PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Inference Machine PIM

Kazuo Taki

First Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, JAPAN
taki@icot.or.jp

Abstract

The parallel inference machine, PIM, is the prototype hardware system in the Fifth Generation Computer Systems (FGCS) project. The PIM system aims at establishing the basic technologies for large-scale parallel machine architecture, efficient kernel language implementation and many aspects of parallel software, that must be required for high performance knowledge information processing in the 21st century. The PIM system also supports an R & D environment for parallel software, which must extract the full power of the PIM hardware.

The parallel inference machine PIM is a large-scale parallel machine with a distributed memory structure. The PIM is designed to execute a concurrent logic programming language very efficiently. The features of the concurrent logic language, its implementation, and the machine architecture are suitable not only for knowledge processing, but also for more general large problems that arise dynamic and non-uniform computation. Those problems have not been covered by commercial parallel machines and their software systems targeting scientific computation. The PIM system focuses on this new domain of parallel processing.

There are two purposes to this paper. One is to report an overview of the research and development of the PIM hardware and its language system. The other is to clarify and itemize the features and advantages of the language, its implementation and the hardware structure with the view that the features are strong and indispensable for efficient parallel processing of large problems with dynamic and non-uniform computation.

1 Introduction

The Fifth Generation Computer Systems (FGCS) project aims at establishing basic software and hardware technologies that will be needed for high-performance knowledge information processing in the 21st century. The parallel inference machine PIM is the prototype hardware system and offers gigantic computation power to the knowledge information processing. The PIM system includes an efficient language implementation of KL1, which is the kernel language and a unique interface between hardware and software.

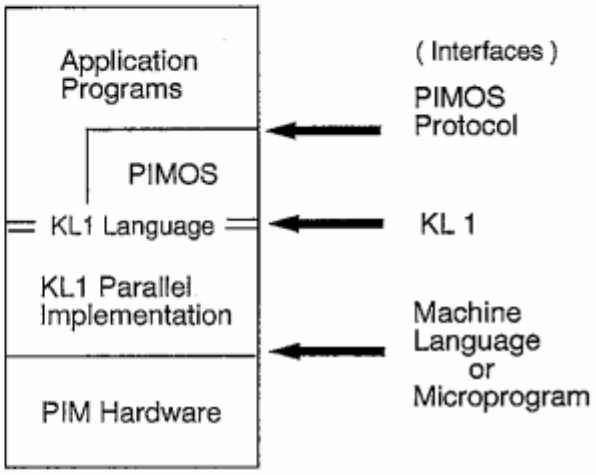

Figure 1: Overview of the PIM System

Logic programming was chosen as the common basis of research and development for the project. The primary working hypothesis was as follows. "Many problems of future computing, such as execution efficiency (of parallel processing), descriptive power of languages, software productivity, etc., will be solved drammatically with the total reconstruction of those technologies based on *logic programming*.

Following the working hypothesis, R & D on the PIM system started from scratch with the construction of hardware, a system software, a language system, application software and programming paradigms, all based on *logic programming*. Figure 1 gives an overview of the system structure.

The kernel language KL1 was firstly designed for efficient concurrent programming and parallel execution of knowledge processing problems. Then, R & D on the PIM hardware with distributed-memory MIMD architecture and the KL1 language implementation on it were carried out, both aiming at efficient KL1 execution in

parallel. A machine roughly with 1000 processors was primarily targeted. Each of these processors was to be a high-speed processor with hardware support for symbolic processing. The PIM system also focused on realizing a useful R & D environment for parallel software which could extract the real computing power of the PIM. The preparation of a good R & D environment was an important project policy.

KL1 is a concurrent logic programming language primarily targeting knowledge processing. Since the language had to be a common basis for various types of knowledge processing, it became a general-purpose concurrent language suitable for symbolic processing, without shifting to a specific reasoning mechanism or a certain knowledge representation paradigm.

Our R & D led to the language features of KL1 being very suitable for covering *the dynamic and non-uniform large problems* that are not covered by commercial parallel computers and their software systems for scientific computation. Most knowledge processing problems are included in the problem domain of *dynamic and non-uniform computation*. The PIM hardware and the KL1 language implementation support the efficiency of the language features. Thus, the PIM system covers this new domain of parallel processing.

This paper focuses on two subjects. One is the R & D report of the PIM hardware and the KL1 language implementation on it. The other is to clarify and itemize the features and advantages of the language, its implementation and the hardware structure with the view that the features are strong and indispensable for efficient parallel processing of large problems with *dynamic and non-uniform computation*. Any parallel processing system targeting this problem domain must consider those features.

Section 2 scans the R & D history of parallel processing systems in the FGCS project, with explanation of some of the keywords. Section 3 characterizes the PIM system. Many advantageous features of the language, its parallel implementation and hardware structure are described with the view that the features are strong and indispensable for efficient programming and execution of *the dynamic and non-uniform large problems*. Section 4 presents the machine architecture of PIM. Five different models have been developed for both research use and actual software development. Some hardware specifications are also reported. Section 5 briefly describes the language implementation methods and techniques, to give a concrete image of several key features of the KL1 implementation. Section 6 reports some measurements and evaluation mainly focusing on a low-cost implementation of small-grain concurrent processes and remote synchronization, which support the advantageous features of KL1. Overall efficiency, as demonstrated by a few benchmark programs, is shown, including the most recent measurements on PIM/m. Then, section 7 concludes this paper.

Several important research issues of parallel software are reported in other papers: the parallel operating system PIMOS is reported in [Chikayama 1992] and the load balancing techniques controlled by software are reported in [Nitta *et al.* 1992].

2 R & D History

This section shows the R & D history of parallel processing systems in the FGCS project. Important research items and products of the R & D are described briefly, with explanations of several keywords. There are related reports for further information [Uchida 1992] [Uchida *et al.* 1988].

2.1 Start of the Mainstream of R & D

Mainstream of R & D of the parallel processing systems started at the beginning of the intermediate stage of the FGCS project, in 1985. Just before that time, a concurrent logic language GHC [Ueda 1986] had been designed, which was chosen as the kernel language of the R & D. Language features will be described in section 3.4.

Development of small hardware and software systems was started based on the kernel language GHC as a hardware and software interface. The hardware system was used as a testbed of parallel software research. Experiences and evaluation results was fed back to the next R & D of larger hardware and software system, which was the *bootstrapping of R & D*.

It was started from development of the Multi-PSI [Taki 1988]. Purpose of the hardware development was not only the architectural research of a knowledge processing hardware, but also a preparation of a testbed for efficient language implementation of the kernel language. The Multi-PSI also focused to be a useful tool and environment of parallel software research and development. That is, the hardware was not just an experimental machine, but a reliable system being developed in short period, with measurements and debugging facilities for software development. After construction of the Multi-PSI/V1 and /V2 with language implementations, various parallel programs and technology and knowhow of parallel software have been accumulated [Nitta *et al.* 1992] [Chikayama 1992]. The systems have been used for the advanced software development environment for the parallel inference machines.

2.2 Multi-PSI/V1

The first hardware was the Multi-PSI/V1 [Taki 1988] [Masuda *et al.* 1988], started in operation in spring 1986. The personal sequential inference machine PSI [Taki *et al.* 1984] was used for processing elements. It was a development result of the initial stage of the

project. Six PSI machines were connected by a mesh network, which supported so called *wormhole routing*. The first distributed implementation of GHC was built on it [Ichiyoshi *et al.* 1987]. (Distributed implementation means a parallel implementation on a distributed memory hardware). Execution speed was slow (1K LIPS = logical inference per second) because an interpreter system was written in ESP (the system description language of the PSI). However, basic algorithms and techniques of distributed implementation of GHC was investigated in it. Several small parallel programs were written and executed on it for evaluation, and primary experimentations of load balancing were also carried out.

2.3 From GHC To KL1

Since GHC had only basic functions that the kernel concurrent logic language had to support, language extensions were needed for the next more practical system. Kernel language KL1 was designed with considerations of execution efficiency, operating system supports, and some built-in functions [Ueda and Chikayama 1990] [Chikayama 1992]. An intermediate language KL1-B, which was the target language of KL1 compiler, was also designed [Kimura and Chikayama 1987]. In the Multi-PSI/V2 and a PIM model, binary code of KL1-B is directly interpreted by microprogram; that is, KL1-B is machine language itself. In the other PIM models, KL1-B code is converted to lower-level machine instruction sequences and executed by hardware.

2.4 Multi-PSI/V2

The second hardware system was the Multi-PSI/V2 [Takeda *et al.* 1988] [Nakajima 1992], which was improved in performance and functions enough to be called as the first experimental parallel inference machine. It started in operation in 1988 and was demonstrated in the FGCS'88 international conference.

