TR-409

An Evaluation of FGHC on a Shared Memory Multiprocessor

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

July, 1988

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

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Evaluation of FGHC on a Shared Memory Multiprocessor

Toshihiro OZAWA Akira HOSOI Akira HATTORI

FUJITSU LIMITED

Kawasaki Japan

Abstract

We are currently developing a FGHC language processor on a multiprocessor with shared memory as one of activities of Japanese fifth generation computer systems project. FGHC is a subset of the parallel logic programming language GHC. The main problems which occur in development of a parallel system such as this are managing the contention for common resources, and realizing an efficient method for load balancing. In this system we distribute common resources to each processor prior to execution and support a mechanism which redistributes them when one processor exhausts one of them. To achieve efficient load balancing, each processor has a local scheduling queue in addition to a common scheduling queue shared among all processors. We evaluated the characteristics of memory consumption of FGHC. Most data is discarded within a very short time after it is created. But data with a life time longer than a certain period is generally alive until the end of execution. We have therefore concluded that a generation scavenging method of garbage collection suits FGHC well.

1. Introduction

This paper stems from research in parallel programming languages in which we have paied special attention to parallel logic languages because of their semantical clearness. This is one of activities of the Japanese fifth generation computer systems project. We are currently developing a set of emulators as an FGHC language processor system on a shared memory multi processor (Symmetry CPU: 80386). Our system is a set of emulators. One emulator is assigned to each processor and shares a common memory area to execute one task together. Each emulator emulates KL1B code[1], which is the intermediate code generated by compiling FGHC source code.

2. FGHC

2.1. Syntax

FGHC is a subset of the parallel logic programming language GHC[2] which doesn't require to support of multiple environments, and is therefore expected to execute efficiently. A sentence of GHC is called a clause and consists of a head, guard goals, commit operator (|), and body goals. The combination of the head and guard goals is called the passive part and the combination of the body goals is called the active part. In FGHC, guard goals are restricted built-in goals. The passive part represents the conditions under which the body goals can be executed.

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

2.2. Execution

To execute a goal in GHC, a clause is chosen from the set of clauses whose head can be unifyed without any assignment to the goal arguments, and whose guard goals succeed. The body goals of selected clause are then executed in parallel. Because no assignment to any goal arguments is allowed, the passive part of all clauses can be tested in the same environment.

In FGHC, guard goals are restricted built-in goals which do not need a separate environment to be solved. So the execution of FGHC's passive part is not burdensome.

3. Implementation of FGHC on a multiprocessor

3.1. Basic control structure

An FGHC goal is represented by a data structure called a goal-record[3]. Arguments and corresponding clause definitions can be accessed from it. When a goal-record is created, it is put into a scheduling queue.

An emulator gets a goal-record from a scheduling queue and tests the passive part of corresponding clauses, using the goal's arguments. In our emulator, the tests are done one by one. If the passive part of a clause succeed, the

body goals of that clause are created and put into a scheduling queue. If the passive part of all clauses fail, the execution of the goal fails. If a clause exists whose passive part neither succeeds nor fails, because some argument of the goal are yet not instantiated, the execution of the goal is suspended. This goal is bound to those uninstantiated arguments in order to be able to resume it when one of them is instantiated. The resumption of a goal is realized by re-inserting the goal into a scheduling queue.

Goal-records are thrown away quickly, and many goal-records are created during execution. This is managed efficiently through a free-goal-list.

3.2. Basic data structures

Besides a value field, data has a tag field for data typing, a gc field, a lock field to the exclusive access. The supported data types are atom, list, vector, integer and floating point. There are two types of uninstantiated cells. One indicates that no goal is bound. This cell's tag is UNDEF. Another indicates that goals are bound. This cell's tag is HOOK.

Data is allocated from a common heap area which is used by all processors.

4. Resource contention

The free-goal-list and the heap are common resources. If we have an only one free-goal-list or only one heap

allocation pointer, contentions will undoubtedly occur frequently.

To avoid this contention, we assign a free-goal-list and a heap allocation pointer to each processor. That is, goal-record areas and heap areas are distributed to each processor prior to execution.

