TR- 906

Paralell Inference System Reseach in the Jaoanese FGCS Project

T.Chikayama & K.Rokusawa

February, 1995

Parallel Inference System Research in the Japanese FGCS Project

Takashi Chikayama and Kazuaki Rokusawa

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

Abstract

The FGCS project is a national project of Japan, that aims at establishing the basic technology required for high performance knowledge information processing systems. The research and development principle throughout the project has been to adopt *logic* as the theoretical backbone of knowledge information processing and *parallel processing* as the key technology for obtaining high performance.

This paper reports an outline of the results obtained in the parallel inference system research in the FGCS project, and describes its implementation scheme of the concurrent logic programming language KL1.

1 Introduction

The FGCS project is a national project of Japan, that aims at establishing the basic technology required for high performance knowledge information processing systems. The research and development principle throughout the project has been to adopt *logic* as the theoretical backbone of knowledge information processing and *parallel processing* as the key technology for obtaining high performance.

Thus, one of the most important subprojects of the FGCS project has been research and development of the parallel inference system, aiming at establishing both hardware and software technologies for massive symbolic computation power through highly parallel processing.

There are two most important sets of technologies for high-performance knowledge processing systems. One is technologies providing problem-solving methods for knowledge information processing; the other is technologies for actually applying such methods, thereby providing massive computational power and ease in programming. The parallel inference system subproject aims to establish the latter, both in hardware and software, through logic-based parallel processing.

Several problems that did not exist with sequential processing arise with parallel processing. The most typical ones are the following.

Programming language. Traditionally, parallel processing software has been written in sequential programming languages augmented with features for parallel processing. With this approach, parallel languages can be designed by modifying already existing and well-established sequential languages, and parallel software can be built by "slightly" modifying existing software.

However, the modification supposed to be "slight" and superficial often turned out to be large and fundamental. In this approach, parallel programs are structured as a set of sequential processes, occasionally communicating one another. One of the most frequently encountered problems with this organization is synchronization failures between processes. Synchronization failures are frequent source of bugs that are hard to fix.

Another problem frequently observed after settling the synchronization problems was that parallelized programs just do not show the expected efficiency. To solve the problems, assignment of computation among processes has to be changed drastically, that often requires considerable program revision. This rewriting may again introduce new synchronization problems.

The above approach made a wrong start. Since parallel processing is so different from sequential processing in many aspects, a "slight modification" will not work, especially for complicated problems in knowledge information processing.

To solve the problem, we adopted a concurrent logic programming language KL1 [1], which was based on GHC [2].

Software development environment. Tools originally designed for sequential programming do not always provide the functionality required for debugging highly parallel software, even with extensions made afterwards. The same can be said about the operating system features, such as interfaces to the resource management mechanism and virtualized I/O devices. The original design relies so much on sequential processing that most of the extensions for parallel processing are only suited for small-scale parallelism.

An original operating system, PIMOS (Parallel Inference Machine Operating System) [3] was thus developed to provide a comfortable software development environment for parallel application software.

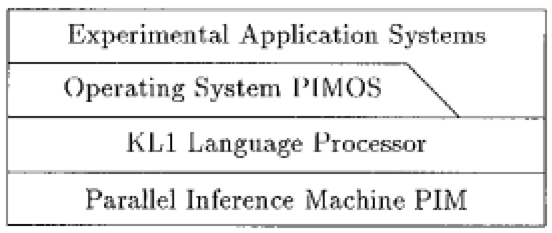

Experimental Application Systems
Operating System PIMOS
KL1 Language Processor
Parallel Inference Machine PIM

Figure 1: Parallel inference system.

The parallel inference system has an overall structure as shown in figure 1. A notable difference with conventional computer systems is that the operating system is built upon the level of the programming language processor. For efficient execution, highly parallel application software has to control parallel processing activities in the system; this control usually was not needed in sequential or small-scale parallel systems. The application layer and the operating system layer thus require the same primitives. We decided to provide a common basis for them, namely, the programming language KL1.

2 Architectural Platforms

When the development of the parallel inference system began in 1986, no hardware systems in the market seemed to fit our purposes. Parallel systems available in the market were either SIMD processors or had too few processors. Thus, we decided to design our own parallel processing hardware for the system.

It was not clear which parallel processing architecture was most suited to knowledge information processing. It was quite difficult to evaluate many architectural ideas through desk-top analysis only. Software simulation was too time-consuming to evaluate using application software with practical complexity and size.

We decided to develop several (five, to be precise) experimental models of parallel inference machines (PIM [4]) with different processor and interprocessor connection architectures and to evaluate them through experimentations with application systems that were experimental but practical in their complexity. In addition, before developing full PIM systems, we developed a smaller-scale system called Multi-PSI with up to 64 processors for accelerating R&D of software systems.

Scalability was our first concern. We were not aiming at developing technologies that could only solve problems we have today, but technologies that will be needed in the next century and on. Thus, we chose the interprocessor network as the global communication medium. Some models also have shared memory clusters, consisting of up to eight processors, but even with such models, clusters are connected by communication network.

Table 1 shows various aspects of Multi-PSI and the five models of PIM. All the models have an implementation of the common kernel language KL1, and most of them run the common operating system PIMOS described below. Thus, the same application software can be run on different architecture.

3 Language

A concurrent logic language, KL1 [1], was designed as the kernel language of the system to give the basis of both hardware and software technologies.

KL1 is a born concurrent language in which concurrent computation is the default. Its automatic dataflow synchronization mechanism eliminates most of the synchronization problems. Physical parallelism is specified as "pragma" in the language, clearly distinguished from logical concurrency. This allows modification of physical parallelism without touching the concurrency and synchronization behavior of programs. This made experimentation with parallel algorithms much easier than the traditional approach.

3.1 Basic Mechanism

KL1 is a concurrent logic programming language based on GHC [2]. Its basic execution mechanism is common with other languages of the family, such as Concurrent Prolog [5], Parlog [6] or Janus [7].

KL1 programs consist of *clauses*, each of which corresponds to a logical axiom. Clauses that define a program have the following syntax:

PredName(*ArgList*, ...) :- *Guard* | *Body*.

Each part has the following operational meanings:

PredName gives the name of the predicate (or subroutine) for which this clause gives (a part of) the definition.

ArgList determines the correspondence of actual arguments given to the predicate and the variables written in the clause definition.

Guard specifies the condition needed to be satisfied to apply the clause. Any number of *goals*, i.e., invocation of predicates, can be written separated by commas, and the condition is considered to be satisfied when all of them are satisfied. In the guard, only unifications and invocations of certain predicates defined in the language can be written.

Body specifies the action to be taken when the clause is selected. Like the guard, any number of goals can be given here and all the goals will be executed when the clause is selected. Unlike in guard, user-defined predicates can be invoked from the body, in addition to unifications and language-defined predicates.

Execution of KL1 programs proceeds roughly as follows.[1]

1. First, the initial goal is the only member of a multiset of goals called the *goal pool*.

2. Some of the goals in the goal pool are picked up.

[1] Actual implementations are more optimized.

Table 1: Summary of PIM hardware systems.