The Multi-PSI/V2 included 64 processors, each of which were equivalent to the CPU of PSI-II [Nakashima and Nakajima 1987], smaller and faster model of the PSI. Processors were connected with two dimensional mesh network with improved speed (10M Bytes/s, full duplex in each channel). KL1-B was the machine language of the system, executed by microprogram. Almost all the runtime functions of KL1 was implemented in microprogram. The KL1 implementation was improved much in execution efficiency, reducing inter-processor communication messages, efficient garbage collections, etc. compared with Multi-PSI/V1. It attained 130K LIPS (in KL1 append) in single processor speed. Table 1 to 4 include specifications of the Multi-PSI/V2. Since 1988, more than 15 systems, large system with 64 processors and small with 32 or 16 processors, have been in operation for parallel software R & D in ICOT and in cooperating companies.

A strong simulator of the Multi-PSI/V2 was also developed for software development environment. It was called the pseudo Multi-PSI, available on the Prolog workstation, PSI-II. A very special feature was caused by similarity of the PSI-II CPU and processing element of the Multi-PSI/V2. Usually, PSI-II executed ESP language with dedicated microprogram. However, it loaded KL1 microprogram dynamically at the activation of the simulator system. The simulator executed KL1 programs as similar speed as that of the Multi-PSI/V2 single processor. Since the PIMOS could be also executed on the simulator, programmers could use the simulator as similar environment as the real Multi-PSI/V2, except for speedup with multiple processors and process scheduling. The pseudo Multi-PSI was the valuable system for initial debugging of KL1 programs.

2.5 Software Development on the Multi-PSI/V2

Parallel operating system PIMOS (the first version) and four small application programs (benchmark programs) [Ichiyoshi 1989] had been developed until FGCS'88. Much efforts was paid in PIMOS development to realize a good environment of programming, debugging, execution and measurements of parallel programs. In the development of small application programs, several important research topics of parallel software were investigated, such as concurrent algorithms with large concurrency without increase of complexity, programming paradigms and techniques of efficient KL1 programs, and dynamic and static load balancing schemes for dynamic and non-uniform computation.

The PIMOS has been improved in several versions, and ported to the PIM until 1992. The small application programs, pentomino [Furuichi *et al.* 1990], best-path [Wada and Ichiyoshi 1990], PAX (natural language parser) and tsume-go (a board game) were improved, measured and analyzed until 1989. They are still used as test and benchmark programs on the PIM.

These development gave observations that the KL1 system on the Multi-PSI/V2 with PIMOS has reached sufficient performance level for practical usage, and has realized sufficient functions for describing complex concurrent programs and for experimentations of software-controlled load balancing.

Several large-scale parallel application programs have been developed from late 1989 [Nitta *et al.* 1992] and still continuing. Some of them have been ported to the PIM.

2.6 Parallel Inference Machine PIM

2.6.1 Five PIM Models

Design of the parallel inference machine PIM was started in concurrent with manufacturing of the Multi-PSI/V2. Some research items in hardware architecture were omitted in the development of the Multi-PSI/V2, because of short development time needed for starting the parallel software development. So, PIM took a greedy R & D plan, focusing both the architectural research and realization of software development environment.

The first trial to the novel architecture was the multiple clusters. A small number of tightly-coupled processors with shared-memory formed a cluster. Many clusters were connected with high speed network to construct the PIM system with several hundred processors. Benefits of the architecture will be discussed in section 3.7.

Many component technologies had to be developed or improved to realize the new system, such as parallel cache memory suitable for frequent inter-processor communications, high speed processors for symbolic processing, improvement of the network, etc. For R & D of better component technologies and their combinations, the development plan of five PIM models was made, so that different component architecture and their combinations could be investigated with assigning independent research topics or roll on each model.

Two models, PIM/p [Kumon *et al.* 1992] and PIM/c [Nakagawa *et al.* 1992], took the multi-cluster structure. They include several hundreds processors, maximum 512 in PIM/p and 256 in PIM/c. They were developed both for the architectural research and software R & D. Each investigated different network architecture and processor structure.

The other two models, PIM/k [Sakai *et al.* 1991] and PIM/i [Sato *et al.* 1992], were developed for the experimental use of intra-cluster architecture. Two-layered coherent cache memory which enabled larger number of processors in a cluster, broadcast-typed coherent cache memory, and a processor with LIW-type instruction set were tested.

The other model, PIM/m [Nakashima *et al.* 1992], did not take the multi-cluster structure, but focused the rigid compatibility with the Multi-PSI/V2, having improved processor speed and larger number of processors. The maximum number of processors will be 256. The performance of a processor will be four to five times larger at peek speed, and 1.5 to 2.5 times larger in average than the Multi-PSI/V2. The processor was similar to the CPU of PSI-UX, the most recent version of the PSI machine. A simulator, pseudo-PIM/m, was also prepared like the pseudo Multi-PSI. The PIM/m targeted the parallel software development machine mostly among the models.

Architecture and specifications of each model will be reported in section 4.

Experimental implementations of some LSIs of these models have started in 1989. The final design was almost fixed in 1990, and manufacturing of whole system was proceeded with in 1991. From 1991 to spring 1992, assembly and test of the five models have carried on.

2.6.2 Software Compatibility

KL1 language is common among all the five PIM models. Except for execution efficiency, any KL1 programs including PIMOS can run on the all models. Hardware architecture is different between two groups, Multi-PSI and PIM/m as the one, and the other PIM models as the other. However, from programmers' view, abstract architecture are designed similar as follows.

The load allocation to processors are fully controlled by programs on the Multi-PSI and the PIM/m. It is sometimes written by programmers directly, and sometimes specified by load allocation libraries. Programmers are often researchers of load balancing techniques. On the other hand, load balancing in a cluster is completely controlled by the KL1 runtime system (not by KL1 programs) among the PIM models with the multi-cluster structure. That is, programmers does not have to think of multiple processors in a cluster, but specify load allocation to each cluster in their programs. It means that a processor of the Multi-PSI or PIM/m corresponds to a cluster of the PIM models with the multi-cluster structure, which simplifies portation of KL1 programs.

2.7 KL1 Implementation for PIM

KL1 system must be the first regular system in the world which can execute large-scale parallel symbolic processing programs very efficiently. Execution mechanisms or algorithms of KL1 language had been developed for distributed memory architectures sufficiently on the Multi-PSI/V2. Some mechanisms and algorithms should be expanded for the multi-cluster architecture of PIM. Ease of porting the KL1 system to four different PIM models was also considered in the language implementation method. Only the PIM/m inherited the KL1 implementation method directly from the Multi-PSI/V2.

To expand the execution mechanisms or algorithms suitable for the multi-cluster architecture, several technical topics were focused, such as avoiding data update contentions among processors in a cluster, automatic load balancing in a cluster, expansion of an inter-cluster message protocol applicable for the message outstripping, parallel garbage collection in a cluster, etc. [Hirata *et al.* 1992].

For easiness of porting the KL1 system to four different PIM models, a common specification of KL1 system "VPIM (virtual PIM)" was written in "C"-like description language "PSL", targeting a common virtual hardware. VPIM was the executable specification of KL1 execution algorithms, which was translated to C language and executed to examine the algorithms. VPIM has been

translated to lower-level machine languages or microprograms automatically or by hands according to each PIM structure.

Preparation of the description language started in 1988. Study of efficient execution mechanisms and algorithms continued until 1991, then, VPIM was completed. Porting the VPIM to four PIM models partially started in autumn 1990, and continued to spring 1992. Now, the KL1 system with PIMOS is available on each PIM model. On the other hand, KL1 system on the PIM/m, which was implemented in microprogram, was made from conversion of Multi-PSI/V2 microprogram by hands or partially in automatic translation. Prior to the other PIM models, PIM/m started in operation with the KL1 system and PIMOS in summer 1991.

2.8 Performance and System Evaluation

Measurements, analysis, and evaluation should be done on various levels of the system shown below.

1. Hardware architecture and implementations
2. Execution mechanisms or algorithms of KL1 implementation
3. Concurrent algorithms of applications (algorithms for problem solving, independent from mapping) and their implementations
4. Mapping (load allocation) algorithms
5. Total system performance of a certain application program on a certain system

Various works have been done on the Multi-PSI/V2. 1 and 2 were reported in [Masuda *et al.* 1988] and [Nakajima 1992]. 3 to 5 were reported in [Nitta *et al.* 1992], [Furuichi *et al.* 1990], [Ichiyoshi 1989] and [Wada and Ichiyoshi 1990].

Primary measurements have just started on each PIM models. Some intermediate results are included in [Nakashima *et al.* 1992] and [Kumon *et al.* 1992].

Total evaluation of the PIM system will be done in the near future, however, some observations and discussions are included in section 6.

3 Characterizing the PIM and KL1 system

PIM and KL1 system have many advantageous features for very efficient parallel execution of large-scale knowledge processing which often shows very dynamic runtime characteristics and non-uniform computation, much different from numerical applications on vector processors and SIMD machines.