A mechanism is supported which redistributes these common resources if they become exhausted in a processor. To realize this mechanism, our system has a special register, REQUEST-FLAG, that is accessible from all processors. If there is an operation which all processors must do together or one processor needs to stop the execution of all the other processors, this register is set with the corresponding operation code. Each processor checks the REQUEST-FLAG at the beginning of every reduction cycle.

For example, when one processor exhausts his own free-goal-list, he sets REQUEST-FLAG to 'REQ_GOAL'. The other processors will stop execution after finding the REQUEST-FLAG set. After all processors stop, the processor which set a flag collects free goal-record and redistributes them. If one processor exhausts the heap area allocated to it previously, garbage collection is invoked.

Because common resources are distributed prior to execution and there exists a mechanism for redistributing them, exclusive access to these common resources is not necessary. Their exhaustion is not regular event. Because they are reused in the way of free list management, this reduces the

overhead of the parallel execution.

5. Dynamic load balancing

Using the same method as was used for the free-goal-list, distributing a scheduling queue is done to reduce the contention for getting a goal-record. But, if the scheduling queue is divided between each processor, we need a mechanism that distributes the goal-records very quickly in order to reduce the idle time of processors[4,5]. It seems to be very common occurrence that a processor becomes idle because there are no goal-records to be solved. If there are almost the same number of goals to be solved as the number of processors, the overhead of idle time in a bad load balancing method becomes greater than the overhead of contention for accessing one scheduling queue.

In order to reduce both the idle time and the overhead of contention for accessing, there is one more scheduling queue besides the one local to each processor has been introduced. This queue is called the extra-queue. We also introduced the register, EXTRA-LENGTH, which keeps the length of the extra-queue. This register and extra-queue are accessible from every processor.

Goal migration for load balancing is done by way of the extra-queue. Goals in a processor's local queue is manipulated by only the owner processor of that queue.

If the value of EXTRA-LENGTH is shorter than

(his own queue length) * CONSTANT

then the processor gets goals from his local queue and puts them in the extra-queue until the above condition is again satisfied. Because the extra-queue is truly shared by each processor, exclusive access to the extra-queue is necessary. But exclusive access to a local scheduling queue is not necessary, because it is accessed by only it's owner processor.

According to this strategy, when there are plenty of goals, each processor's scheduling queue has goals and the contention is eliminated. When there are not enough goals, only extra-queue has goals and idle time is reduced.

Table 1 shows the relation between CONSTANT, and contentions and idle time. We can reduce the overhead by tuning the value of CONSTANT. At the moment, FGHC is being executed on emulator, but if we execute FGHC on a special hardware, the execution speed will be very high. In such environment, reducing the overhead of the contention will be even more important than it is now.

But, we can not control the number of suspensions. Table 2 shows the relation between CONSTANT and the number of suspensions. The number of suspensions is not related with CONSTANT in general.

CONSTANT	1	4	8	16	32
Contention time	120	190	260	380	400
Idle time	3190	2300	2280	2200	2200
Total overhead time	3310	2490	2540	2580	2600

The unit is Milli-Second.
The number of processors is 8.
Program is MAXFLOW.

Table 1 The overhead of contentions and idle

CONSTANT	1	4	8	16	32
The number of suspensions	38000	32000	37000	33000	31000

Program is MAXFLOW.

Table 2 The number of suspensions

6. Performance evaluation

Figure 1 shows the relationship between the number of processors and the execution speed. CONSTANT is set to 4. From results of the Eight Queens program (which doesn't suspend), the common bus seemed to be saturated. One reason why bus traffic is high and speedup becomes dull according to an increase to the number of processors is because our emulator's registers are actually in memory. Table 3 shows the percentage of time spent performing various types of operations necessary to execute each program. Context switch is the operation that is getting a goal-record and setting the registers. Idle is idle time because no goal

