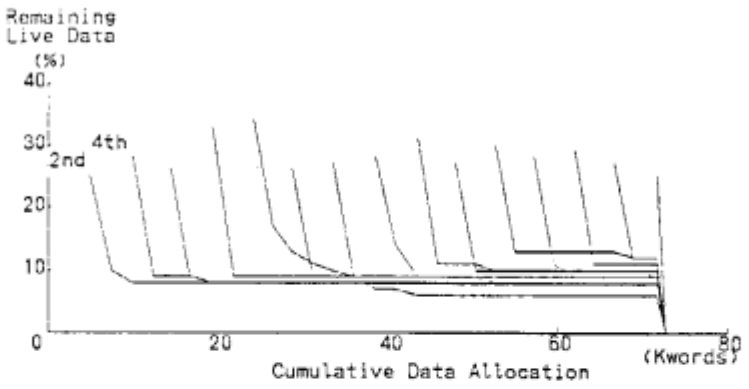
4. Memory consumption

We have evaluated the memory consumption of FGHC using copying garbage collection[8]. When one processor exhausts its heap area, it stops the other processors. After all processors stop, they begin to collect garbage sequentially.

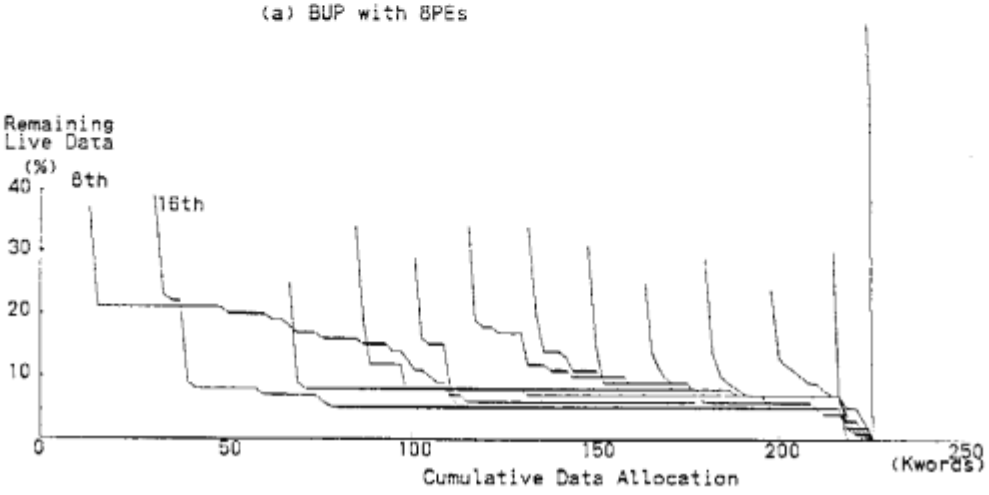
Table 4 shows the total amount of data allocated during execution along with the mean and maximum amount of live data which is active at the time. This was measured by invoking garbage collection after every 2K words of new data were allocated. A lot of data is allocated, but the amount of live data is always small compared with the amount of data allocated. The distribution of live data in the memory space would be quite sparse if data compaction was not done. Compaction is needed to eliminate page faults and achieve high performance.

Table 4. Total allocated data and live data

Benchmark		BUP	DB1	MAXF	PRIME	QUEEN
Total allocated data (K words)		73	1700	266	32	40
Mean live data (K words)	1PE	6.7	11.2	42.6	0.3	0.7
	8PE	7.1	10.8	10.8	0.3	0.8
	16PE	7.0	11.0	11.9	0.3	1.0
Max live data (K words)	1PE	13.6	17.4	78.2	2.0	1.4
	8PE	13.7	18.6	17.0	0.7	1.3
	16PE	13.4	19.0	19.1	0.7	1.5



(a) BUP with 8PEs



(b) MAXF with 8PEs

Figure 3. Lifetime of Data

We have also measured the lifetime of data. Figure 3 gives examples. Each line of the graph shows the percentage of data allocated between the (N-1)th and the Nth garbage collection that remains live as more data is allocated. Most data is thrown away quickly. This characteristic comes from the fact that a large portion of the data is used only to pass information between goals. But, data with lifetimes longer than a certain period is generally alive until the program ends. All benchmarks we tried have the same characteristics. The time is almost the same for all generations in all benchmarks. So, data can be divided clearly into two types. Data with short lifetimes used mainly to pass information between goals, and data with longer lifetimes. This should be true for all parallel logic languages and other parallel languages that create a lot of processes which communicate with each other.