This section clarifies the characteristics of the targeted problem domain shortly, and describes the various advantageous features of PIM and KL1 system, that are dedicated for the efficient programming and processing in the problem domain. They will give the total system image and help to clarify the difference and similarity of the system with other large-scale multiprocessors, recently available in the market.

3.1 Summary of Features

The total image of PIM and KL1 system are briefly scanned as follows. Detailed features and their benefits, and reasons why they were chosen are presented in the following sections.

Distributed memory MIMD machine: Global structure of the PIM is the distributed memory MIMD machine in which hundreds computation nodes are connected by highspeed network. Scalability and ease of implementations are focused. Each computation node includes single processor or several tightly-coupled processors, and large memory. Processors are dedicated for efficient symbolic processing.

Logic programming language: The kernel language KL1 is a concurrent logic programming language, which is single language for system and application descriptions. Language implementation and hardware design are based on the language specification.

KL1 is not a high-level knowledge representation language nor a language for certain type of reasoning, but a general-purpose language for concurrent and parallel programming, especially suitable for symbolic computations.

KL1 has many beneficial features to write parallel programs in those application domains, described below.

Application domain: Primary applications are large-scale knowledge processing and symbolic computation. However, large numerical computation with dynamic features, or with non-uniform data and non-uniform computation (non-data-parallel computation) are also targeted.

Language implementation: One KL1 system is implemented on a distributed memory hardware, which is not a collection of many KL1 systems implemented on each processing node. A global name space is supported for code, logical variables, etc. Communication messages between computation nodes are handled implicitly in KL1 system, not by KL1 programs. An efficient implementation for small-grain concurrent processes is taken.

These implementations focus to realize the beneficial features of KL1 language for the application domains described before.

Policy of load balancing: Load balancing between computation nodes should be controlled by KL1 programs, not by hardware nor by the language system automatically. Language system has to support enough functions and efficiency for the experiments of various loadbalancing schemes with software.

3.2 Basic Choices

(1) Logic programming: The first choice was to adopt logic programming as the basis of the kernel language. The decision is mainly due to the insights of ICOT founders, who expected that logic programming was suitable for both knowledge processing and parallel processing. A history, from vague expectations on logic programming to the concrete design of the KL1 language, is explained in [Chikayama 1992].

(2) Middle-out approach: A middle-out approach of R & D was taken, placing the KL1 language as the central layer. Based on the language specification, design of the hardware and the language implementation started downward, and writing the PIMOS operating system and parallel software started upward.

(3) MIMD machine: The other choices concerned with basic hardware architecture.

Dataflow architecture before mid 1980 was considered not providing enough performance against hardware costs, according to observations for research results in initial stage of the project.

SIMD architecture seemed inefficient on applications with dynamic characteristics or low data-parallelism that are often seen in knowledge processing.

MIMD architecture remained without major demerits and was most attractive from the viewpoint of ease of implementation with standard components.

(4) Distributed memory structure: Distributed memory structure is suitable to construct very large system, and easy to implement.

Recent large-scale shared memory machines with directory-based cache coherency mechanisms claims good scalability. However, when the block size (the coherency management unit) is large, the interprocessor communication with frequent small data transfer seems inefficient. KL1 programs require the frequent small data transfer. When the block size becomes small, large directory memory is needed, which increases the hardware cost.

Single assignment languages need special memory management such as dynamic memory allocation and garbage collection. These management should be done as locally as possible for the sake of efficiency. Local garbage collection requires separation of local and global address spaces with some indirect referencing mechanism or address translation, even in a scalable shared memory architecture. Merits of the low-cost communication in the shared memory architecture decrease significantly for such the case.

These are the reasons to choose the distributed memory structure.

3.3 Characterizing the Applications

(1) Characterization: Characteristics of knowledge processing and symbolic computation are often much different from those of numerical computation on vector processors and SIMD machines. Problem formalizations for those machines usually based on data-parallelism, parallelism for regular computation on uniform data.

However, the characteristics of knowledge and symbolic computations on parallel machines tend to be very dynamic and non-uniform. Contents and amount of computation vary dynamically depending on time and space. For example, when a heuristic search problem is mapped on a parallel machine, workload of each computation node changes drastically depending on expansion and pruning of the search tree. Also, when a knowledge processing system is constructed from many heterogeneous objects, each object arises non-uniform computation. Computation loads of these problems are hardly estimated before execution.

Some classes of large numerical computation without data-parallelism also show the dynamic and non-uniform characteristics.

Those problems which has dynamism and non-uniformity of computation are called *the dynamic and non-uniform problems* in this paper, implying not only the knowledge processing and symbolic computation but also the large numerical computation without data-parallelism.

The dynamic and non-uniform problems tends to include the programs with more complex program structure than the data-parallel problems.

(2) Requirements for the system: Most of the software systems on recent commercial MIMD machines with hundreds of processors target the data-parallel computation, but they almost don't care other paradigms.

The dynamic and non-uniform problems arise new requirements mainly on software systems and a few on hardware systems, which are listed below.

1. Descriptive power for complex concurrent programs
2. Easy to remove bugs
3. Ease of dynamic load balancing
4. Flexibility for changing the load allocation and scheduling schemes to cope with difficulty on estimating actual computation loads before execution

3.4 Characterizing the Language

This subsection itemizes several advantageous features of KL1 that satisfy the requirements listed in the previous section. Features and characteristics of the concurrent logic programming language KL1 are described in detail in [Chikayama 1992].

The first three features have been in GHC, the basic specifications of KL1. These features make descriptive power of the language large enough to write complex concurrent programs. They are the features of *concurrent programming* to describe logical concurrency, independent from mapping to actual processors.

(1) **Dataflow synchronization:** Communication and synchronization between KL1 processes are performed implicitly at all within a framework of usual unification. It is based on the dataflow model. Implicitness is available even in a remote synchronization. The feature drastically reduces bugs of synchronization and communication compared with the case of explicit description using separate primitives. The single-assignment property of logic variables supports the feature.

(2) **Small-grain concurrent processes:** The unit of concurrent execution in KL1 is each body goal of clauses, which can be regarded as a process invocation. KL1 programs can thus involve a large amount of concurrency implicitly.

(3) **Indeterminacy:** A goal (or process) can test and wait for the instantiation of multiple variables concurrently. The first instantiation resumes the goal execution, and when a clause is committed (selected from clauses that succeed to execute guard goals), the other wait conditions are thrown away. This function is valuable to describe "non-rigid" processing within a framework of side-effect free language. Speculative computation can be dealt with, and dynamic load distribution can be also written.

The next features have been included in KL1 as extensions to GHC. (4) was introduced to describe mapping (load allocation) and scheduling. They are the features for *parallel programming* to control actual parallelism among processing nodes. (5) is prepared for operating system supports. (6) is for the efficiency of practical programs.

(4) **Pragma:** Pragma is a notation to specify goal allocation to processing nodes or specify execution priority of goals. Pragma doesn't affect the semantics of a program, but controls parallelism and efficiency of actual parallel execution. Pragmas are usually attached to goals after making sure that the program is correct anyway. It can be changed very easily because it is syntactically separated from the correctness aspect of a program.

Pragma for load allocation: Goal allocation is specified with a pragma, @node(X). X can be calculated in programs. Coupled with (1) and (2), the load allocation pragma can realize very flexible load allocation. Also coupled with (3) and the pragma, KL1 can describe a dynamic load balancing program within a framework of the pure logic programming language without side-effect. Dynamic load balancing programs are hard to be written in pure functional languages without indeterminacy.

Pragma for execution priority: Execution priority is specified with a pragma, @priority(Y). More than thousands priority levels are supported to control goal scheduling in detail, without rigid ordering.

Combination of (3) and the priority pragma realizes the efficient control of speculative computations. Large number of priority levels can be utilized in e.g. parallel heuristic search to expand good branch of the search tree at first.

(5) **Shoen function (meta-control for goal group)** The *shoen* function is designed to handle a set of goals as a task, a unit of execution and resource management. It is mainly used in PIMOS. Start, stop and abortion of tasks can be controlled. Limit of resource consumption can be specified. When errors or exception conditions occur, the status are frozen and reported outside the *shoen*.

(6) **Functions for efficiency:** KL1 has several built-in functions or data types whose semantics is understood within the framework of GHC but which has been provided for the sake of efficiency. Those functions hide demerits of side-effect free languages, and also avoid an increase of computational complexity compared with sequential programs.

3.5 Characterizing the Language Implementation

Language features, just described in the previous section, satisfy the requirements for a system by *the dynamic and non-uniform problems* discussed in section 3.3. Most of special features of the language implementation focused to enlarge those advantageous features of KL1 language.

(1) **Implicit communication:**
Communication and synchronization among concurrent processes are implicitly done by unifications on shared logical variables. They are supported both in a computation node and between nodes. It is especially beneficial that a remote synchronization is done implicitly as well as local.