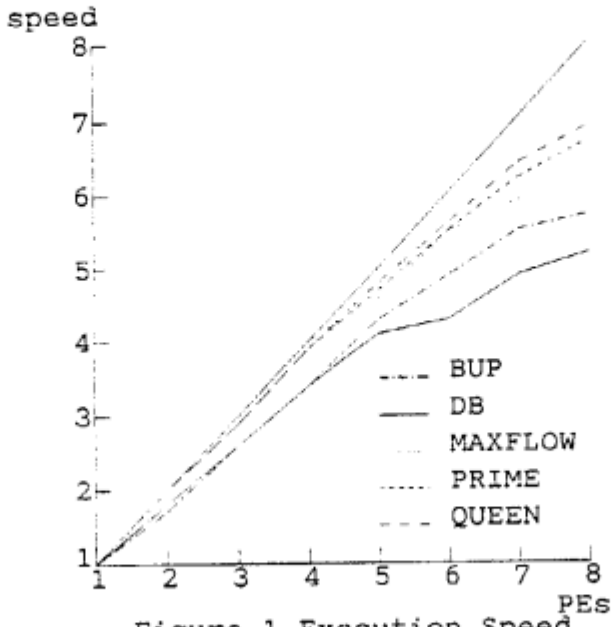

Figure 1 Execution Speed

can be found in his local queue and the extra-queue. Passive part is the operation that is execution of a passive part. Active is the operation that is solving built-in goals in a active part. Goal creation is the operation that is creating goals and putting them into queues. Suspension is the operation that is suspension and resumption of a goal. Table 4 shows how long each operation takes for single operation. These indicate that idle time is very small in this load balancing method. The cost of one suspension is rather larger and the total cost dominants the overhead of parallel execution.

But, we can not control the number of suspensions. Suspensions can be reduced with single queue scheduling in

Operation	BUP	DB	MAXFLOW	PRIME	QUEEN
Context switch	16	12	10	8	13
Idle	1	1	2	1	1
Passive part	21	24	47	44	41
Active	32	39	26	45	22
Goal creation	20	14	4	0	20
Suspension	8	7	10	2	0
Others	2	3	1	0	3

The number of processors is 8.

Table 3 The percentage of time spent in various types of operation

Operation	BUP	DB	MAXFLOW	PRIME	QUEEN
Context switch	150	105	120	120	150
Passive part	165	180	720	210	210
Active	195	270	270	225	120
Goal creation	360	270	345	240	360
Suspension	405	255	345	195	

The unit is Micron Second.
The number of processors is 8.

Table 4 The time spent performing various types of operation

some benchmark programs, but suspension is still the biggest overhead for parallel execution of FGHC. Reducing the number of suspensions appears to be very difficult using only dynamic load balancing. To achieve a substantial reduction in general, static analysis of program is necessary.

7. Memory consumption

We have evaluated the characteristics of memory consumption of FGHC, using garbage collection (GC) called copying GC. When one processor exhausts his heap area, it sets the REQUEST-FLAG to stop the other processors. After all processors stop, the processors begin to collect garbage in their scheduling queues.

To estimate the life time, each data is assigned a generation according to when it was born. Data from Nth generation indicates that it was born between the end of (N-1)th garbage collection and the beginning of Nth garbage collection. Figure 2, 3 show the life time of data. Each line of the graph indicates the percentage of remaining live data of a certain generation. Each GC is invoked after about 3K words of new data are allocated. Most of all data is thrown away very shortly. These characteristics come from the fact that many goals are thrown away quickly and most goals have input arguments which are not shared by other goals. But, the data whose life time is longer than a certain time, is generally alive until the end of execution. That time is almost same for all generations. Most of these data are pointed to by suspended goals. We conclude that the generation scavenging type GC suits FGHC and is also important in reducing the suspension from point of view of memory consumption. If the creation of the goal which will be suspended for a long time can be postponed, much more