5. Garbage collection for parallel logic languages

We can use this lifetime characteristic to choose a suitable type of garbage collection for parallel logic languages. A generation type garbage collection[9][10][11][12] would suit them well. This type of garbage collection compacts data and exploits the lifetime characteristics to reclaim garbage efficiently. Our observations show that two generations are enough. The 1st generation is for claiming data with short lifetimes. The 2nd generation is for storing data with long lifetimes. Each generation has its own heap area. More than two generations would be redundant since there are only two types of data. The memory configuration is shown in Figure 4. In each generation, the heap area is divided into two regions, the new space and the old space. Each PE has a GC stack in which it records pointers in the 2nd generation which point to data in the 1st generation. When such a pointer is created during execution of a goal, its address is put into a GC stack by the PE that created it.

Data allocation and generation garbage collection are done as follows.

1) The new space of the 1st generation is distributed evenly among all PEs prior to execution. Each PE has its own heap allocation pointer to avoid contention for data allocation. Each PE allocates new data from its 1st generation new space.

2) When a PE exhausts its 1st generation new space, it sets a flag in shared memory to notify the other PEs. All PEs check this flag before executing a goal. If the flag is set, execution stops until all other PEs stop. When all PEs have stopped, the PE that originally set the flag, switches the roles of the new and the old space. This is called a flip. All PEs then begin garbage collection together.

3) 1st generation garbage collection scavenges only the 1st generation, so only the 1st generation

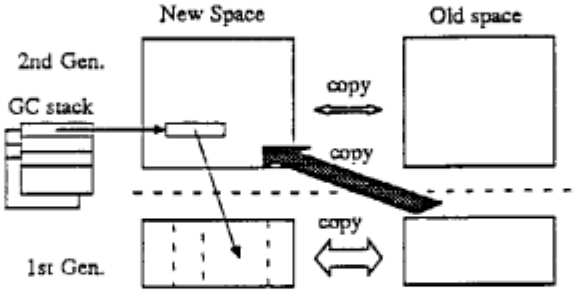


Figure 4. Structure of Two Generation GC(1)

is flipped. All live data in the 1st generation can be traversed, starting from the goals in the scheduling queues, GC stacks, and the registers of the PEs. Each PE performs garbage collection locally at the same time by following these pointers in its scheduling queue, GC stack, and registers to find live first generation data and copy them. The data in the 2nd generation is not traversed at all. It then follows the pointers of goals in the extra queue which have not yet been followed by any PEs. If any of this data has been alive for a certain amount of time, t, it is copied into the new space of the 2nd generation. If not, it is copied into the new space of the 1st generation. If data copied to the 2nd generation points to data in the 1st generation, the address of the 2nd generation data is put into the GC stack. This threshold time, t, is the lifetime of short term data. The only exception to this occurs if the new space of the 2nd generation is exhausted during garbage collection. In this case, all data which would have gone there is copied to the new space of the 1st generation.

4) The PE that originally exhausted its 1st generation space checks the 2nd generation to see if it is almost full when it performs the flip. "Almost" full is a tunable parameter. If so, then 2nd generation garbage collection is done instead of the 1st generation garbage collection. In 2nd generation garbage collection, both the 1st and 2nd generation are flipped, and all live data is copied to the appropriate new space. All live data in the 1st or 2nd generation can be traversed from the scheduling queues and the PE registers. The GC stack is redundant in this case. Each PE traverses the pointers contained in the scheduling queue goals and in the PE registers, copying data into the new space in the same way as for the 1st generation garbage collection. Since the 1st and 2nd generation are being collected together, the 2nd generation filling up could actually result in data from the 2nd generation being copied into the 1st generation. But this would only occur shortly before running out of memory, so it does not affect normal operations.