A process (goal) can migrate between computation nodes only being attached a pragma, @node(X). When the process has reference pointers, remote references are generated implicitly between the computation nodes. The remote references are used for the remote synchronizations or communications.

These functions hide the distributed memory hardware from the "concurrent programming". That is, programmers can design concurrent processes and their communications, independent from their allocations to a same computation node or different nodes. Only the "parallel programming" with pragmas, a design of load allocation and scheduling, has to concern with hardware structure and network topology.

Implementation features of those functions are summarized below, including the features for efficiency.

- Global name space on a distributed memory hardware — in which implicit pointer management among computation nodes are supported for logical variables, structured data and program code
- Implicit data transfer caused by unifications and goal (process) migration
- Implicit message sending and receiving invoked with data transfer and goal sending, including message composition and decomposition
- Message protocols able to reduce the number of messages, and also protocols applicable to message outstripping

(2) **Small-grain concurrent processes:** Efficient implementation of small-grain concurrent processes are realized, coupled with low-cost communications and synchronizations among them.

Process scheduling with low-cost suspension and resumption, and priority management are supported. Efficient implementation allows actual use of a lot of small-grain processes to realize large concurrency. A large number of processes also gives flexibility for the mapping and load balancing.

Automatic load balancing in a cluster is also supported. It is a process (goal) scheduling function in a cluster implemented with priority management. The feature hides multiprocessors in a cluster from programmers. They do not have to think about load allocation in a cluster, but only have to prepare enough concurrency.

(3) **Memory management:** These garbage collection mechanisms are supported.

- Combination of incremental garbage collection with subset of reference counting and stop-and-collect copying garbage collection
- Incremental releasing of remote reference pointers between computation nodes with weighted reference counting scheme

Dynamic memory management including garbage collections looks essential both for symbolic processing and for parallel processing of *the dynamic and non-uniform problems*. Because the single assignment feature, strongly needed for the problems, requires dynamic memory allocation and reclamation.

Efficiency of garbage collectors is one of key features for practical language system of parallel symbolic processing.

(4) **Implementation of shoen function:** *Shoen* represents a group of goals (processes) as presented in the previous subsection. *Shoen* mechanism is implemented not only in a computation node but also among nodes. Namely, processes in a task can be distributed among computation nodes, and still controlled all together with *shoen* functions.

(5) **Built-in functions for efficiency:** Several built-in functions and data types are implemented to keep up with the efficiency of sequential languages.

(6) **Including OS kernel functions:** Figure 2 shows the relation of KL1 implementation and operating system functions. KL1 implementation includes so called OS kernel functions such as memory management, process management and scheduling, communication and synchronization, virtual single name space, message composition and decomposition, etc. While, PIMOS includes upper OS functions like programming environment and user interface.

The reason why the OS kernel functions are included in the KL1 implementation is that the implementation needs to use those functions with as light cost as possible. Cost of those functions affect the actual

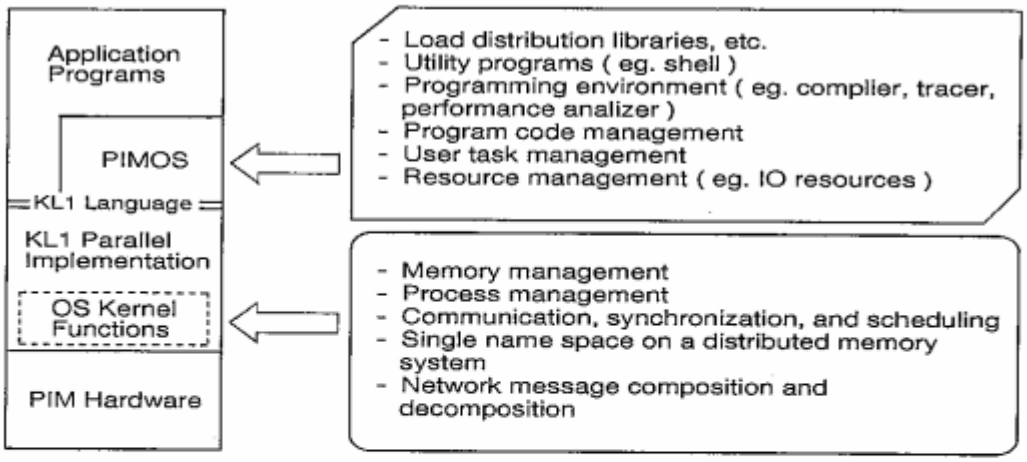

Figure 2: KL1 Implementation and OS Functions

execution efficiency of the advantageous features of KL1 language, such as large number of small-grain concurrent processes, implicit synchronization and communication among them (even between remote processes), indeterminacy, scheduling control with large number of priority levels, process migration specified with pragmas, etc. Those features are indispensable for concurrent and parallel programming and efficient parallel execution of large-scale symbolic computation with dynamic characteristics, or large-scale non-data-parallel numerical computations.

Considering a construction of similar purpose parallel processing system on a standard operating system, interface level to the OS kernel may be too high (or may arise too much overhead). Some reconstruction of OS implementation layers might be needed for the standard parallel operating systems for those large-scale computation with dynamic characteristics.

3.6 Policy of Load Balancing

Such a basic policy has been taken that load balancing between computation nodes should be completely controlled by KL1 programs, not by hardware nor by language system automatically. There are two reasons.

One is that KL1 can describe load balancing programs within usual logic programming features. Since many research topics on load distribution have been remained unsolved especially on dynamic problems, experiments on software controlled load balancing is advantageous in an aspect of flexibility. It does not include significant overhead because the KL1 language system realize a very low-cost implementation.

The other is that distributed memory architecture needs strong locality of computation, for which some programmers' help is important for better load balancing.

Language system has to support enough functions and efficiency for the experiments of various load balancing schemes by software.

Some load balancing schemes are prepared as utility programs, available for application programmers.

3.7 Characterizing the Hardware Architecture

Features of PIM hardware architecture are listed below. Some of them are specialized for symbolic processing and large-scale parallel computation of dynamic problems, and some of them are standard.

(1) **Distributed memory MIMD machine:** Target hardware is the large-scale MIMD machine with distributed memory structure. Hundreds processing nodes are connected by highspeed network. It was a basic choice of the R & D. The structure was considered to have large scalability, to be mostly easy for implementation, and to be suitable to separate local garbage collections and global.

(2) **Cluster structure:** Eight processors, that are tightly coupled with shared bus and shared memory, form a cluster. Many clusters are connected with highspeed network to form the total system. Programmers deal with a cluster as a computation node with large computation power and large memory, since automatic load balancing is supported by language system within a cluster.

Cluster is a substructure of the PIM, realizing a low latency and high bandwidth connection between processors. There are two major advantages of

the cluster structure. The first is its applicability to those problems which have less locality, while distributed memory architecture hardly processes those problems efficiently. The second is higher efficiency of memory usage compared with full distributed memory systems with the same memory size. A substructure with higher bandwidth inter-processor connection is effective to reduce needs of memory size per processor, keeping the same efficiency of parallel processing. It affects the total system cost significantly.

A disadvantage is heterogeneous inter-processor connections that increase the complexity of hardware implementations, however, the cluster with tightly coupled processors will be a standard component in the near future.

(3) **Large memory against processing power:** Non-uniform computation or dynamic computation with wide variation of grain size require larger memory to keep the processing efficiency, compared with data-parallel computation. Because extra work is needed to fill the idling time caused by irregular synchronization, which requires more working space in a memory.

(4) **Highspeed network:** Highspeed network connection between processing nodes has already become standard. However, the ratio of network load and processor load, caused by network communications, is different from the case of numerical processing. Management of virtual single name space usually arises extra processor loads for each communications, compared with the case of simple data transfer in numerical processing. It causes less needs to network bandwidth against processing power.

On the other hand, parallel symbolic computation with dynamic features often arises remote synchronizations with small data transfer. Response of the network communication is more important than bandwidth for such cases.

(5) **Coherent cache memory:** Each processor in a cluster has coherent cache memory with write back strategy. Basic technology is similar to the standard coherent cache memory used in commercial tightly coupled multiprocessors. However, the occurrence of cache to cache data transfer, caused by inter-processor communications, is larger than the usual time sharing use of commercial multiprocessors. Optimizations of cache commands and bus protocols for such usage is important to reduce bus traffic.

(6) **Dedicated processors:** Processors include special features of tag handling, data type checking and branching, and dereferencing pointers for efficient KL1 execution. These features are useful not only for symbolic processing, but also for an efficient implementation of a single-assignment language needed for the parallel processing of *the dynamic and non-uniform problems.*

The processors have dedicated instruction sets derived from the abstract instruction set KL1-B.

Pipelining and RISC-like instruction sets are also used, that are standard techniques.

4 Machine Architecture and Hardware

Overall structure and features of the PIM system were presented in the previous section. This section shows the machine architecture, hardware implementations and some technical data of each PIM models in detail.

