TR-250

KL1 Execution Model for PIM Cluster with Shared Memory

by

M. SATO, H. SHIMIZU, A. MATSUMOTO
K. ROKUSAWA and A. GOTO

April, 1987

KL1 Execution Model for PIM Cluster with Shared Memory

Masatoshi SATO Hajime SHIMIZU Akira MATSUMOTO
Kazuaki ROKUSAWA Atsuhiro GOTO

ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 JAPAN

Abstract

The parallel inference machine (PIM) is now being developed. The target language of PIM is KL1, a parallel logic programming language based on GHC. PIM consists of a dozen or more clusters. The PIM cluster is a tightly coupled multiprocessor with shared memory. This shared memory architecture possesses not only the advantage of low communication overhead among processors, but also disadvantages, such as access path bottleneck to shared data. This paper describes the effective KL1 parallel execution mechanism, introducing the various kinds of locality in the data structure allocation, scheduling, and load balancing to overcome these disadvantages. Evaluation results are shown using a software simulator. As a result, this KL1 parallel execution model can make the most of the advantages of the shared memory.

1 Introduction

ICOT is conducting research and development of the parallel inference machine, *PIM* [3]. The PIM target language is the parallel logic programming language, KL1, based on GHC [7].

Reducing the communication cost between processors is one of the most important factors in parallel processor architecture. From the viewpoint of the communication cost, the PIM hardware construction has a locality in processor connection. The PIM consists of a dozen or more clusters. Each cluster is a tightly coupled of eight or ten processors with shared memory. These clusters are connected by networks, as a loosely coupled multiprocessor. Therefore, the communication cost within the cluster is lower than that between the different clusters.

This paper describes the parallel execution model within the PIM cluster. Parallel execution of KL1 for the loosely coupled multiprocessor is being studied in the ICOT multi-PSI project [2] [6]. This will be also applied as the intercluster parallel execution in the PIM.

In the design of a parallel execution model, the key issue is how to enhance locality even in tightly coupled multiprocessors with shared memory. This is because the problems of data access bottleneck and data access synchronization with lock/unlock in the shared memory architecture must be solved. These problems can be avoided if there is enough locality in the data accesses by processors.

Section 2 briefly introduces the abstract execution model of KL1 with major data structures. Section 3 discusses how to enhance the locality in parallel execution model using shared memory. Section 4 gives the implementation features of this model. Finally, the primary evaluation results of parallel execution are shown as collected from a software simulator.

2 KL1 Execution Overview

2.1 Brief Introduction to KL1

KL1 is a parallel logic programming language based on GHC [7]. A KL1 program is a finite set of guarded Horn clauses of the following form:

$$H :- G_1, \cdots, G_m | B_1, \cdots, B_n. \ (m \geq 0, n \geq 0)$$

where H, G_i, and B_i are called *the clause head, guard goals,* and *body goals.* | is called a commitment operator. The part of a clause preceding | is called the passive-part (or guard), and that following it is called the active-part (or body). A guarded clause with no head is a goal clause, as in Prolog.

KL1 requires sufficient language functions to be used as a system programming language of PIM, and must also be efficiently implemented on PIM. We added several language functions to, and put restrictions on GHC with the design of KL1 in mind.

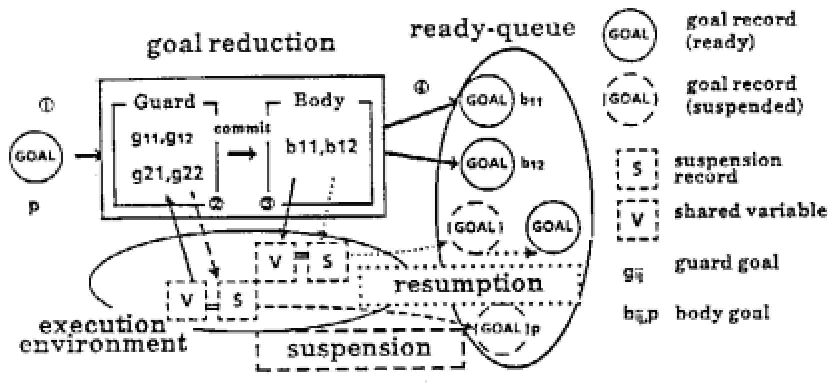

Figure 1: Abstract Features of KL1 Goal Reduction

Major additions to GHC are metalogical functions and *pragma*. Metalogical functions treat logical values of goals as arguments, so that programmers can describe system software such as operating systems. *Pragma* enables the programmers to give hints for dynamic control of processing load in the PIM. Generally, the programmer knows the expected load characteristics of programs and, it is difficult and costly for the system to balance the load fully automatically. *Pragma* will be discussed in a later section.