Model	Year	#Proc.	/node	Mem./node	Internode connection
PIM/p	'92	512	8	256MB	hypercube, 33MB/s
PIM/m	'91	256	1	80MB	2-D mesh, worm hole, 8MB/s
PIM/c	'92	256	8	160MB	crossbar, 40MB/s
PIM/k	'92	16	4	256MB	hierarchical bus, 80MB/s
PIM/i	'92	16	8	320MB	SCSI
Multi-PSI	'88	64	1	80MB	2-D mesh, worm hole, 5MB/s

3. Goals picked up are matched against *clauses* of the program.

4. If there is some clause with its *head* matching a goal and its *guard* is satisfied, the original goal will be reduced to goals in the *body* of the clause and the resultant new goals will be put back to the goal pool.

5. Steps 2 through 4 are repeated until the goal pool becomes empty.

Steps 2 through 4 can be done in parallel for many goals at a time. This is the source of concurrency in this language.

The most notable features of the concurrent logic programming languages are their side-effect-free semantics and implicit dataflow synchronization mechanism. Since no notion of *assignment* is in the language, the value of a variable, once defined, will never change as the computation progresses. The dataflow synchronization mechanism assures that, whenever a decision is to be made for conditional execution, it is suspended automatically until all the data required for the decision, such as operands to a comparison, get ready.

The combination of these features assures that there will never be synchronization problems such as

- Overwriting a variable before its value is read
- Reading a variable's value before it is set

Programs in KL1 are usually organized using the object-oriented programming style [8]. Almost the whole PIMOS operating system and many of the application systems running on PIM are written in this way.

3.2 Computation Mapping

KL1 provides only low-level process distribution and priority-based scheduling features for controlling computation mapping. It seems that, at least with the status quo technology, no automatic load-distribution schemes are universally effective to all kinds of algorithms. Our decision thus was to provide lower level primitives in the programming language level and to make the software written in it responsible for computation mapping.

The primitives provided in KL1 are as follows. Note that they are no more than pragmas that only suggest the language processor for better performance; they will not change the meaning of the programs.[2]

Processor specification. Each body goal may have a processor specification that designates the processor on which to execute the goal.

Priority specification. Each body goal may have a priority specification. Each goal has an integer priority associated with it.

Although process distribution is specified by pragmas, data referenced by distributed processes are fetched from remote memory automatically on demand. The side-effect-free semantics of KL1 allows copying of any data except for undefined variables without affecting the meaning of programs. Executable codes are also distributed on demand; i.e., when a certain piece of code is needed on some processor and the code is not in the memory of that processor, it will be fetched from some other processor automatically. Memory

[2]To be precise, pragmas will not change the partial correctness of programs, but certain diverging programs may be assured to stop through pragma specifications.

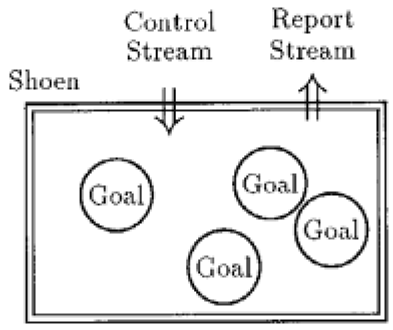


Figure 2: Shoen and related streams.

areas occupied by data structures or executable code no longer needed are reclaimed with the garbage collection mechanism by using the scheme described below.

Several automatic mapping strategies have been developed for diverse problems in the software level using the above straightforward mechanism. Relatively universal ones are provided as libraries and are used in many application software systems [9].

3.3 Metalevel Control

With the basic semantics of the concurrent logic programming languages, all the goals in the system form one logical conjunction. This means that a failure or an exception in one of the goals makes the whole system fail. Also, there is no way to control execution of such goals. With this semantics, it is almost impossible to build a system that requires efficient metalevel control on computation activities, such as an operating system.

KL1 thus provides a metalevel execution control feature called "shoen."[3] A shoen is a group of goals. This group is used as the unit of metalevel control, namely, initiation, interruption, resumption, and abortion of execution. The shoen construct also provides exception handling and resource-consumption control mechanisms.

A shoen has two communication streams as its interface: one, called the "control stream," directs inwards from outside of shoen for sending messages to control the execution; the other, called the "report stream," directs the reverse way for reporting events internal to the shoen, such as exceptions (figure 2).

PIMOS uses this shoen structure to construct a higher-level notion of "task," which is the operating-system-level unit of resource management. Note that tasks are *not* a unit for parallel execution. There are usually many parallel activities within one task.

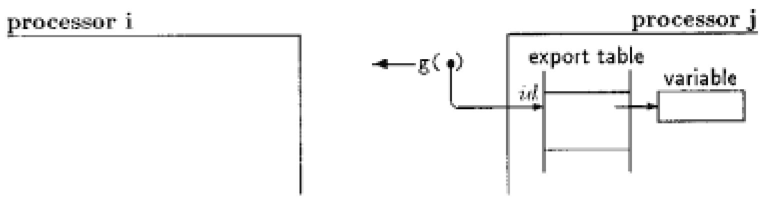
4 Distributed Implementation

This section describes the management of external references, distributed unification, and distributed goal control.[4]

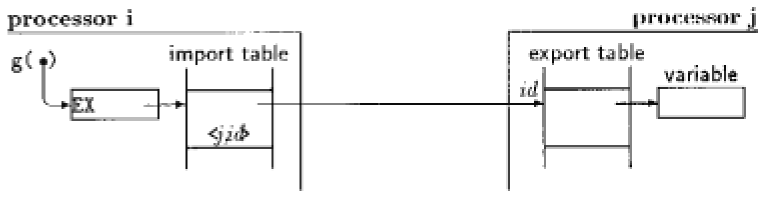
In a distributed environment, each processor may have references across processors, called *external references*. A unify request to an external reference causes message sending to the referenced processor to perform unification, which is called *distributed unification*. A body goal with processor specification pragma may migrate among processors. A shoen

[3]The Japanese word "shoen" roughly corresponds to the English word "manor."

[4]Further details can be found in [10, 11].



(1) **processor i** generates an external reference ID and throws a goal **g**.



(2) **processor j** receives the goal and registers the external reference ID.

Figure 3: Extenal reference.

is implemented using a proxy shoen called *foster parent* on each processor where the goals of the shoen reside.

4.1 External Reference Management

4.1.1 External Reference

When a goal is thrown to another processor and the arguments of the goal are references to undefined variables or structures, the references across processors consequently appear. These are the *external references*.

When a goal with reference to an object in **processor j** is thrown to **processor i**, the original **processor j** *exports* the reference to the object to **processor i**, and foreign **processor i** *imports* the reference to the object from **processor j**.

An external reference could have been straightforwardly represented by a pair $< proc, addr >$, where *proc* is the processor number in which the referenced object resides, and *addr* is the memory address of the object. However, such an implementation causes crucial problem; efficient local garbage collection (garbage collection within a processor) is impossible. If locations of objects change as a result of local garbage collections, it must be announced to all processors that may have references to the objects.

In order to overcome this problem, each processor maintains an *export table* to register all locations of objects that are referenced from outside. An external reference is represented by a pair $< proc, ent >$, called *external reference ID*, where *ent* is the entry number of the export table (figure 3).

When externally referenced objects are moved as a result of garbage collections, the references from the export table entries are updated to reflect the moves, while the external reference IDs are not affected.

4.1.2 Reexportation

Since an exported object is identified by its external reference ID, distinct IDs are regarded as distinct objects even when they are identical. Since an undefined variable or a structure may be exported to the same processor more than once, if the reexported object is given a different ID, redundant read/write request messages may be sent.[5] To eliminate redundant interprocessor communications, an exported object should not have more than one external reference ID.