4.1 Overview of Five PIM Models

Five PIM models have been developed, that have different architectures or different combinations of component technologies, and have different rolls of R & D.

PIM/p : PIM/p is the largest PIM model which contains maximum 512 processors. PIM/p focuses both architectural research and actual use in software R & D.

PIM/p took the multi-cluster architecture shown in Figure 3. Maximum 64 clusters can be connected. Connection network took hypercube topology. Two independent networks are connected to each clusters.

Each cluster contains eight processors connected with a shared bus and shared memory. A processor has coherent cache memory, a network interface unit "NIU", and an I/O device interface (SCSI bus) [Kumon *et al.* 1992].

Processors in all PIM models have SCSI buses, which are used to connect FEPs (Front End Processors) and hard disks. The PSI-UX [Nakashima *et al.* 1992] is used for the FEP, as an intelligent I/O device for human-machine interface.

PIM/m : PIM/m targets the software development machine and rigid compatibility with the Multi-PSI/V2. 256 processors are connected with two dimensional mesh network. The structure is shown in Figure 4. 32 hard disks, which are 20GB in total, and many FEPs are connected [Nakashima *et al.* 1992].

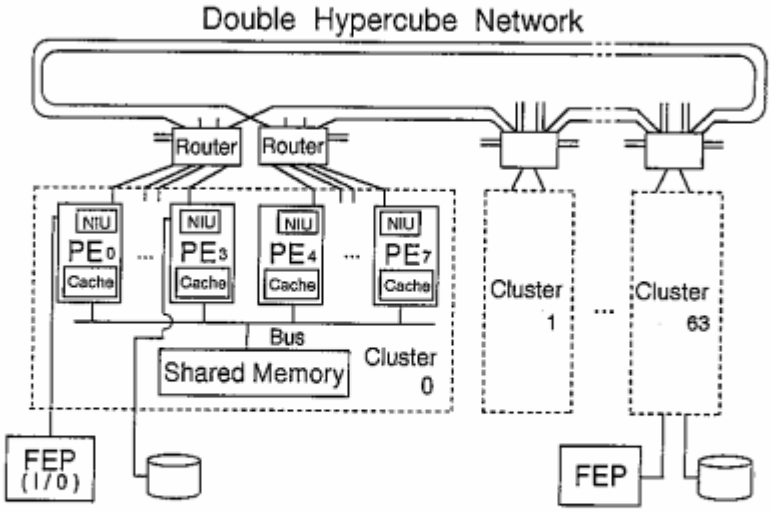

Figure 3: Overview of PIM/p Architecture

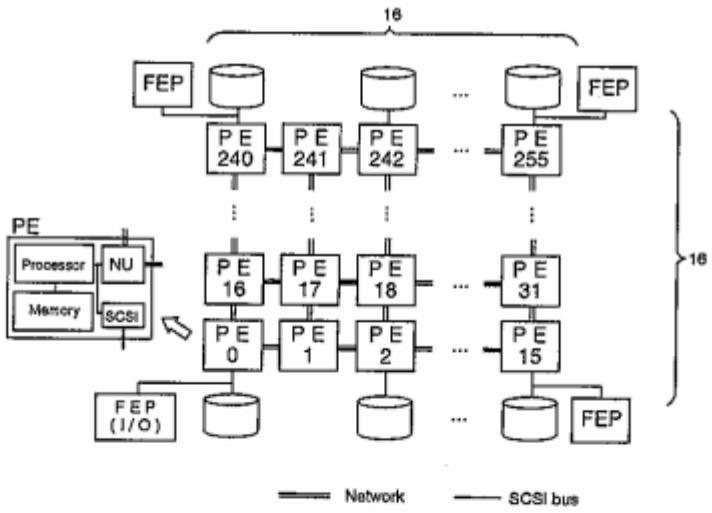

Figure 4: Overview of PIM/m Architecture

PIM/c : PIM/c also takes the multi-cluster architecture including 256 processors in total. A cluster contains eight processors. 32 clusters are connected with a crossbar switch network [Nakagawa *et al.* 1992].

PIM/k : PIM/k focuses on architectural research within a cluster. Hierarchical cache system has been investigated to connect larger number of processors in a cluster [Sakai *et al.* 1991]. Four processors share a local bus and second cache. They form a mini-cluster. Four mini-clusters are connected to a shared memory-bus and shared memory (Figure 5).

PIM/i : PIM/i is also a research use system. LIW-type instruction set and cache protocol with broadcasting type has been investigated [Sato *et al.* 1992].

The global configuration of five PIMs are summarized in table 1.

Specifications of components, that are processors, networks, and cache systems, will be reported in the following subsections.

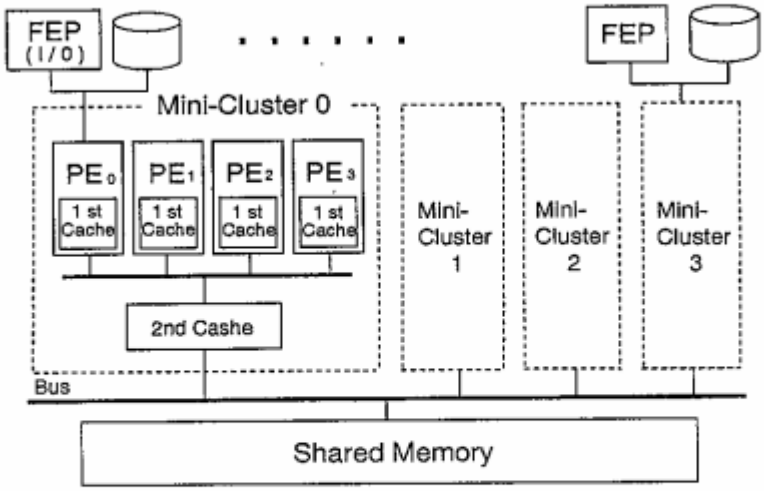

Figure 5: Overview of PIM/k Architecture

Table 1: Global Configuration

	Topology	Number of Clusters	Total Number of PEs	Memory Size/Cluster
PIM/p	hypercube × 2	64	512	256 MB
PIM/m	mesh	256	256	80 MB
PIM/c	crossbar	32	256	160 MB
PIM/k	—	1 †	16	1 GB
PIM/i	—	2	16	320 MB
Multi-PSI/V2	mesh	64	64	80 MB

(† : four mini-clusters included)

4.2 Processing Element

Since KL1 implementation requires frequent runtime type checking, all CPUs of PIM models are designed as the tagged-architecture similar to the Multi-PSI.

PIM/p, PIM/i and PIM/k have RISC-like instruction set whereas PIM/m and PIM/c have CISC-like micro programmable instruction set (Table 2). The former processors execute machine instructions which are at a level still lower than KL1-B. The latter processors interpret KL1-B code by horizontal micro program.

The CPU of PIM/p [Kumon *et al.* 1992] has a unique feature called *macro-call* [Shinogi *et al.* 1988] instructions for light-weight subroutine calls. The instructions enable the size of compiled user program codes to be kept small and to reduce the overheads of subroutine calls. It also has some more instructions dedicated to KL1 implementation, such as dereference instructions and MRB [Chikayama and Kimura 1987] incremental garbage collection instructions. The CPU takes four-stage pipeline structure.

The CPU of PIM/m [Nakashima *et al.* 1992] is a microprogram controlled processor with five-stage pipelining. The instruction set is KL1-B itself, which is binary compatible with Multi-PSI/V2. Sophisticated data type checking and the automatic dereference mechanism are special features.

The CPU of PIM/i tries the LIW(long instruction word)-type instruction set.

4.3 Network

Networks are summarized in table 3.

In PIM/p, each processor has a NI and four NIs are connected to a router. The router works as a node in the network. There are two hypercube networks to attain large band width.

PIM/m has a two dimensional mesh network, similar to the Multi-PSI. The networks of PIM/p and PIM/m realize so-called the worm-hole routing.

Table 2: Specification of Processing Element

	Instruction set	Cycle time	LSI fabrication	Line interval
PIM/p	RISC + macro instruction	60 nsec †	standard-cell	0.96 μm
PIM/m	CISC (micro programmable)	65 nsec	standard-cell	0.8 μm
PIM/c	CISC (micro programmable)	50 nsec †	gate-arrays	0.8 μm
PIM/k	RISC	100 nsec	custom	1.2 μm
PIM/i	RISC	100 nsec †	standard-cell	1.2 μm
Multi-PSI/V2	CISC (micro programmable)	200 nsec	gate-arrays	2.0 μm

(† are design specifications. They are under testing with longer cycle time.)