The threshold time, t, is the minimum time a piece of data is guaranteed to be in the 1st generation. The passage of time is measured by the amount of data allocated. One way to do this is to attach a generation field to each datum indicating how many times it has survived a garbage collection. The amount of data allocated between each garbage collection is also recorded. With this information we can determine a lower boundary for how much data has been allocated since the creation of the data in question. If the value of the generation field is N, then the lower bound is the amount of data allocated since the Nth previous garbage collection. This

value is compared to t to determine whether a piece of data is short or long term. Short-term data is garbage collected back into the 1st generation and long-term data is collected into the 2nd generation .

Another way to measure the passage of time is to divide the 1st generation memory into 3 areas as in Figure 5. Each area is equal to t. New data is allocated from the new space, and data which has not yet encountered a garbage collection is copied into the old space. The intermediate space is used for storing data which has survived one garbage collection. Garbage collection is invoked when the new space is exhausted. To collect garbage, a flip is done and the name of the old and intermediate spaces are changed, the new space is not changed. The live data in the new space, which has not yet encountered a garbage collection, is copied into the old space and the live data in the intermediate space, which has survived one garbage collection, is copied into the 2nd generation new space. Since the new space is empty immediately after garbage collection, and the new space is the same size as t, invocation of garbage collection marks the passage of t. The live data copied into the 2nd generation has been in the 1st generation for at least t. This method requires more memory, but does not require a generation field in the data.

To execute a garbage collection in parallel, each PE traverses the live data from the goals in its local queue and GC stack (if it is a 1st generation garbage collection). When a PE is finished processing all its own live data, it chooses an untraversed goals in the extra queue and tries to

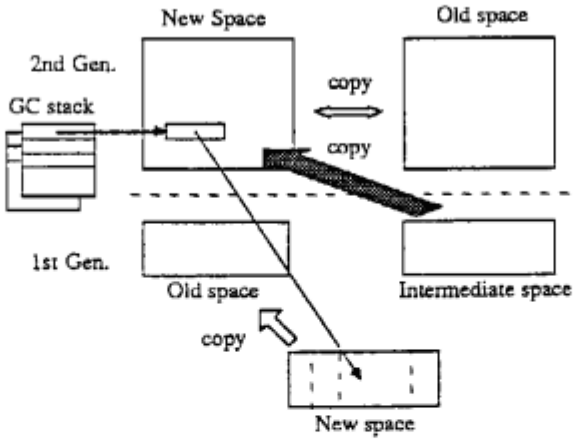


Figure 5. Structure of Two Generation GC(2)

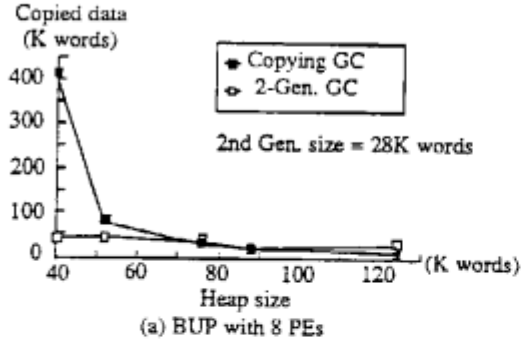
traverse the live data attached to it. Thus garbage collection and goal execution all use the extra queue for load balancing.

6. Evaluation of 2-generation garbage collection

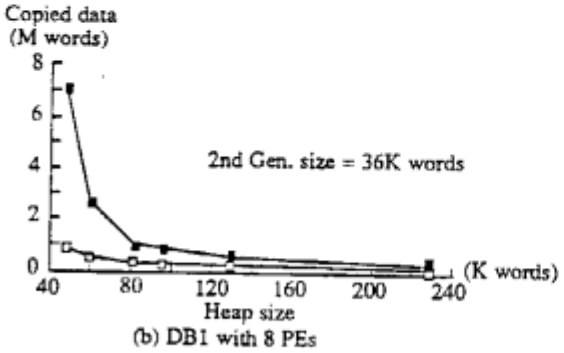
We compared our 2-generation garbage collection with a simple copying garbage collection. The heap size is the same for each application. In a simple copying garbage collection, it is divided equally between the new and old spaces. In our 2-generation garbage collection, the 2nd generation is twice the size of the maximum amount of live data observed in our tests (Table 4). This is divided equally among the new and old spaces. Whatever remains in the heap is allocated evenly between the three areas of the 1st generation (Figure 5).