A hash table is attached to an export table to retrieve the same export table entry from the same exported object. Also, each processor maintains an *import table* to register all imported external reference IDs. There is a hashing mechanism for retrieving the import table entry from the imported ID, so that when a processor imports the same ID more than once, only one import table entry is allocated.

4.1.3 Interprocessor Garbage Collection by WEC

Since export table entries cannot be freed by a local garbage collection, there must be some interprocessor garbage collection mechanism to free those entries that become garbage. To realize interprocessor garbage collection, the *weighted export counting* (WEC) scheme [12] is employed, which is based on the *weighted reference counting* (WRC) scheme [13, 14] and is a generalization of standard reference counting.

The WEC scheme associates some weight (positive integer) to external references (import table entries in processors and external reference IDs in messages) and export table entries, so that the following is invariant for every export table entry E :

$$weight\ of\ E = \sum_{x\ in\ references\ to\ E} weight\ of\ x$$

The above equality ensures that the weight of export table entry E reaches zero if and only if there are no external references to E either in processors or in messages.

When a new export table entry is allocated, the same weight is assigned to both the export table entry and the external reference ID. When a processor receives a external reference ID, it adds the weight assigned to the received ID to the weight of the import table entry, which registers the same ID. If there is no corresponding entry, the processor allocates a new import table entry and registers the received ID.

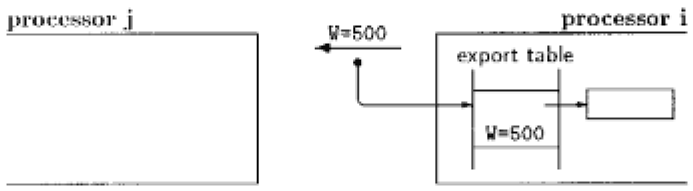
When a processor throws a external reference, the processor assigns a weight to the external reference ID and subtracts the same amount from the weight of the corresponding import table entry. The new weight of the entry and that assigned to the thrown external reference ID should be both positive, and the sum of the two weights is equal to the original weight of the entry.

When an import table entry is released, its weight is returned to the corresponding export table entry by a `%release` message. On receiving a `%release` message, the weight of the export table entry is decreased by the returned weight. If the weight of the export table entry reaches zero, the entry is freed. Figure 4 shows external reference management under the WEC scheme.

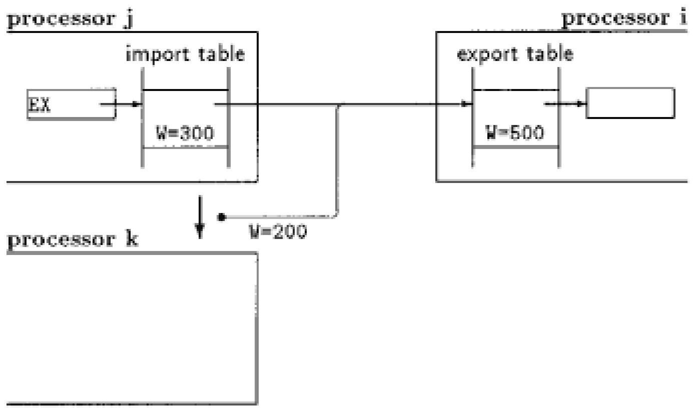
4.1.4 Indirect Exportation

When the weight of an import table entry is one, the processor cannot throw the external reference, because nonzero weight must be assigned to the thrown external reference ID and nonzero weight must also remain in the import table entry after throwing. In this situation, the processor performs an *indirect exportation*; it registers the external reference itself and generates a new external reference ID (see figure 5).

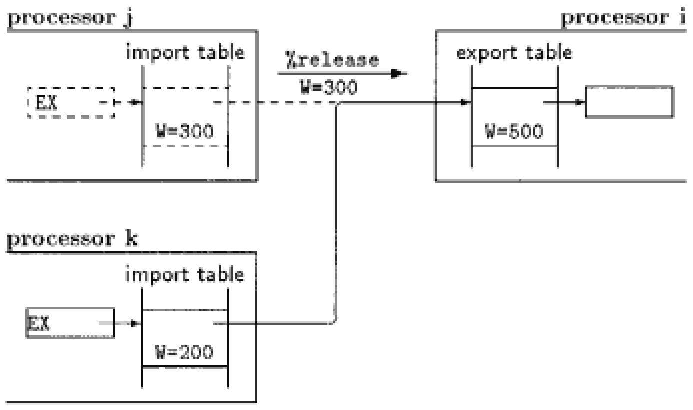
[5]Read/write operations (guard/body unification) are described in section 4.2.



(1) An external reference with weight 500 is sent.



(2) Weight is split into two positive weights.



(3) The external reference in **processor j** is released.

Figure 4: The WEC scheme.

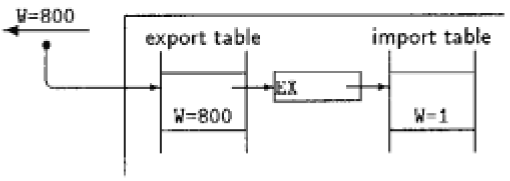
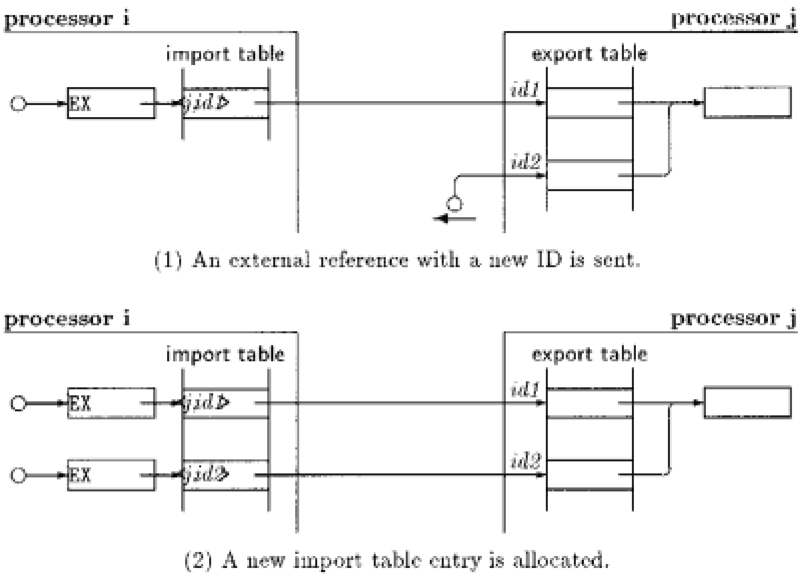


Figure 5: Indirect exportation.



(1) An external reference with a new ID is sent.

(2) A new import table entry is allocated.

Figure 6: Low-cost exportation and importation.

4.1.5 Low-Cost Exportation and Importation

The WEC mechanism mentioned above has overhead in terms of maintaining weight and of looking up the hash table to check the reexporting and reimporting.

To minimize the cost of exportation and importation, on each exportation of a single reference object, a simple export table entry is newly allocated and a simple external reference ID with no weight is generated. Also, on each importation of a simple external reference ID, a simple import table entry is allocated.[6] When a simple import table entry is released, a `%release` message with no weight is sent to the corresponding simple export table entry. On receiving the `%release` message, the export table entry is released.

Since both a simple export and a simple import table entry consist of only one field, and no hashing mechanism is necessary, the cost of the management of the table is quite low. Figure 6 shows the low-cost exportation and importation.