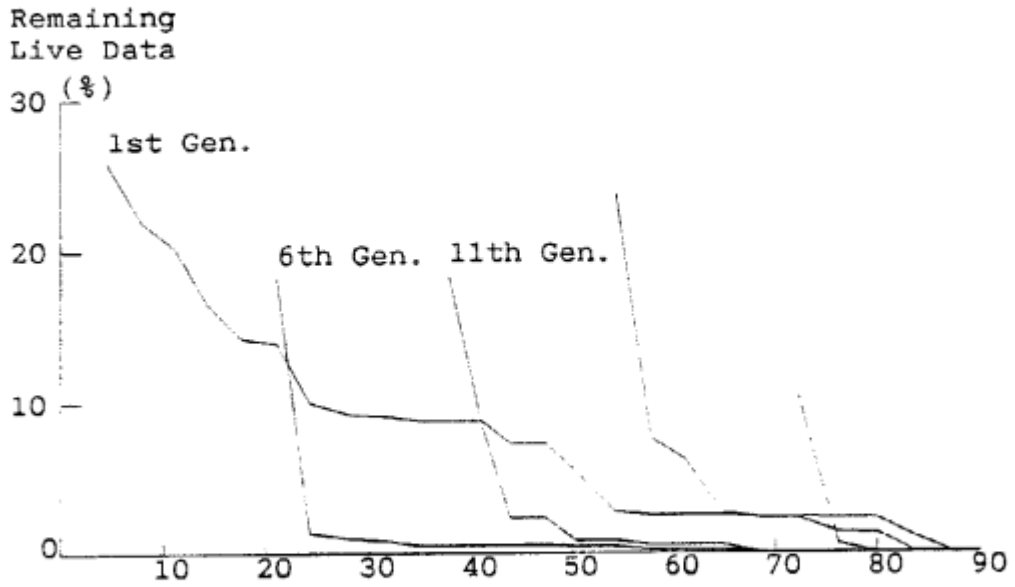

Figure 2 Life Time of Data (BUP)

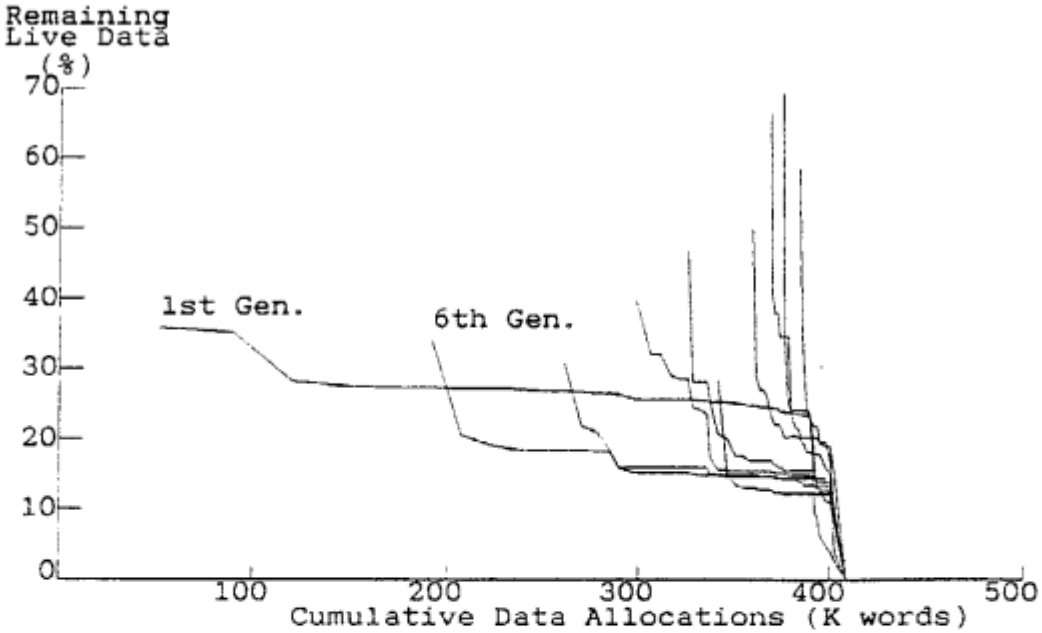

Figure 3 Life Time of Data (MAXFLOW)

garbage can be collected.

8. Conclusion

Distributing common resources prior to execution and redistributing them when the resources of one processor becomes exhausted reduce the overhead of the contention of accessing common resources.

We can support an efficient dynamic load balancing method, using one global queue and processor's own queue.

The method of generation scavenging for garbage collection suits FGHC well. Reducing the number of suspensions is important not only for speed up but also for memory consumption.

Acknowledgements

The authors would like to thank Manager Hayashi and General Manager Tanahashi for their encouragement in out research. We also thank our colleagues and the members of the fourth research laboratory at ICOT for their discussions.

References

[1] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction of Flat GHC on the Multi-PSI. In Proceedings of the Fourth International Conference on Logic Programming, 1987.

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

[3] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In Proceeding of the Fourth International Conference on Logic Programming, 1987.

[4] M. SATO, 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

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