Figure 6 compares the total amount of data copied by each of the two methods. To evaluate the performance, we measured the amount of data copied instead of the garbage collection time because we cannot measure the 2-generation garbage collection time exactly. The 1st generation garbage collection is less than 100 ms. Since the total garbage collection time is proportional to the total amount of data copied, we can use this measure to estimate the efficiency of a garbage collection method. This also eliminates effects due to parallelism. Figure 7 shows how the garbage collections performed. All tests were run using 8 PEs. In 2-generation garbage collection, execution of goals is about 5 % slower than in simple copying garbage collection because 2-generation garbage collection must record pointers from the 2nd generation to the 1st generation.

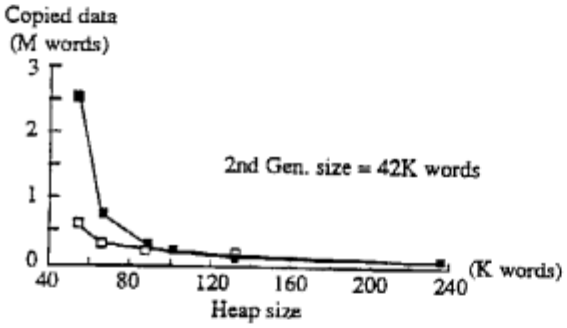
When the heap is large, the efficiency of simple copying garbage collection is about the same as that of 2-generation garbage collection because there are only a few garbage collections. When the heap is small, there are a lot of garbage collections, and the efficiency of 2-generation garbage collection is about 10 times that of simple copying garbage collection. In simple copying garbage collection, the amount of data copied depends on the size of the heap, but in 2nd generation garbage collection it is independent. The shorter time required for each garbage collection and the compactness of live data within the memory space are other advantages of generation type garbage collection. We can conclude that 2-generation garbage collection is an efficient method independent of the ratio of the amount of live data to heap size.