Single references and multiple references are distinguished by the *Multiple Reference Bit* (MRB) mechanism [15, 16] (see figure 7 and 8). Since there exist at most two single references to an object, exportation through a single reference can be done at most twice. Therefore, allocating a new export table entry and attaching a new external reference ID on each exportation of a single referenced object causes no redundant communication; at

[6]These simple tables and simple IDs are called *white* export/import tables and *white* external reference IDs, while another complex ones are called *black* ones [12].

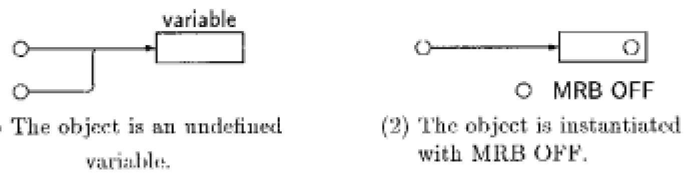

variable
○ MRB OFF
(1) The object is an undefined variable.
(2) The object is instantiated with MRB OFF.

Figure 7: Single references.

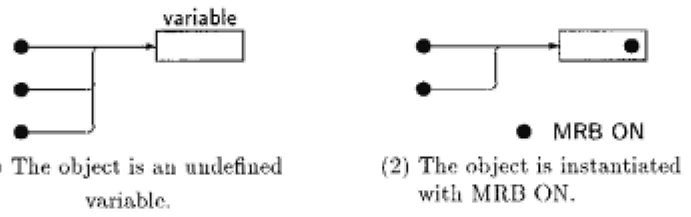

variable
● MRB ON
(1) The object is an undefined variable.
(2) The object is instantiated with MRB ON.

Figure 8: Multiple references.

most, two read/write request messages may be sent.

4.2 Distributed Unification

Distributed unification is implemented based on message passing.

4.2.1 Guard Unification

The guard unification is performed in a *read and compare* manner. To read the value of the object referenced by the external reference X, the return address for the response is allocated, and the following read request message is sent to the referenced processor:

```
%read(X,RetAdr)
```

After sending a `%read` message, the goal that causes the guard unification is hooked to the external reference X.

If the referenced object has a concrete value, it is returned by an `%answer_value` message, shown below:

```
%answer_value(RetAdr,value)
```

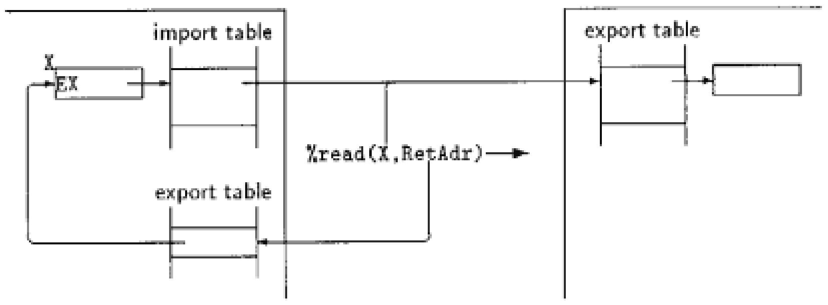
Figure 9 shows the operations described above.

If the referenced object is an undefined variable, the read request is hooked to the variable, and the return of the value is suspended. If the object is an external reference, a `%read` message is forwarded to the referenced processor.

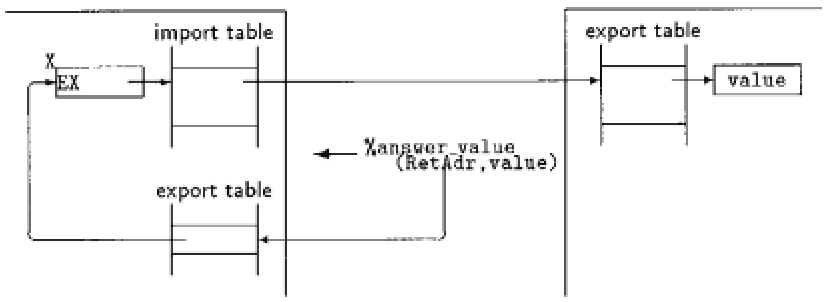
On receiving an `%answer_value` message, the external reference X is overwritten by the value carried, and the import table entry referenced by X can be freed. Consequently, the hooked goal waiting for the reply is resumed.

4.2.2 Body Unification

If an argument of body unification is an external reference, the body unification has to realize the unification in a remote processor. Body unification between an external reference X and a term Y is done by sending the following message:



(1) Allocating a return address and sending a **%read** message.



(2) Sending back an **%answer_value** message carrying a value.

Figure 9: Distributed guard unification.

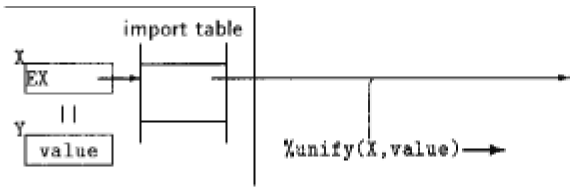


Figure 10: Distributed body unification.

`%unify(X,Y)`

This is a request to unify the object referenced by X with a term Y (figure 10). The processor that receives the above message performs the body unification. When the referenced object is an external reference, a **%unify** message is passed to the referenced processor.

If both arguments are external references, a **%unify** message is sent to one of the referenced processors.

4.2.3 Avoidance of Reference Loop Creation

A *reference loop* is a closed chain of references. If there were a reference loop, the objects on the loop would not have dereferenced results, and they could not be unified with any concrete value. An unrestricted unification algorithm can create reference loops as follows.

> **processor i** has an external reference `Xi` that references an undefined variable `Xj` in **processor j**, while **processor j** has an external reference `Yj` that references an undefined variable `Yi` in **processor i**. If body unification between

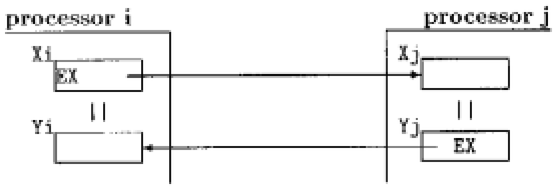


Figure 11: Creation of a reference loop.

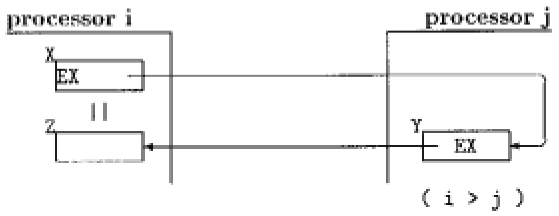


Figure 12: Simply comparing processor numbers is insufficient.

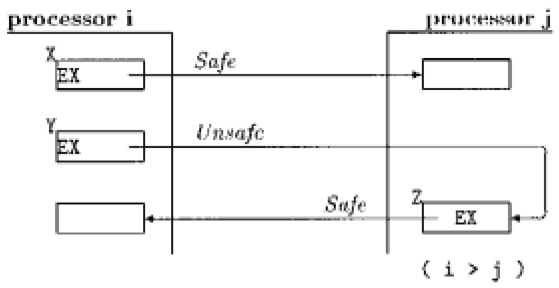


Figure 13: Safe/Unsafe attribute.

Xi and Yi in **processor i** causes Yi to be bound to Xi, and body unification between Yj and Xj in **processor j** causes Xj to be bound to Yj, a reference loop is created (figure 11).

In [17], this problem is solved by imposing the binding order rule: a binding of an undefined variable to an external reference is permitted only when the current processor number is larger than the referenced processor number; otherwise, a `%unify` message is sent to the referenced processor.