Table 3: Network

	# PEs in a cluster	# NIs in a cluster	Transfer Rate †
PIM/p	8	8	33 MB/sec ‡ ×2
PIM/m	1	1	8 MB/sec
PIM/c	8	1	40 MB/sec ‡
PIM/k	16	—	—
PIM/i	8	1	—
Multi-PSI/V2	1	1	10 MB/sec

(PE = processing element, NI = network interface)
(†: per channel, full duplex ‡: design specifications)

PIM/c has one special processor named cluster controller in each cluster. The cluster controller is connected to a shared bus and works as a network interface to a crossbar network. The cluster controller has overall responsibility for network communications.

4.4 Cache System

Since KL1 programs arise asynchronous communications among processors very frequently, shared bus traffic tends to become very heavy. To solve this problem, an optimized coherent cache protocols were designed [Goto *et al.* 1989][Matsumoto *et al.* 1987], which can keep the locality high and reduce the shared bus traffic [Nishida *et al.* 1990]. All PIMs have write-back type coherent cache protocols (Table 4). Low cost locking mechanisms are also supported with utilizing the cache block status.

5 KL1 Language Implementation

KL1 language has many beneficial features to write efficient concurrent and parallel programs of *the dynamic and non-uniform problems*, which was explained in section 3.4. The KL1 implementation is focused to realize the execution efficiency of those language features. This section looks at the language implementation methods and techniques briefly, that correspond to the implementation features presented in section 3.5. The purpose of this section is to give a concrete image of several key features of the KL1 implementation. Detailed information are presented in [Hirata *et al.* 1992] [Nakajima 1992].

5.1 Execution Model of KL1

For the help of getting the image, the execution model of KL1 is shown briefly. KL1 program is made up of a collection of clauses, whose form is:

$$\underbrace{H :- G_1, ..., G_m}_{\textit{guard part}} \mid \underbrace{B_1, ..., B_n.}_{\textit{body part}}$$

where H is the *head*, G_i the *guard goal*, that are collectively called the *guard part*. The B_i are the *body goals* and the vertical bar (|) is the *commitment operator*.

The guard part can be considered as a pattern match and condition tests. If there are alternative clauses, their guard parts are tested sequentially. When a clause succeeds the pattern match and the condition tests, the clause commits. The caller goal is reduced to the body

Table 4: Specification of Cache System

	Coherence Control		Mapping	Cache Size	
	Protocol	# States †		Instruction	Data
PIM/p	invalidation	4	4 way	64 KB	
PIM/m	—	—	direct-map	5 KB	20 KB
PIM/c	invalidation	5	2 way	80 KB	
PIM/k	hierarchical invalidation	4	(1st) direct-map	128 KB	256 KB
			(2nd) 4 way	1 MB	4 MB
PIM/i	broadcasting	6	direct-map	160 KB	160 KB
Multi-PSI/V2	—	—	direct-map	20 KB	

(† does not include *locking* state.)

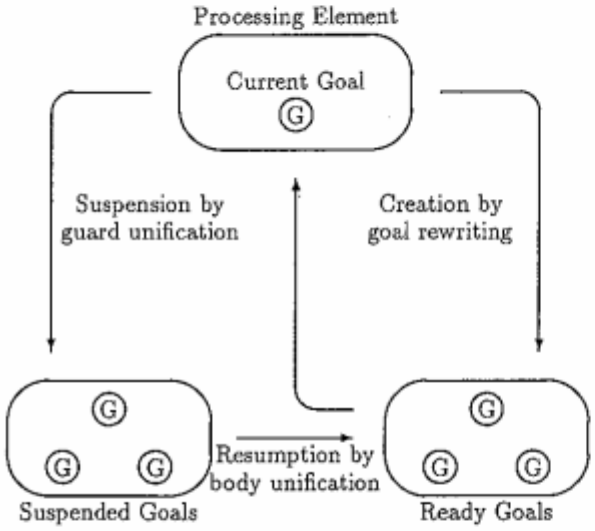

Figure 6: Execution Model of KL1

goals of the committed clause. These body goals are executed concurrently (AND-parallel). A KL1 clause can be considered as a rewrite rule, which rewrites the caller goal to the body goals.

An execution model of KL1 is shown in Figure 6. There is a goal pool which holds the ready goals to be rewritten. One of ready goals is taken from the goal pool for the execution, which is the current goal. When there is a clause, which matches the current goal and succeeds the condition tests, the current goal is rewritten. The rewritten goals are placed back to the goal pool.

Goals may have common variables, that are used for the communication and synchronization. Let us assume that there are two goals sharing a logical variable. A body unification, produced in a goal rewriting, can instantiate the variable. Guard unifications, that appear in a execution of the other goal, test the instantiated value of the variable. This is the communication between the goals. When the variable is not instantiated before the guard unification, and no other clause can commit, the current goal is suspended. Instantiation of the variable resumes the suspended goal. This is the synchronization [Ueda and Chikayama 1990].

5.2 Supports for the Implicit Communication

There are several important mechanisms that realize the implicit communication between computation nodes.

Let us assume that there are two goals sharing a variable in a computation node. Each goal has a reference to the variable. When a goal is sent to the other computation node, a remote reference has to be generated implicitly. The implicit communication between the goals in the different nodes will be performed along with this remote reference.

The important mechanisms are shown briefly.

5.2.1 Global Name Space

The implicit reference management across the computation nodes are supported for logical variables, structured data and program code. It is a support of the virtual global name space on a distributed memory hardware.

The **export/import tables** realize the feature. The export/import tables are the indirect reference tables that separate the local address space in a computation node and the global space for the remote references (Figure 7). The remote reference (external reference) is identified by the pair (A,e), where A is the node number in which the referenced data resides, and e is the entry number of the export table. Registration to the tables are performed dynamically when a new remote reference is made [Ichiyoshi *et al.* 1987].

The entry number e does not change even when a local garbage collection occurs which moves the location of the exported cell. When a **duplicated exportation/importation** occurs, the same table entry number is used (reducing a new registration to the table)

which eliminates useless data transfer between nodes [Ichiyoshi *et al.* 1988].

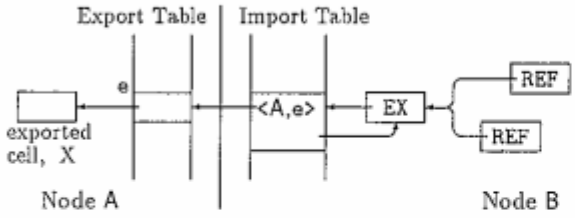

Figure 7: Export and Import Tables

5.2.2 Implicit Data Transfer

Data Transfer by Unifications: The implicit data transfer between computation nodes is initiated by unifications.

A guard unification tries to test an instantiation of a logical variable. When it is an external reference (`EX` in Figure 7), a read request message, `%read(X, ReturnAddress)`, is sent to the node A. Where `X` is the external reference ⟨A,e⟩, and `ReturnAddress` is a newly created export table entry in the node B.

The goal execution, which initiated the guard unification, is suspended when no other clause can commit.

When the referenced cell has a concrete value `V`, it is returned by the message, `%answer_value( ReturnAddress, V )`. The message resumes the suspended goal, which waits for the value `V`. If the referenced cell is not bound to a fixed value, the read request is suspended until the variable is instantiated.

When a body unification tries to unify a remote cell X with a term Y, a message `%unify(X, Y)` is sent to the referenced cluster. When Y is an atomic data or a structure, a simple data transfer occurs.

The unifications between two uninstantiated variables in different clusters may make reference loops between clusters. This problem can be solved by controlling the direction of reference pointers [Ichiyoshi *et al.* 1988].

Lazy Transfer: When a structured data is transferred between nodes, one-level transfer is performed. The components of a structure may be atomic data or nested structures. The atomic data are copied and transferred directly, while the nested structures are remained as pointers and transferred as external references. This is called the one-level transfer. The policy is that the data transfer should be delayed as lazily as possible, until the data is really needed for some operation.

Code Transfer: Program codes are handled as large structured data. They are loaded on one cluster by a loader program at first. Any KL1 goal hold the reference to the corresponding code object. When a goal is sent to a cluster and the cluster does not contain the corresponding code object, the goal execution is suspended and the code is dynamically transferred from the cluster which is pointed by the external reference held in the goal.

5.3 Small-Grain Concurrent Processes

5.3.1 Process Group Management

KL1 goals can be considered as lightweight processes. For the efficient parallel processing, a user task have to include a lot of lightweight processes. It is needed for the parallel operating system that a group of goals (lightweight processes) can be handled all together as a task. The **shoen** supports the meta control facilities of execution control, resource management and status monitoring for the goal group.

Shoen and Foster Parent: Any goals have to belong to a certain *shoen*. The *foster-parent* `fp` is a proxy *shoen*, which is created in every computation nodes where the goals of the *shoen* are executed. Each goal points their foster-parent in the node, and test the request for meta-controls in a certain interval (e.g. in every goal reductions). Figure 8 shows the relationship among *shoens*, foster-parents and goals.