KL1 clauses are restricted to the flat clauses, i.e., each guard can contain only built-in goals. This restriction makes it easy to execute head unification and guard evaluation, and allows sequential search for candidate clauses.

2.2 Goal Reduction Feature in KL1

Figure 1 shows the abstract features of KL1 goal reduction.

The following data structures are used in KL1 goal reduction (Table 1). Parallel goals are represented by *goal records* and their execution environments of variable cells. Each goal record has one of two states, *ready* or *suspended*. The ready goal records represent the currently reducible goals, and they are arranged in a *ready queue*. The suspended goal record is associated with the uninstantiated variable cell by a *suspension record*. This is used for synchronization between parallel goals through the shared variables.

The goal records form a *goal tree* with *metacall records* to manage their logical results (success/failure). The goal tree consists of metacall records as

Table 1: Data Structures

Goal record	Status of a goal
	Link to goal records
	Arguments list
	Pointer to code
	Pointer to metacall record
Metacall record	Status of a metacall
	Link to its brother metarecord
	Link to its parent metarecord
	Counter of children goals
	Pointer to a result variable
Suspension record	Link to a next suspension record
	Link to a goal record
	Link to an OR-waiting flag

nodes and goal records as leaves. The count of these leaves is managed by the children counter in the belonging metacall record. This goal tree is used to encapsulate the logical value of goals within a part of a subtree, so that the whole system can avoid failure if the subtree fails.

From the implementation viewpoint, clauses in KL1 programs are compiled into WAM [8] like machine codes, called KL1-B [4]. Then the processor performs goal reduction by executing the corresponding KL1-B codes. Details of KL1-B are not discussed here. However, the abstract features of goal reduction and the corresponding KL1-B codes are shown briefly through a *scheduler* process and a *reducer* process in the following section.

The role of the *scheduler* is to select a reducible goal and to invoke the *reducer*. Various strategies can be used for this reducible goal selection. However, it is assumed here that the *scheduler* simply dequeues a goal record from the ready queue and passes it to the *reducer*.

The *reducer* performs a goal reduction through the following three stages, *guard test*, *body unification*, and *body goal fork*.

(1) Guard test

In the *guard test*, each candidate clause is tested sequentially by head unification and guard evaluation to choose one clause whose body goals will be executed. Assuming that a candidate clause is:

$$p(X,Y,Z) \quad :- \quad X = a \mid \cdots.$$

The *guard test* is represented by `wait` code as:

Guard test: `wait(X,atom(a),`*Next Clause*`).`

The code of `wait` can be described as:

```
wait(X,atom(a),Label):    read X and check the equality
                                between the value X and atom 'a'.
                          if Success then goto next code
                          elseif the caller variable X is uninstantiated
                             then stack X on Stack and jump to Label
                             else jump to Label
```

When the *reducer* finishes all guard tests without choosing any clause, the *reducer* executes the **suspend** code, and then checks whether the stack is empty or not. If the stack is empty, the *reducer* knows that this goal reduction fails. If not empty, this goal must wait till one of those variables in the stack will be instantiated. Therefore, the **suspend** code provides each link between the variables and the goal records in order to activate the suspended goal immediately after the variables are instantiated.

(2) Body unifications

After one clause is chosen by the *guard test*, the *reducer* executes the unifications in the selected clause body. Such *body unification* is represented by the `get` code as:

$$p(X,Y,Z) :- \cdots \mid Y = a,\ Z = b, \cdots.$$

Body unification:
```
get(Y,atom(a)).
get(Z,atom(b)).
...
```

Then the `get` code is described as:

```
get(Y,atom(a)):     read the value Y.
                    case Y of
                      instantiated :
                        check the equality between the value Y
                        and atom 'a'
                      uninstantiated : write 'a' in Y
                      uninstantiated with link to suspension records :
                        resume the suspended records
```

When a `get` code instantiates the variable with suspension record in this body unification, the *reducer* finds the suspended goal and enqueues it to the ready queue.