However, in the presence of indirect exportation, this *simply comparing processor numbers rule* is no longer sufficient. For example, in figure 12, body unification between an external reference X and an undefined variable Z causes Z to be bound to X and creates a reference loop.

To cope with the presence of indirect exportation, the *Safe/Unsafe* attribute is attached to each external reference [12] (figure 13). An external reference E is unsafe, if and only if the processor number in which E resides is smaller than the processor number referenced by E, or the object referenced by E is an unsafe external reference. An external reference E is safe if it is not an unsafe reference.

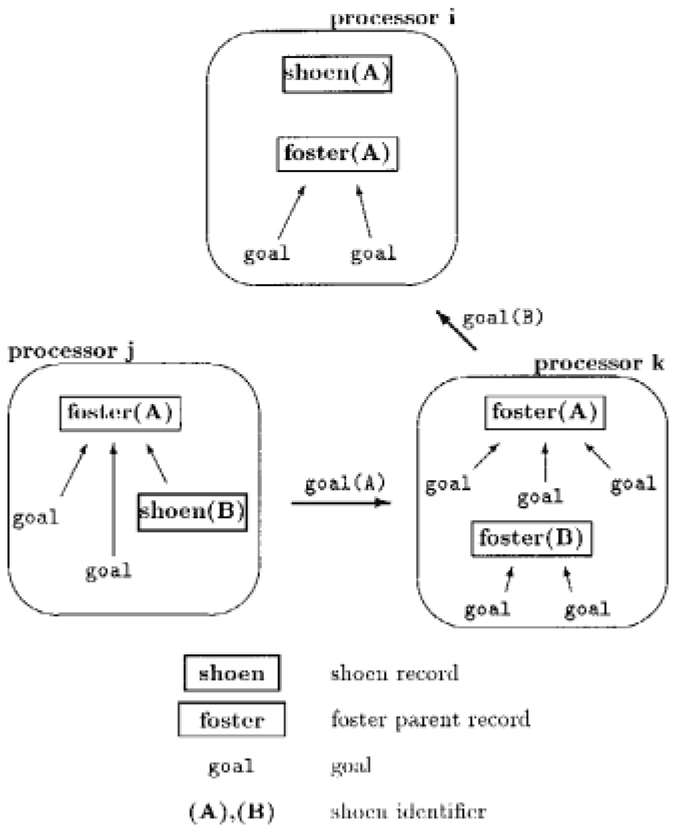


Figure 14: Shoens, foster parents, and goals.

Body unification between an external reference X and an undefined variable Y is made as follows:

- If X is safe, Y is bound to X.
- If X is unsafe, a `%unify` message is sent to the processor referenced by X.

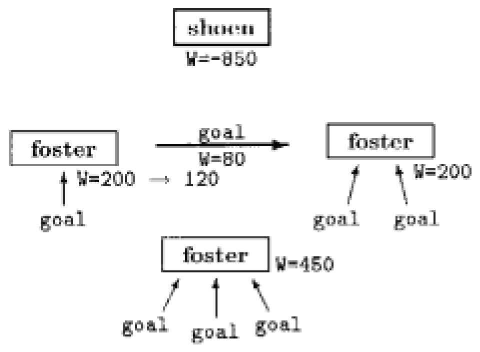
4.3 Distributed Goal Control

4.3.1 Shoen and Foster Parent

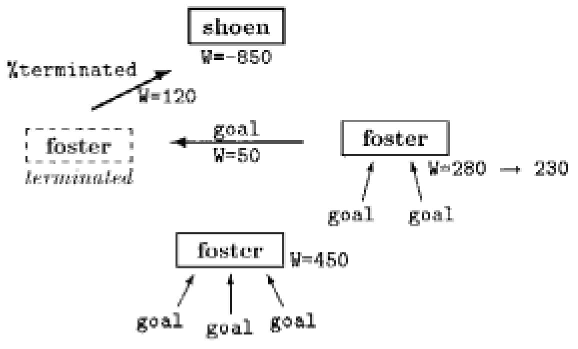
A shoen supports the metacontrol facilities of execution control, resource management, and status monitoring for the goals. The implementation model for a shoen on a distributed environment introduces a foster parent to reduce an access bottleneck at the shoen. A foster parent is a proxy shoen located on processors where the goals of the shoen are executed. A shoen and a foster parent are realized by record structures that store their details, such as status, resources, and number of goals.

Figure 14 shows the relationship among shoen records, foster parent records, and goals. Each goal in a processor has a pointer that points to its foster parent record, and in transit has a *shoen identifier*. When shoens are nested, the descendant shoen record points to its parent foster parent record like a goal.

On creation of a shoen, a shoen record and a foster parent record are allocated, and the initial goal points to the foster parent record. When a goal arrives at a processor



(1) Throwing a goal with weight.



(2) Returning the weight of the terminated foster parent.

Figure 15: The WTC Scheme.

where no corresponding foster parent record exists, a foster parent record is created in the processor.

A foster parent terminates when all goals and all descendant shoens belonging to the foster parent terminate. Furthermore, a shoen terminates when all foster parents belonging to the shoen terminate and there are no goals in transit.

4.3.2 Termination Detection of Shoen

Termination detection of a shoen is a difficult subject in a distributed environment. Each foster parent can detect the termination of all goals and all descendant shoens belonging to the foster parent, and can report it to its shoen. However, even if a shoen receives the report from all the foster parents, it is not sure that all goals have terminated. There may be goals in transit, which will arrive at a certain processor and a foster parent will be created.

To detect the termination of a shoen efficiently, the *weighted throw counting* (WTC) scheme [18] [7] is introduced. This scheme is an application of the *weighted reference counting* (WRC) scheme [13, 14], which is a garbage collection scheme for parallel processing systems, and can efficiently detect termination without probing or acknowledgment.[8]

In this scheme, the shoen, each foster parent, and each message in transit have some *weight*. The weight of a message in transit and that of a foster parent are positive integers,

[7] Essentially the same scheme named the *Credit Distribution and Recovery* algorithm is presented in [38]. *Credit* in [38] corresponds to *weight* in the WTC scheme.

[8] Derivation of the WTC scheme from the WRC scheme is described in [39].

while the weight of the shoen is a negative integer. The WTC scheme maintains the invariant that

The sum of the weights is *zero*.

This ensures that the weight of the shoen reaches zero if and only if all foster parents, all goals, and all descendant shoens terminate and there are no messages in transit.

When all goals and all descendant shoens in it terminate, the foster parent terminates and sends a `%terminated` message to the shoen. The `%terminated` message carries the weight of the terminated foster parent. On receiving a `%terminated` message, the shoen adds the weight to its (negative) weight (figure 15). If the weight of the shoen reached zero, the termination of a shoen is detected.

4.3.3 Forced Termination of Shoen

If a shoen broadcasts an `%abort` message causing the foster parent to terminate, it is possible to abort all the goals and all the descendant shoens belonging to the foster parent, but impossible to abort the goals in transit. After receiving an `%abort` message and aborting the foster parent, the processor may receive a thrown goal and a foster parent may be created.

Since termination is detected using the WTC scheme described above, only delivery of an `%abort` message to each foster parent is required to achieve abort. To send an `%abort` message to such foster parents that are created during abort, and not to send to processors that have no foster parents, a `%ready` message is introduced. When a foster parent is created, it sends a `%ready` message to the shoen, which gives notification of the creation of a foster parent. On receiving a `%ready` message, the shoen memorizes the sender processor of the `%ready` message, which is deleted on receiving a `%terminated` message.