A *shoen* and a foster-parent keep their environments, such as status, resources, and the number of goals. Foster-parents reduce the communication between each goal and their *shoen*, to avoid an access bottleneck at the *shoen*.

Termination Detection: The termination detection of a goal group is one of the difficult subjects in parallel computation systems, especially when messages may be in transit on the network. Even if all the foster parents report their terminations, the *shoen* should not terminate when there are goals in transit.

One of the solutions is the *Weighted Throw Counting* (WTC) scheme [Rokusawa *et al.* 1988], which is an application of the *Weighted Reference Counting* (WRC) scheme [Watson and Watson 1987].

5.3.2 Goal Scheduling

The goal scheduling, discussed here, is a different concept with the goal group management by *shoen*. The goal scheduling is the state transition management of each goals, among *ready*, *execution*, and *suspension* states. Execution priority is also managed.

Basic Goal Scheduling Scheme: The *ready* goals in a computation node are linked into a list forming a *ready-goal-stack*. In principle, a current goal is popped from the

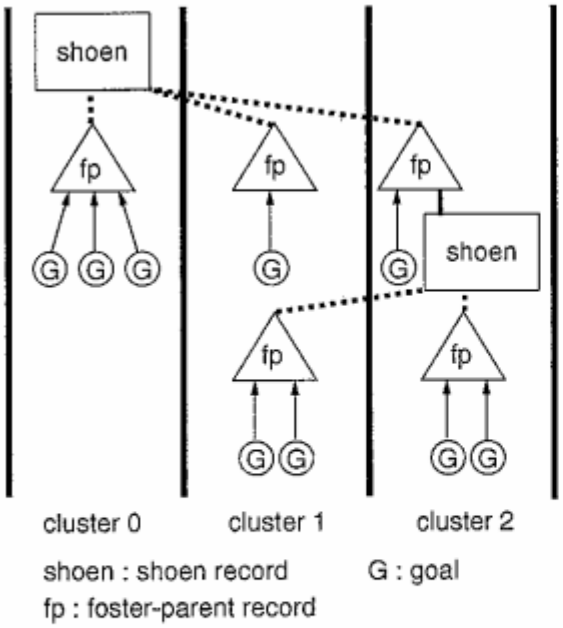

Figure 8: Relationship of *shoen* and foster-parents

ready-goal-stack, then the goal rewriting is performed. The rewritten goals are pushed to the ready-goal-stack, which is the **depth-first scheduling** in a computation node.

When any unification suspends, the goal is linked as a suspended goal to the variable which caused the suspension. Here, the **non-busy waiting** method has been adopted. That is, the suspended goal is not scheduled until the variable will be instantiated. When a suspended goal is resumed, it is linked to the ready-goal-stack again.

Execution priority of goals can be specified by pragmas. The ready-goal-stack is managed with the priority of goals.

Goal Distribution within a Cluster: An automatic load balancing scheme is tried within a cluster. An individual ready-goal-stack is provided for the highest priority goals in each processing element, to avoid conflicts of access to the common goal-stack [Sato *et al.* 1987]. The highest-priority goals are distributed to keep the processor loads in good balance [Hirata *et al.* 1992].

Inter-cluster Goal Distribution: A body goal, `goal@node(CL)`, is thrown with a message `%throw` to a node `CL` when the clause commits. The node (more precisely, a certain processing element in the cluster `CL`), that received the `%throw` message, links the goal to its ready-goal-stack as well as to the foster-parent. If there is no foster-parent, one will be created on the spot.

5.4 Memory Management

Memory management like dynamic memory allocation, reclamation, and garbage collection are indispensable for concurrent symbolic processing languages.

5.4.1 Incremental Garbage Collection by MRB

The MRB method is a subset of the reference counting scheme which maintains one-bit information in pointers indicating whether the pointed data object has multiple references to it or not [Chikayama and Kimura 1987] [Inamura *et al.* 1988]. Garbage cells that have only a single reference can be reclaimed incrementally.

The MRB is also useful to optimize the updating of structured data. Structured data must be copied in principle when it is updated partially, because of the single-assignment feature. However, it can be rewritten destructively when the structure has only a single reference, keeping a semantics of the single-assignment language.

5.4.2 Garbage Collection within a Cluster

Another garbage collection is implemented, which is performed locally within a cluster accompanied with the incremental garbage collection by MRB. Because the MRB scheme leaves some garbages.

So-called *stop and copy* scheme is adopted basically. The parallel mechanism has been investigated to collect garbages by all processing elements in parallel in a cluster [Imai and Tick 1991].

5.4.3 Inter-Cluster Garbage Collection by WEC

An incremental inter-cluster garbage collection scheme, the weighted export counting (WEC) scheme is employed [Ichiyoshi *et al.* 1988]. It is an application of the weighted reference counting (WRC) scheme [Watson and Watson 1987]. The scheme has several advantages. One is the incremental garbage collection capability with fewer message exchanges compared with the full reference counting. The other is also a capability of reducing the messages for the case when a imported data has to be exported again to the different clusters.

5.5 Abstract Instruction Set KL1-B

KL1-B is the abstract instruction set which is common in PIM models. The role of KL1-B is similar to that of WAM [Warren 1983]. An explanation of each KL1-B instruction can be found in [Kimura and Chikayama 1987].

Most of the KL1 implementation schemes, presented in previous sections, are realized as runtime routines that are invoked by certain KL1-B instructions implicitly.

The KL1 compiler for PIM has two phases. The first phase compiles a KL1 program into an KL1-B code. The second phase translates the KL1-B code into a native code, making a linkage with runtime routines.

6 Measurements and Evaluation

This section describes some measurements results and evaluations for the parallel inference machines and the language system. The measurements focused on a low-cost implementation of small-grain concurrent processes and remote synchronization and communication. Measurements on a few benchmark programs are also reported, including the most recent measurements on PIM/m.

6.1 Measurements and Evaluation on the Multi-PSI/V2

The KL1 language implementation includes so-called OS kernel functions, as shown in section 3.5. Most of the implementation features, that were presented in section 5, concern with the OS kernel functions. Efficient implementations of these functions enable the actual use of the beneficial features of KL1 language (presented in section 3.4) to write efficient parallel programs of *the dynamic and non-uniform problems* for large-scale parallel machines.

The actual execution cost of some of these functions have been measured on the Multi-PSI/V2. Goal scheduling cost within a computation node, communication cost between nodes, and communication overhead in benchmark programs are reported. Measurements results shows the quite low-cost implementations.

Note that the Multi-PSI/V2 has a mesh structure with 64 processing elements (PEs). There are 64 computation nodes each of which is one PE.

6.1.1 Goal Scheduling Cost in a Node

Goal scheduling and synchronization cost within a processing element (PE) have been measured [Onishi *et al.* 1990].

The **enqueue and dequeue** cost of a simplest goal is 5.4 μs (27 micro-instruction steps). When a goal is rewritten to several goals in a goal reduction, they are pushed on the ready-goal-stack once (except for one goal which can be executed directly). The enqueue and dequeue cost is the summation of the pushing and popping cost of a goal to the ready-goal-stack. The enqueue and dequeue cost can be considered as a part of the *process fork cost*.

The **single-suspension** cost of a simple goal is 14 μs (70 steps). When a goal is suspended waiting for a variable instantiation, the goal is hooked to the variable cell. When the variable is instantiated, the goal becomes executable and is pushed on the ready-goal-stack. The single-suspension cost is a summation of the hook, enqueue, and dequeue cost. The single-suspension cost can be considered as the *synchronization cost* between processes in a processor.

The **two-way multiple-suspension** cost of a simple goal is 28 μs (140 steps). A goal can wait for the variable instantiation of several different variables. The first instantiation resumes the goal execution. If the instantiation causes a comitment of a clause, the other waiting conditions are thrown away. The two-way multiple-suspension is a case of two variables. The feature is a combination of the indeterminacy and the synchronization. Cost increase from the single-suspension corresponds to the implementation cost of the *indeterminacy*.

These low-cost implementations encourage the actual use of a lot of small-grain processes. These costs of the goal scheduling also give a guideline for the lower bound of process grain size for efficient execution within a computation node.

6.1.2 Communication Cost Between Nodes

Cost of the communication primitives have been measured on the Multi-PSI/V2 system [Nakajima and Ichiyoshi 1990]. A goal sending to another PE (a remote call of a lightweight process) is realized by `%throw_goal` message. Inter-PE reading of values (used for remote synchronization and communication) is realized by `%read` & `%answer_value` protocols.

Figure 9 shows the cost of handling those three messages at both sending and receiving PE.

The cost is broken down into three parts. `Encode/decode KL1 term, etc.` is for encoding and decoding message packets to/from internal representations of KL1 term. It also includes the maintenance of the export/import tables and the foster parent records (c.f. section 5). It is the essential part of the message handling.