(3) Body goal fork

If the selected clause body includes any user goals such as:

$$p(X,Y,Z) :- \cdots \mid \cdots\ q(Y,Z)\ \cdots,$$

the *reducer* creates the new goal records and enqueues them into the ready queue as new reducible goals and updates the children counter of the parent metacall record. At this point, these new goal records are linked to the metacall record where the reduced goal was linked in order to form a goal tree.

These operations are represented by `set` and `enqueue` codes as:

```
Body goal fork:   ...
                  set(Y,new goal record for q).
                  set(Z,new goal record for q).
                  enqueue(new goal record for q).
                  ...
```

When one of the body fork goal can execute recursively, the code `execute` is used instead of `enqueue`. At this execution, it is not necessary to update the children counter. If there is no fork goal, this goal reduction is terminated with the `proceed` code, and the *reducer* decreases the children counter.

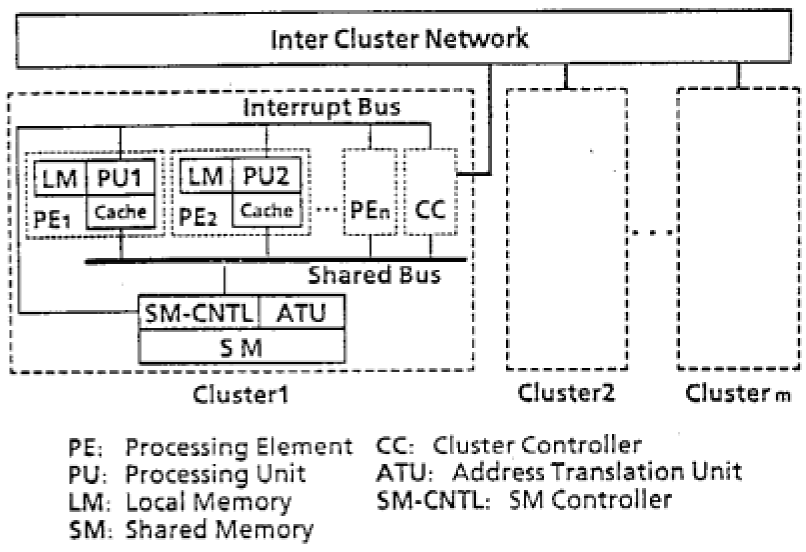

Figure 2: Configuration of PIM

3 Extension to Parallel Reduction

3.1 Consideration of PIM Cluster Architecture

Before considering the extension to parallel reduction, this section discusses the features of the target machine, PIM. Figure 2 shows the overall configuration of PIM that we are currently designing. As described before, PIM has a hierarchical structure with cluster concepts.

Each cluster consists of several processors (PEs) and a shared memory (SM), and each PE has a local memory (LM) and cache [1] for the shared memory. Each PE can also interrupt other PEs and communicate by passing short messages.

This section focuses on parallel goal reduction by a multiprocessor with shared memory, where each processor performs goal reduction in parallel communicating through shared variables in the shared memory.

The major advantage of using shared memory is the reduction in communication overhead among processors compared with message-oriented loosely coupled multiprocessors. Using exclusive data access by lock/unlock, the goal reduction, discussed in Section 2, can be extended to a parallel mechanism. However, such a simple extension is not enough for realizing the high performance PIM. This is because parallel execution on the shared memory machine architecture also needs to solve the following problems.

- Exclusive data access mechanism, such as lock/unlock operations.
- Overhead in exclusive access to shared data.
- Access path bottleneck.

3.2 Locality

First, the concept of locality is defined for both software and hardware.

On the software side, application programs for PIM consist of various sized modules (parallel processing components). It seems natural that such modules are represented as goals in KL1 programs. Then some KL1 goals, followed by many children goals or recursively invoked goals, correspond to large modules (or large subgoal trees). In other words, a KL1 goal can be considered both as a large module and as a component of modules. Some goals tightly couple with each other and some goals do not. The degree of this combining strength is referred to as *locality of goals.*

On the architecture side, we are now designing a coherent cache mechanism to enable quick access to SM from each PE. The coherent cache mechanism generally depends on two kinds of memory access locality. One is time-dependent access locality as in conventional computers. The other is interprocessor locality. In other words, it is suitable for the coherent cache if there are less overlapped memory areas among each processor's execution. LM is also available to use the locality. The local data structures which do not have to be shared can be stored in LM. These are called *localities in architecture.*

Application of locality of goals to the localities in architecture makes it possible to overcome the above problems. Therefore, the following three strategies are introduced into the parallel execution model. They are:

- data structure allocation,
- depth-first scheduling,
- load balancing when using program locality.