The shoen performs the following operations to achieve the abort:

(A) Sending an `%abort` message to each processor memorized;

(B) Sending an `%abort` message to the sender processor of the `%ready` message received after (A).

An `%abort` message has the shoen identifier and some weight like a goal in transit.

All foster parents already detected by the shoen are aborted by (A). (B) aborts foster parents that were not recognized by the shoen when (A) was carried out, namely, a foster parent that is created after (A), or created before (A) but whose `%ready` is still in transit.

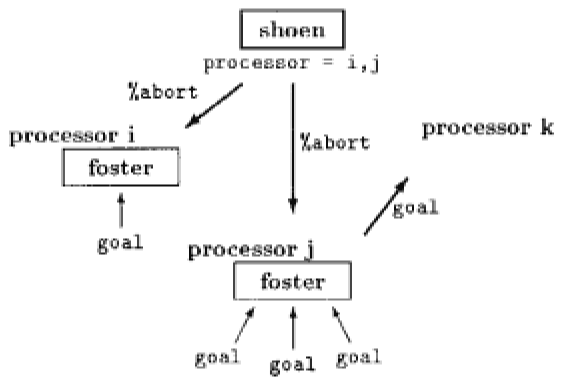
On receiving an `%abort` message, the foster parent terminates and sends back a `%terminated` message which carries the sum of the weight of the terminated foster parent and the `%abort` message. If there is no corresponding foster parent, the weight assigned to the `%abort` message is returned back to the shoen.

Figure 16 shows the abort operations described above.

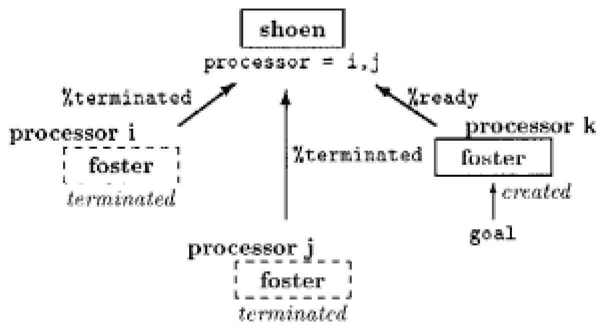
4.3.4 When the Weight Becomes One

When the weight of a foster parent becomes *one*, it cannot throw a goal, because nonzero weight must be assigned to the goal, and nonzero weight must remain also in the foster parent after throwing.

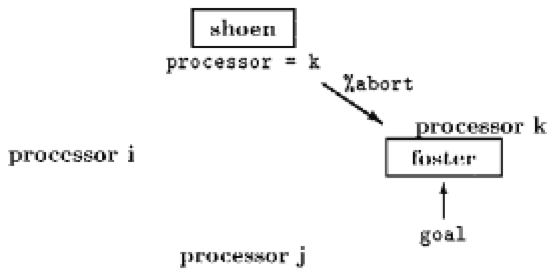
In this case, the foster parent sends a `%request` message requesting more weight to the shoen. Goal throwing is suspended until the weight of the foster parent becomes more than one. On receiving a `%request` message, the shoen sends back a `%supply` message



(1) Sending an %abort to the processors memorized.



(2) Receiving a %ready.



(3) Sending an %abort as a response of the %ready.

Figure 16: Abort operations.

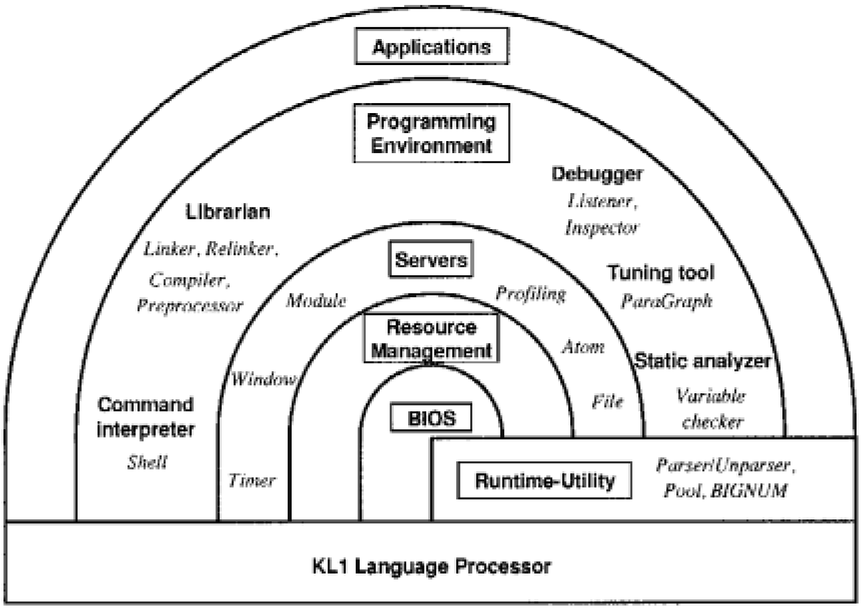


Figure 17: The structure of PIMOS.

that carries some weight to the sender foster parent and reduces the same amount from its own weight. When a foster parent receives a `%supply` message, it adds the weight carried by the message to its weight, which enables it to throw any suspended goal.

Since receiving a thrown goal also increases the weight of the foster parent, a foster parent may terminate before receiving a `%supply` message. A `%supply` message may thus reach a processor that contains no corresponding foster parent. In this case, the weight carried by the `%supply` message is sent back to the shoen. This is similar to the action when an `%abort` message reaches a processor with no corresponding foster parent.

5 Programming Environment

PIMOS is an operating and programming system for all models of parallel inference machines. Its overall structure is shown in figure 17. Some of the characteristic submodules of PIMOS are as follows.

BIOS provides the most basic I/O through the SCSI interface. KL1 provides a process model of the SCSI interface through built-in predicates.

Resource management provides a layer for communication management. Since all the I/O devices and tasks have a message-stream interface, resource management by PIMOS is effected by controlling usage of such streams [19].

Tasks, implemented using the shoen mechanism, can be nested with arbitrarily many levels. Thus, processes to control tasks form a tree structure, called the "resource tree." Processes to control communication with "servers" (described below) are also in this tree. By distributing such processes to the processor where the user programs made the request, the overhead of the operating system can be distributed without increasing the amount of communication.

Servers are processes to implement virtual devices upon physical devices [19]. Services provided by software systems, such as the database management system kawa:m or the librarian of the PIMOS described below, also have this server interface.

Debugging tools provide features tailored for debugging parallel programs. It includes the "listener," with its tracing and spying functions, and the "inspector," which inspects data structures either statically or during their dynamic generation.

ParaGraph is a program tuning tool providing a graphic display of execution profile information [21]. Such a visualization tool is quite powerful in tuning the performance of programs through changing the mapping pragmas as stated above.

Note that, as stated above, changing pragmas will not change the meaning of the programs. The change only affects the performance. Data and the executable code required are automatically distributed. This makes the tuning of load distribution much easier.

Librarian is responsible for maintaining the correspondence with executable code modules and their names. It is implemented as a server.

Note that the side-effect-free nature of the KL1 does not allow even executable code to be overwritten. Updating a program module means generating a new one and changing the name correspondence. The older version may still be running somewhere in the system (possibly on the same processor). This scheme may seem inefficient, but actually it is not, since updating executable code is not so frequent. The clean semantics, on the other hand, makes users' understanding much easier.