(a) BUP with 8 PEs

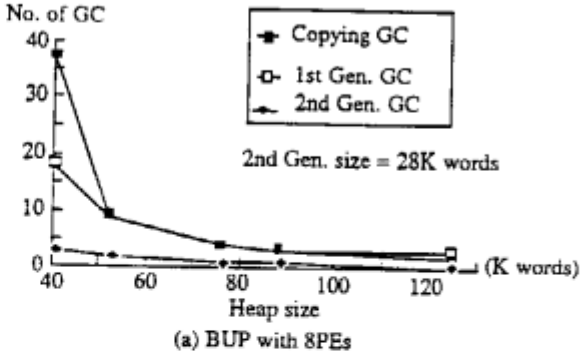


(b) DB1 with 8 PEs

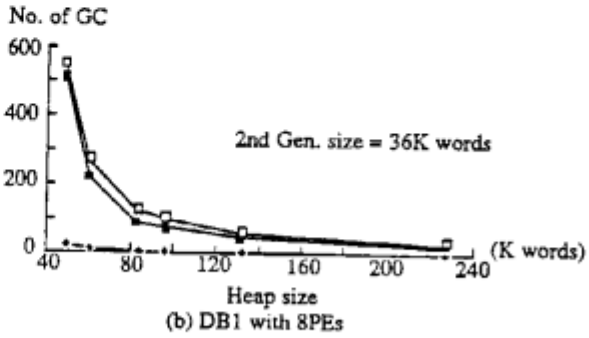


(c) MAXF with 8 PEs

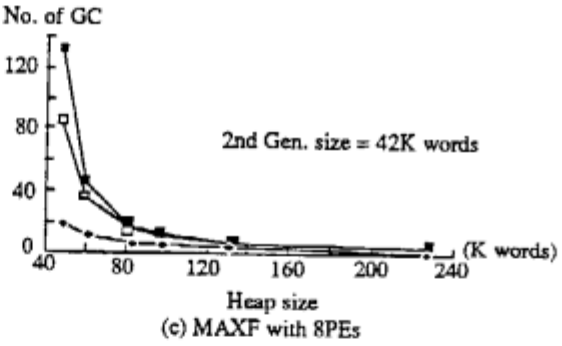
Figure 6. Performance of GCs



(a) BUP with 8PEs



(b) DB1 with 8PEs



(c) MAXF with 8PEs

Figure 7. Number of GCs

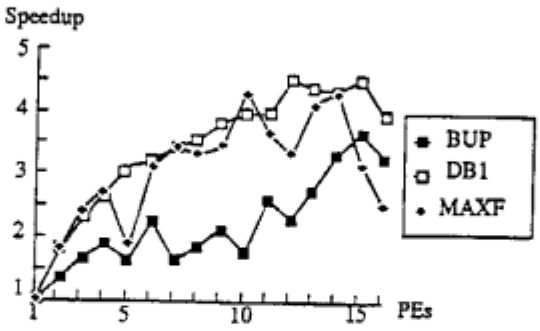


Figure 8. GC Speedup

During any parallel garbage collection, one of the PEs will copy the maximum amount of data, which is some percentage of the total amount of data copied. One whole garbage collection time is proportional to this percentage. From the average of this, we can calculate the increase in speed that results from using parallelism in garbage collection. Figure 8 shows the relationship between the number of processors and garbage collection speed. The speed increase of garbage collection saturates at about 4 times, even if we use more than 4 PEs. This is because the amount of data traversed from different goals varies greatly. If there is one goal from which a lot of data can be traversed in the extra queue, garbage collection time is dominated by it. To improve the performance of parallel garbage collection, the unit size of execution must be made smaller. This is where our research will focus next.

7. Conclusions

We evaluated the memory consumption of the parallel logic language FGHC. The amount of live data is always smaller than the total amount of data allocated. There are two kind of data: short-term and long-term. They are clearly distinguishable. Since these results come from the fact that a lot of processes are created and a large amount of data is consumed in passing information, these result should be valid for other similar parallel languages, especially logic languages.

The generation type garbage collection seems to suit this kind of language well. Using only 2 generations, we constructed an efficient garbage collection method which is independent of the ratio of the amount of live data to the size of the heap.

Acknowledgements

We would like to thank Manager Hiromu Hayashi and General Manager Junichi Tanahashi for their encouragement. We also thank our colleagues at Fujitsu and the members of the fourth research laboratory at ICOT for their discussions, and especially Mark Feldman for helpful suggestions after reading draft version of this paper.

References

[1] K. Ueda, *'Guarded Horn Clauses'*, Technical Report TR-103,ICOT,1985.

[2] K. L. Clark and S. Gregory, *'Parlog: Parallel programming in logic'*, ACM Trans. On Prog. Lang. and Syst., 1986.

[3] E. Shapiro, *'Concurrent Prolog: A Progress Report'*, IEEE Computer, August 1986.

[4] Y. Kimura and T. Chikayama, *'An Abstract KL1 Machine and Its Instruction Set'*, Proceedings of 1987 Symposium on Logic Programming, 1987.

[5] N. Ichiyoshi, T. Miyazaki, and K. Taki, *'A Distributed Implementation of Flat GHC on the Multi-PSI'*, Proceeding of the Fourth International Conference on Logic Programming, 1987.

[6] M. Sato and A. Goto, *'Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor'*, IFIP WG 10.3 Working Conference on Parallel Processing in Pisa, Italy, 1988.

[7] M. Sugie, M. Yoneyama, and A. Goto, *'Analysis of Parallel Inference Machines to Achieve Dynamic Load Balancing'*, Proceeding of International Workshop Artificial Intelligence for Industrial Applications, 1988.

[8] H. G. Backer, *'List Processing in Real Time on a Serial Computer'*, Comm. ACM, Vol. 21, No. 4, pp. 280-294, 1978.

[9] H. Lieberman and C. Hewitt, *'A Real-Time Garbage Collector Based on the Lifetimes of Objects'*, Comm. of the ACM, Vol. 26, No.6, 1983.

[10] S. Ballard and S. Shirron, *'The Design and Implementation of VAX/Smalltalk-80'*, Smalltalk-80: Bits of History, Words of Advice, G. Krasner(editor), Addison Wesley, pp. 127-150, 1983.

[11] D. Ungar, *'Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm'*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 157-167, 1984.

[12] K. Nakajima, *'Piling GC --- Efficient Garbage Collection for AI Languages'*, Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing, pp. 201-204, 1988.