These strategies can reduce the communication traffic between parallel processes (or processors), and also can reduce accesses to the shared memory. Therefore, this model can make the most of the advantages of the shared memory. The following subsections describe these strategies.

Table 2: Map of Data Structures

Data	Local/Shared
Ready queue	Local
Goal records	Local
Environments	Shared
Suspension records	Shared
Metacall records	Shared
Code	Shared

3.3 Data Structure Allocation

Local data structures are separated from data structures, discussed in Section 2.2, according to their data access characteristic (Table 2). Local data structures can be handled by each processor individually, so that such data can be stored in LM and accessed in parallel without lock/unlock operations.

First, the ready queue and the parallel goal records are chosen as local data structures for the following reasons. In KL1 goal reduction, the ready queue is accessed when the *reducer* selects and dequeues a goal in *scheduling,* or when it enqueues forked goals in *body goal fork.* Levy gave dequeue and enqueue algorithms to one ready queue shared among processors [5]. However, these accesses occur so often that the shared ready queue may cause an access bottleneck. Therefore, we adopt individual *scheduling* by the individual ready queue for each processor.

We introduce the following distribution method by which each processor can access goal records independently. Each processor dequeues a goal from its own ready queue, and enqueues most of the new forked goals to it, except for the following two cases. One is when it becomes necessary to make each processor load in balance. In this case, goal records are distributed among the processors. The other is when a suspended goal is resumed by other processors. However, the number of times a goal is sent can be decreased by adopting the distribution method. (The detailed distribution method will be described later.) Therefore, a goal record area is allocated in each LM.

In addition to this separation, we introduce individual memory management for the shared data structures. In case of *environments,* each processor has its own environments for allocating new variable cells. Other records, even in the shared data structure, are managed by individual free lists in each processor. As a result, most of the overhead can be avoided by the exclusive data access in the memory management of the shared data areas.

3.4 Depth-first Scheduling

Considering a goal reduction, each processor can be expected to fetch the goal record into the internal registers and accesses the goal environment. Then the processor may fork new goals from the input goal. In this case, these new goals often inherit the part of the input goal. For example, the goal $append(X, Y, Z)$, which matches the following recursive clause, forks a new goal $append(X1, Y, Z1)$. If the new goal $append(X1, Y, Z1)$ is scheduled continuously, the *reducer* can reuse the goal record and the environment of the input goal $append(X, Y, Z)$.

$$append(X, Y, Z) :- X = [U|X1] \mid Z = [U|Z1], append(X1, Y, Z1).$$

In addition to this, such recursive scheduling enhances the memory access locality when the local cache memories are provided for each PE in the PIM cluster. In general, depth-first scheduling is more suitable for enhancing such a locality than breadth-first scheduling.

3.5 Load Balancing when Using Program Locality

To make full use of the processing power in the PIM cluster, the parallel goals should be distributed not only to keep the processor loads in good balance but also to reduce the amount of wasteful communication between processors.

To realize this distribution, we adopt *on-demand distribution* and use the designator to select the split goal which will grow to a large subgoal tree. This designator is called the *pragma*. *On-demand distribution* and *pragma* make it possible to suppress wasteful goal send/receive operations. In *on-demand distribution*, goals are distributed only when the receiver processor has none goals to reduce. In *pragma*, processors can easily select the distribution goal which has heavier reduction loads than communications overhead. This distribution is called *On-Demand with Pragma (ODP) distribution*.