6 Application Systems

Many experimental application systems have been developed and are running on PIMOS and PIM (currently, PIM/m and PIM/p systems are mainly used). In parallel with the development of independent application systems, performance analysis study from a more general standpoint is also ongoing (see [22], for example.)

Some of these application systems are described below. All of them and many others are available as free software from ICOT.

GDCC is a parallel constraint logic programming system. It provides highly declarative, flexible, and efficient constraint logic programming languages, dealing with various kinds of constraints including nonlinear polynomial equations [23] .

Quixote is a language system providing fundamental facilities for knowledge information processing, such as very high-level knowledge representation and inferences [24].

Kappa-P is a parallel database management system based on a nested relational model [20].

MGTP is a massively parallel, high performance bottom-up theorem prover on first-order problems [25].

The prover tries to generate models satisfying a given axiom set. The system can be used in two ways. As a theorem prover, it will show that no models can satisfy a given axiom set augmented with the negation of the given theorem. As a constraint satisfaction system, models found by the system to satisfy a given axiom set are answers to a constraint satisfaction problem. The system showed almost linear speed-up up to 512 processors.

Genetic information processing software includes several software systems that help analysis of genetic information by biologists.

The protein sequence alignment systems have several variations, based on parallel dynamic programming [26], parallel simulated annealing [27] and the knowledge-based approach using Quixote as mentioned above [28].

The protein conformation prediction system is for predicting the 3-D structure of proteins from amino acid sequences based on geometrical stochastic reasoning [29].

Parallel logic simulator is a system to simulate VLSI circuits to verify their logical and timing specifications.[30]

This system adopts a virtual time algorithm, in which the simulation proceeds in a speculative way, assuming input signal to be notified from other processors will not change. If such a change is notified later, the simulation will be rolled back. This strategy extracted much higher parallelism than a nonspeculative algorithm, obtaining 166-fold speed-up on 256 processors of PIM/m (534k events/sec). Although the parallel algorithm used is rather complex, it took only several man-months to build its first version.

Other systems in the VLSI CAD area include cell placement systems [31, 32], LSI router [33], and circuit minimization system [34].

Helic-II is a parallel legal reasoning system referencing to laws and precedents [35].

Reasoning from the legal viewpoint is not a simple inference process based only on laws and regulations, since many words and phrases appearing in themselves are left undefined. Their interpretation is based on precedents. The system keeps two kinds of databases, one on laws and regulations and another on precedents. New cases are matched against precedent cases using higher level interpretation (such as matching a taxi driver and a flight pilot as a traffic operator). The laws and regulations are applied afterwards, using the above-mentioned MGTP as the inference engine.

Mendels zone is a software synthesis system for concurrent programs that allows very high-level specification.

The specification of methods in the system is first described in equational logic, the correctness of which is verified automatically. Timing constraints within the methods themselves and between different methods are described in temporal logic. According to the constraints, the structure of the whole program will be converted to a Petri net, which is then converted to a KL1 program automatically. All the conversion and verification processes are executed in parallel, much reducing the computation time required.

In an experiment of building a plant control expert system, 6200 lines of KL1 program were generated from 4000 lines of high level specification, requiring five man-months of human effort. In comparison with a separate activity of writing the system directly in KL1, the program size increased by 34%, but the original description decreased by 15%. The mon-months required were almost halved, and, notably, the debugging process required only one twelfth as much of human effort in direct description.

7 Through Our Experiences

This section summarizes our experiences with building software systems in KL1 on the parallel inference machines.

7.1 Programming Ease

The automatic synchronization mechanism and fine grain concurrency of KL1 made programming much easier. The software productivity became far better than in sequential programming languages with baroque parallel processing extensions.

When we started developing the first version of PIMOS in 1987, there were no parallel KL1 language implementations available. Thus, the operating system was first debugged on a sequential (pseudoparallel) implementation, which had only fixed scheduling strategy.

When the system was ported to a prototype parallel machine, Multi-PSI, in 1988, we were ready to deal with the annoying synchronization bugs that will not reproduce themselves, although the automatic synchronization mechanism of the language should avoid such problems in theory.

The theory turned out to be the reality. We found almost no synchronization problems except for a small number of design problems at a very high level, although the scheduling on the real parallel machine is quite different from the emulator. We knew this in theory, but actually experiencing it made us much more confident of the virtue of writing a system in a language with dataflow synchronization. When PIMOS was later ported to several other models of PIM systems, each with its own scheduling strategy, we almost never encountered synchronization problems.

Most of the experimental application systems written for PIM were coded by programmers with no experience in parallel processing. Nevertheless, they did not seem to have much problem with synchronization.[9]

7.2 Development Environment

Software development tools, including debugging tools and performance tuning tools tailored for parallel processing, have been found indispensable for high software productivity.

Most of the application software systems were first developed on the Multi-PSI system with up to 64 processors, then ported to PIM/m with 128 processors when it was ready for use, then to its 256 processor version when its production had completed, and then to PIM/p with 512 processors. Many programs that showed almost linear speed-up with 16 processors would do so not with 512 processors.

Using the tuning tool ParaGraph was *very* effective in finding bottlenecks. Clear distinction between logical concurrency and physical parallelism contributed considerably to making program tuning easier. Automatic data and code distribution also helped considerably. Goal distribution specification pragmas were the only changes needed for load distribution tuning.

As a whole, it usually took only a few weeks before a new version with an improved load distribution algorithm showed good parallelism on larger-scale systems.

[9]However, those who had been very accustomed to Prolog found it difficult to realize the large semantic differences of two languages with similar syntax.

7.3 High-Performance Hardware

The largest-scale PIM system, PIM/p with 512 processors, showed a total throughput of more than 150 MLIPS.[10] This 150 MLIPS is comparable to 5 to 10 GIPS (giga instructions per second).

Hardware systems with such amazing performance have been very helpful in accelerating software research. In the earliest phases in the development, application developers did not have to write a highly optimized version. The high-performance hardware allowed running programs written without too much concern about execution performance for problems of realistic sizes. The developers could then tune their programs gradually by using performance tuning tools. This was especially helpful for developing considerably complicated knowledge processing systems.

Comparative study of different architectures is still going on, but unfortunately, comparison has not been easy. The behavior of the software is more influenced by slight differences in language implementation rather than by the architectures, and redoing the language implementation requires too much effort. One thing we can say is that relatively low throughput and a long delay of interprocessor network can usually be hidden by sophisticated software ideas.

8 Current Status and Future Work

The parallel inference system on PIM has been and still is used as tools for R&D of parallel application software research.

A serious problem we have with the system is that it runs only on specially devised hardware. Although the system is efficient and self-contained, requirement of special hardware is a great obstacle in sharing the environment with researchers world-wide.

Research in subsetting the language to allow more concise and efficient implementation has been conducted with promising preliminary results [36]. A separate effort to implement KL1 by translating into C [37] shows reasonable performance with very high portability. These results indicate the future direction of implementing the language and the system on stock hardware to be shared among a wider range of researchers in parallel software area.

The Fifth Generation Computer Systems project ended in March 1993. The Japanese Ministry of International Trade and Industry, considering the above-mentioned recent research on implementation, launched a new two-year project beginning from April 1993, aiming at disseminating the technologies established in the FGCS project by amalgamating it with conventional computer technologies, such as UNIX and RISC processors. In this project, the following results are expected:

- KL1 implementation with reasonable software development environment on commercially available hardware. An implementation with excellent portability by compilation into C (nicknamed KLIC, for KL1 in C) for UNIX workstations, parallel UNIX systems, and network-connected UNIX systems is being developed. Its prototype is already working, showing excellent portability with single-processor efficiency much better than already available Prolog implementations.