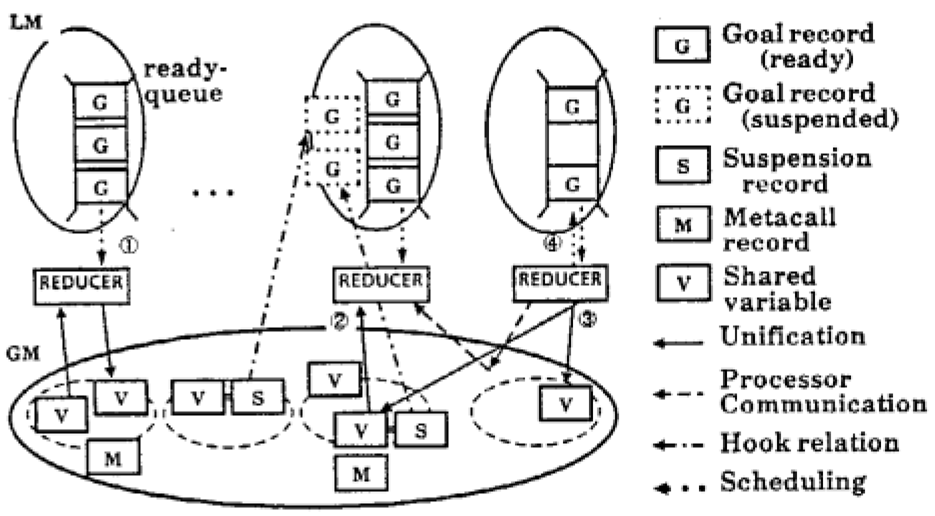

Figure 3: Structure of execution model

4 KL1 Parallel Execution Model

4.1 Overview of this Model

In this section, KL1 parallel execution is shown as a *scheduler* and three-stage *reducer* process as in Section 2. Figure 3 shows the overall structure.

The *scheduler* on each processor has its own ready queue. It dequeues one goal record and invokes the *reducer*. Here, each *scheduler* can access its own ready queue without lock/unlock.

(1) Guard Test

`Wait` test operations, even for parallel goal reductions, do not need exclusive access for the shared data, because the call variables are only read in `wait` operations. Then the `wait` code is almost the same as described in Section 2.

However, when this goal is suspended, the *reducer* must obtain exclusive access to the shared variables in the `suspend` code execution. Before `suspend` code execution, the stacked variables have not been locked, so another goal may have instantiated the stacked variables. Therefore, the `suspend` code can be described as:

```
suspend:   create the OR waiting flag.
           reread the stacked variable with lock.
           if Instantiated
               then unlock it and retry this goal at first clause.
               else create a suspension record,
                   set the pointer of OR waiting flag,
                   link the suspension record and unlock it
           repeat reread and link for stacked variables.
```

(2) Body Unifications

In the body unifications using the `get` code, exclusive access to the call variables is necessary if the variable has not been instantiated. The `get` code is represented as:

```
get(Y,atom(a)):    read the value Y with lock.
                   case Y of
                     instantiated :
                       unlock and check the equality
                         between the value Y and atom 'a'
                     uninstantiated : write 'a' in Y and unlock
                     uninstantiated with link to suspension records :
                       resume the suspended records
```

If the shared variables have been linked with suspension records, it is necessary to resume the suspended goals. A more detailed representation of this resumption is:

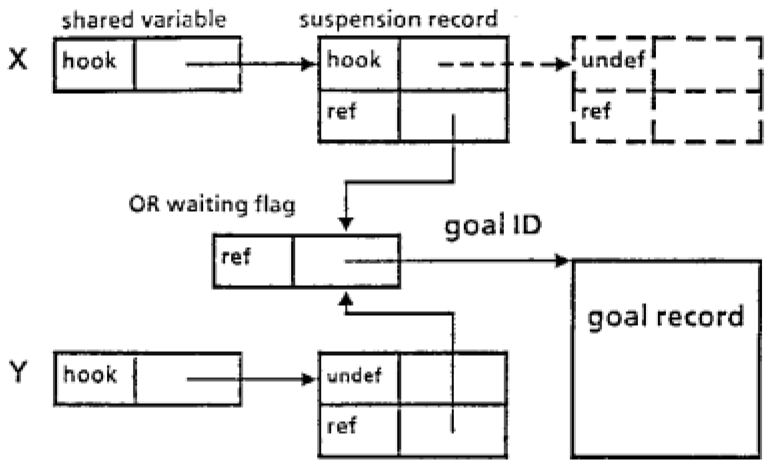

Figure 4: Structure of suspended goal

resumption:

```
save the pointer to the suspension record,
write the instantiate value with unlock,
check the OR waiting flag with lock,
if goal ID then
  write NULL with unlock
  and check to which PE the goal belongs,
  if self PE then enqueue into self ready queue
    else send the enqueue message to the belonging PE
elseif NULL then do nothing
repeat check and enqueue for linked suspension record.
```

Since the goal records are allocated in LM, this **resumption** may require processor communications if the suspended goal belongs to another processor.

Additionally, this **resumption** requires synchronization between OR waiting clauses if the suspended goal has been waiting for one of variables. As described above, the OR waiting flag is used for this synchronization. Figure 4 shows the structure of a suspended goal using an OR waiting flag. In this case, the goal is waiting for one of two variables, X and Y, which shares an OR waiting flag. The content of the OR waiting flag shows whether the other variable has already been instantiated or not. The goal ID, which indicates the goal address of LM, shows that the other variable has not been instantiated. NULL shows that the other variable has already been instantiated.

(3) Body Goal Fork

The `set` code sets the goal's arguments at the new goal records in LM or creates new variables for the goal's arguments in the shared variable areas. Even for the creation of shared variables, the `set` code does not need lock operations because the shared variable areas are divided for each processor as described in Section 3.2. Thus the `set` code is also the same as described in Section 2.

4.2 ODP Distribution

Distributed goals are restricted by *pragma* as described in Section 3.5. Goals with *pragma* are distinguished in the goal fork operation, `enqueue`. The goal fork operations with *pragma* are represented by `p-enqueue` codes instead of `enqueue` as follows. Here, @ is the notation for *pragma*. Processors can recognize that the goal, r, is the candidate goal to be distributed.

$$p\ \ :-\ \ \cdots \mid p, q, @r, \cdots.$$

```
Body goal fork:   ...
                  enqueue(new goal record for q).
                  ...
                  p-enqueue(new goal record for r).
                  ...
                  execute(goal p).
```

These goals with *pragma* are distributed from busy processors to idle processors in an *on-demand* manner. Here, two-level goal requests from the idle processor, the *weak request* and the *strong request*, are introduced to realize effective goal distribution. When the processor becomes idle, this idle processor first sends a request to a busy processor by setting the request flag in the shared memory *(weak request)*, and waits for distributed goals for a short period. If the idle processor does not receive goals after the weak request, this idle processor sends the interrupt message for goal distribution to the busy processor *(strong request)*.

Figure 5 shows this goal distribution feature. Each processor has two ready queues, the high priority queue and the low priority queue. The processor enqueues the fork goal without *pragma* into the high priority queue by the enqueue code. When the fork goal with *pragma* is created, the p-enqueue code first tests the request flag.

If the request flag is off, the fork goal with *pragma* is enqueued in the low priority queue. In the former example, body goal p is scheduled recursively, body goal q is enqueued in the high priority queue, and the body goal r is enqueued in the low priority queue if there is no request. The processor can execute the goal which is in the low priority queue if there is no goal in the high priority queue. In fact, most goals with *pragma* are executed by the processor which created the goal.

If the request flag is on, the processor creates the goal record directly to the message buffer and sends the message to the idle processor. This check is not costly because the cache memory can usually keep the request flag, and the cache miss-hit occurs only when the request flag is set. Then the p-enqueue code can be represented as:

```
p-enqueue(goal record):    check the request flag
                           if Requested
                               then create its goal record
                                    as the message and send it.
                               else enqueue the goal record
                                    into the low priority queue.
```

Sometimes the busy processor does not execute p-enqueue codes after the idle processor has set the request flag. In such cases, the idle processor sends the interrupt message for goal distribution to the busy processor. Then the busy processor which has received the interrupt message searches its own low priority queue and sends the distributed goals. This request scheme is more expensive than the first one described above.

5 Evaluation

In Section 3, the three strategies, which can reduce the overhead caused by sharing data structures, were introduced into the parallel execution model. Section 4 showed that the strategies of *data structure allocation* and of *depth-first scheduling* are very efficient both for reducing the amount of lock/unlock operations and for enhancing the data access locality. However, to use inde-

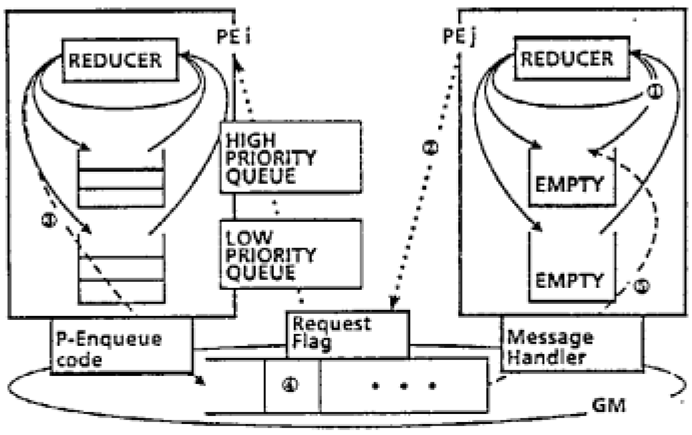

Figure 5: Goal distribution

pendent ready queues on each processor makes it difficult to balance the load in the cluster. Therefore, we introduced the load balancing strategy, *ODP* distribution.