- Knowledge processing software and experimental application software further refined and ported to KLIC. It is also planned to include software already available on UNIX

[10]LIPS (logical inference per second) is a unit used to measure performance of logic programming language implementations. 1 LIPS means that one logical inference step of "naive reverse" program can be executed in one second.

systems as components of such software by utilizing the foreign language interface provided by KLIC.

All the resultant software is planned to be freely available worldwide to be utilized as the basis of further research in the area of knowledge information processing systems. Some part of the work, including KLIC, has already been released as free software. Further inquiries should be directed to the electronic mail address `ifs@icot.or.jp`.

References

[1] K. Ueda and T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," *The Computer Journal,* **33 (6)**: 494–500, Oxford University Press, 1990.

[2] K. Ueda, "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard," Technical Report **208**, ICOT, 1986.

[3] T. Chikayama, "Operating System PIMOS and Kernel Language KL1," *Proc. International Conference on Fifth Generation Computer Systems 1992,* pp. 73–88, Ohmsha, 1992.

[4] K. Taki, "Parallel Inference Machine PIM," *Proc. International Conference on Fifth Generation Computer Systems 1992,* pp. 50–72, Ohmsha, 1992.

[5] E. Shapiro, "Systems Programming in Concurrent Prolog," M. van Canegham and D. H. D. Warren (eds.), *Logic Programming and its Applications,* pp. 50 74, Albex Publishing Co., 1986.

[6] K. Clark, *et al.,* "PARLOG: Parallel Programming in Logic," *ACM Trans. Programming Language Systems,* **8 (1)**, 1986.

[7] V. A. Saraswat, *et al.,* "Janus: A Step Towards Distributed Constraint Programming," *Proc. North American Conference on Logic Programming 1989,* pp. 497–512, MIT Press, 1990.

[8] E. Shapiro and A. Takeuchi, "Object-oriented Programming in Concurrent Prolog," *New Generation Computing,* **1 (1)**: 25–49, Ohmsha, 1983.

[9] M. Furuichi, *et al.,* "A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI," *Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 50–59, 1990.

[10] K. Nakajima, *et al.,* "Distributed Implementation of KL1 on the Multi-PSI/V2," *Proc. International Conference on Logic Programming 1989,* pp. 436 451, 1989.

[11] K. Hirata, *et al.,* "Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1," *Proc. International Conference on Fifth Generation Computer Systems 1992,* pp. 436–459, Ohmsha, 1992.

[12] N. Ichiyoshi, *et al.,* "A New External Reference Management and Distributed Unification for KL1," *New Generation Computing,* **7 (2, 3)**: 159–177, Ohmsha, 1990.

[13] P. Watson and I. Watson, "An Efficient Garbage Collection Scheme for Parallel Computer Architectures," *Proc. Parallel Architectures and Languages Europe 1991,* LNCS, **259 (II)**: 432–443, Springer-Verlag, 1987.

[14] D. I. Bevan, "Distributed Garbage Collection Using Reference Counting," *Parallel Computing*, **9 (2)**: 179–192, North-Holland, 1989.

[15] T. Chikayama and Y. Kimura, "Multiple Reference Management in Flat GHC," *Proc. International Conference on Logic Programming 1987*, pp. 276–293, 1987.

[16] Y. Inamura, *et al.*, "Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2," *Proc. North American Conference on Logic Programming 1989*, pp. 907–921, MIT Press, 1989.

[17] I. Foster, "Parallel Implementation of Parlog," *Proc. International Conference on Parallel Processing 1988*, **II**: 9–16, 1988.

[18] K. Rokusawa, *et al.*, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," *Proc. International Conference on Parallel Processing 1988*, **I**: 18–22, 1988.

[19] H. Yashiro, *et al.*, "Resource Management of PIMOS," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 269–277, Ohmsha, 1992.

[20] M. Kawamura, *et al.*, "Parallel Database Management System: Kappa-P," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 248–256, Ohmsha, 1992.

[21] S. Aikawa, *et al.*, "ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 286–293, Ohmsha, 1992.

[22] K. Kimura and N. Ichiyoshi, "Probabilistic Analysis of the Optimal Efficiency of the Multi-Level Dynamic Load Balancing Scheme," *Proc. Sixth Distributed Memory Computing Conference*, 1991.

[23] S. Terasaki, *et al.*, "Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 330–346, Ohmsha, 1992.

[24] H. Yasukawa, *et al.*, "Object, Properties, and Modules in QUIXOTE," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 257–268, Ohmsha, 1992.

[25] M. Fujita, *et al.*, "Model Generation Theorem Provers on a Parallel Inference Machine," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 357–375, Ohmsha, 1992.

[26] M. Ishikawa, *et al.*, "Protein Sequence Analysis by Parallel Inference Machine," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 294–299, Ohmsha, 1992.

[27] M. Hirosawa, *et al.*, "Folding Simulation using Temperature Parallel Simulated Annealing," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 300–306, Ohmsha, 1992.

[28] M. Hirosawa, *et al.*, "Protein Multiple Sequence Alignment using Knowledge," *Proc. 26th Annual Hawaii International Conference on System Science*, **1**: 803–812, 1993.

[29] K. Onizuka, *et al.*, "A Multi-Level Description Scheme of Protein Conformation," *Proc. First Intelligent Systems for Molecular Biology*, pp. 301–310, 1993.

[30] Y. Matsumoto and K. Taki, "Adaptive Time-Ceiling for Efficient Parallel Discrete Event Simulation," *Western Multiconference on Computer Simulation*, pp. 101–106, 1993.

[31] H. Date, *et al.*, "LSI-CAD Programs on Parallel Inference Machine," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 237–247, Ohmsha, 1992.

[32] T. Watanabe, *et al.*, "Co-HLEX: Co-operative Recursive LSI Layout Problem Solver on Japan's Fifth Generation Parallel Inference Machine," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 1173–1180, Ohmsha, 1992.

[33] H. Date and K. Taki, "A Parallel Lookahead line Search Router with Automatic Ripup-and-reroute," *Proc. EDAC-EUROASIC 93*, 1993.

[34] Y. Minoda, *et al.*, "A Cooperative Logic Design Expert System on a Multiprocessor," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 1181–1189, Ohmsha, 1992.

[35] K. Nitta, *et al.*, "HELIC-II: A Legal Reasoning System on the Parallel Inference Machine," *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp. 1115 1124, Ohmsha, 1992.

[36] K. Ueda and M. Morita, "A New Implementation Technique for flat GHC," *Proc. International Conference on Logic Programming 1990*, pp. 3–17, MIT Press, 1990.

[37] T. Chikayama, *et al.*, "A Portable and Efficient Implementation of KL1," *Proc. Programming Language Implementation and Logic Programming 1994*, LNCS **844**: 25–39, Springer-Verlag, 1994.

[38] F. Mattern, "Global Quiescence Detection Based on Credit Distribution and Recovery," *Information Processing Letters*, **30 (4)**: 195–200, 1989.

[39] G. Tel and F. Mattern, "The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes," *Proc. Parallel Architectures and Languages Europe 1991*, LNCS, **505 (I)**: 137–149, Springer-Verlag, 1991.