To evaluate the load balancing and localization effect in this model, we define the following metrics and have developed a software simulator on the VAX/11.

- **Idle percentage:** The percentage of idle time to execution time in each PE. The unit time is assumed as one reduction in this simulator.
- **Distribution ratio:** The ratio of the total number of goals to be sent to the other processors versus the given processor execution goals.
- **Reference ratio:** The ratio of the reference amounts to shared data which are allocated in the shared memory by other processors versus the reference amounts to shared data allocated by the given processor itself. If this value can be kept low, the interaction among processors can also be kept low.

Data on memory access patterns, goal distribution, and execution time are preliminary. The evaluation programs are the 8-queen program and the Bottom Up Parser (BUP) program. The 8-queen program searches for the positions of queens on an 8 × 8 chess board so that no queen captures other queens. Because this program has a balanced search tree, it is easy to balance

Table 3: Effectiveness of localization

	8-queen		BUP	
	Random	*ODP*	Random	*ODP*
Idle percentage	3.2	0.4	13.4	6.6
Distribution ratio	0.566	0.001	0.053	0.031
Reference ratio (passive part)	1.991	0.023	0.827	0.322
Reference ratio (active part)	0.612	0.019	0.238	0.138

the load for the system. The BUP program generates parsing trees by analyzing natural language sentences. This search tree is unbalanced, so it is rather difficult to handle the load balance for the system.

Table 3 shows the simulation results, the average data for the four processors. Here, the *random distribution*, in which the goals with *pragma* are distributed randomly, is used as a comparison with *ODP distribution*. This *random distribution* also uses the *pragma* to restrict the distributing goals and to enhance the locality of data access. As shown in Table 3, this *ODP distribution* can keep not only the distribution ratio, but also the idle percentage, low. It also keeps the reference ratio low enough to make full use of the advantages of shared memory.

6 Conclusion and Further Work

This paper described the parallel execution model in PIM cluster and evaluated the effectiveness of localization. By using the locality concepts, the parallel execution model was able to reduce not only the communication traffic between parallel processes (or processors) but also the accesses to shared memory. It also described how to realize the goal distribution.

We have developed the software simulator of this model on the sequential machine and evaluated it as discussed above. We are implementing the detailed parallel emulator on an actual multiprocessor, Balance 21000, and we are planning to evaluate this model on this parallel emulator.

Acknowledgment

The research and development described in this article is being conducted mainly by the members of the PIM, multi-PSI and KL1 groups both in the ICOT Research Center and the participating companies. We also wish to thank the ICOT Director, Dr. Kazuhiro Fuchi, and Dr. Shun-ichi Uchida for valuable suggestions and guidance.

References

[1] A. Goto et al. *Parallel Cache and Hardware Lock Mechanism for PIM Cluster.* TR 247, ICOT, 1987.

[2] N. Ichiyoshi et al. A Distributed Implementation of Flat GHC on the Multi-PSI. In *The proceedings of the Fourth International Conference on Logic Programming*, May 1987.

[3] A. Goto and S. Uchida. *Toward a High Performance Parallel Inference Machine -The Intermediate Stage Plan of PIM-.* TR 201, ICOT, 1986.

[4] Y. Kimura and T. Chikayama. *An Abstract KL1 Machine and its Instruction Set.* TR 246, ICOT, 1987.

[5] Jacob Levy. Shared Memory Execution of Committed-choice Languages. In *Lecture Notes in Computer Science : Third International Conference on Logic Programming*, July 1986.

[6] K. Taki. The Parallel Software Research and Development Tool : Multi-PSI system. In *France-Japan Artificial Intelligence and Computer Science Symposium 86*, October 1986.

[7] K. Ueda. *Guarded Horn Clauses.* TR 103, ICOT, 1985.

[8] David H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, Artificial Intelligence Center, SRI, 1983.