`Basic message handling routine` in Figure 9 corresponds to the simple data conversion between 40-bit tagged words and byte-serial messages. The routine includes data transfer to/from the hardware buffer. The cost can be potentially reduced by hardware supports. `Copy_RPKB` stands for copying a message packet from the hardware buffer to the software buffer. It is only executed when the hardware buffer tends to be full.

The network transfer speed is 0.2 μs/byte. It takes below 1 μs to hop one network node. It means that the message handling cost, just explained before, is dominant in the communication cost.

`Send_throw` (a) shows the cost of sending a 65 byte `%throw_goal` message containing a goal with three arguments. It takes 419 micro-instruction steps or 85 μs (cycle time = 200 ns). `Receive_throw` (b) shows the cost of receiving the same `%throw_goal` message and storing it in a goal stack.

The bar graphs (c), (d), (e) and (f) describe the cost of sending and receiving a `%read` message and

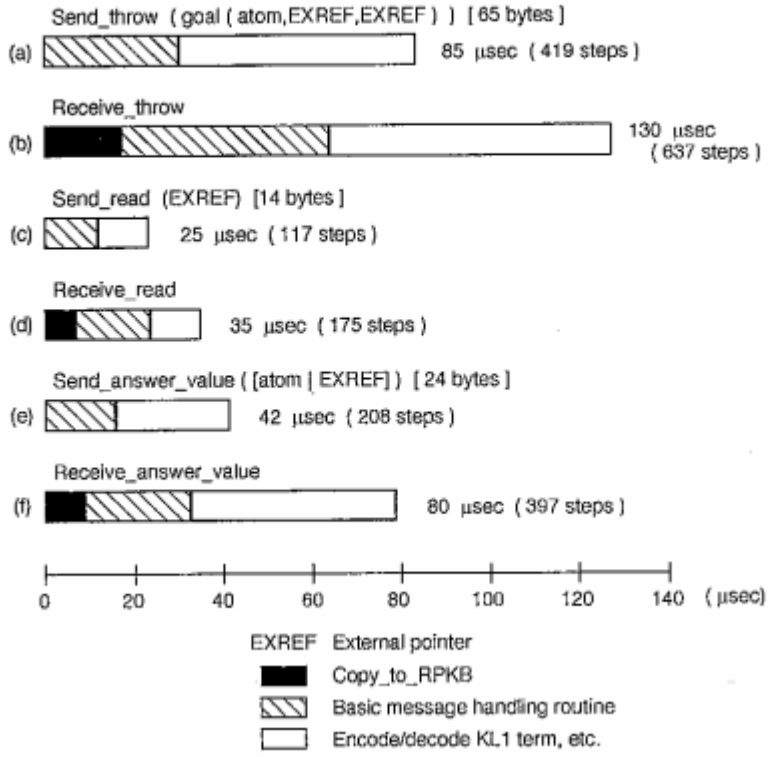

Figure 9: Message Handling Cost

Table 5: Message Frequency and Reductions

Pentomino (39.3 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (sec)	54.63	14.62	4.35
total reductions (×1000)	8,317.	8,332.	8,340.
reductions/sec (KRPS)	152.2	570.1	1,919.4
reductions/msg	221.	108.	88.
msg bytes/sec (×1000)	14.5	108.1	440.5

Bestpath (23.4 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (sec)	10.655	4.062	1.691
total reductions (×1000)	987.7	1213.6	1,505.2
reductions/sec (KRPS)	92.7	298.8	890.1
reductions/msg	21.9	11.7	6.2
msg bytes/sec (×1000)	114.0	692.5	3,854.3

(KRPS: Kilo Reductions Per Second)

Table 6: Single Processor Performance of PIM/m

benchmark	condition	PIM/m	Multi-PSI/v2	$\frac{Multi-PSI/v2}{PIM/m}$
append	1,000 elements	1.63 msec	7.80 msec	4.8
best-path	90,000 nodes	142 sec	213 sec	1.5
pentomino	8 × 5 box	107 sec	240 sec	2.2
15-puzzle	5,885 K nodes	9,283 sec	21,660 sec	2.3

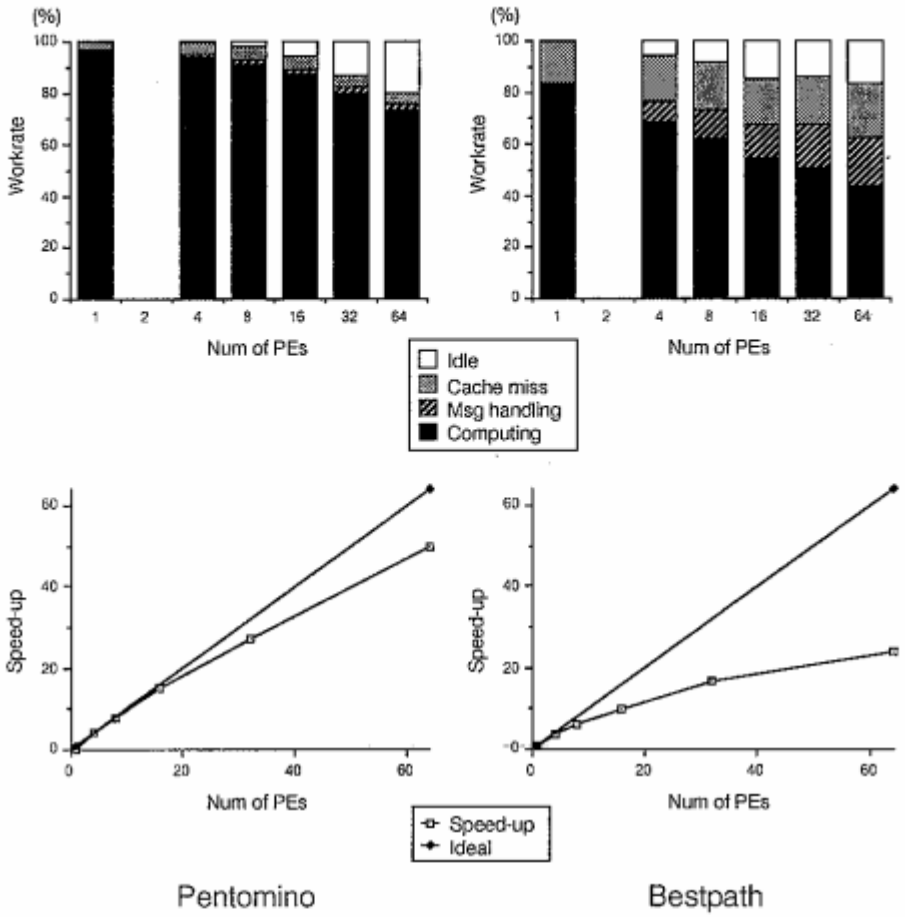

Figure 10: Decomposition of Processor Time and Speed-up

Table 7: System Performance on Pentomino (8 × 5 box)

No. of PEs	PIM/m		Multi-PSI/v2		$\frac{Multi-PSI/v2}{PIM/m}$
	Time	Speedup	Time	Speedup	
256 PE	1,124 ms	95.41			
128 PE	1,290 ms	83.13			
64 PE	2,162 ms	49.60	4,679 ms	51.20	2.16
32 PE	3,694 ms	29.03	8,278 ms	28.94	2.24
16 PE	6,910 ms	15.52	15,686 ms	15.27	2.27
1 PE	107,238 ms	1.00	239,545 ms	1.00	2.23

`%answer_value` message.

Sending and receiving cost of the `%throw_goal` message, 215 μs (1056 steps) in total, can be considered as the cost of a process fork to a different PE, or a remote procedure call. Cost of the `%read` and `%answer_value` messages, 182 μs (897 steps) in total, correspond to the cost of the remote synchronization.

Comparing these value with the cost of local operations in the previous section, the remote synchronization takes around 10 times higher cost than local. The remote procedure call costs more but below 40 times of the local process fork. These *remote/local ratio* seems low enough

Table 8: System Performance on Pentomino (10 × 6 box)

No. of PEs	PIM/m		Multi-PSI/v2		$\frac{Multi-PSI/v2}{PIM/m}$
	Time	Speedup	Time	Speedup	
256 PE	103,655 ms	234.29			
128 PE	188,452 ms	128.87			
64 PE	359.268 ms	67.60	886,325 ms		2.47
32 PE	694,553 ms	34.96	1,729,430 ms		2.49
16 PE	1,367,240 ms	17.76			
1 PE	24,285,015 ms	1.00			

to encourage the small-grain concurrent processing between PEs. Measurements of the communication cost give a guideline for the process grain size (communication rate) to keep the communication overhead low. When a process garin size decreases, becoming close to the communication cost, communication overhead increases significantly (close to 50% of CPU time).

6.1.3 Measurements on Benchmark Programs

Benchmark Programs: The followings are the two benchmark programs used here.

- **Pentomino:** A program to find out all solutions of a packing piece puzzle (Pentomino) by exploring the whole OR tree. Two-level dynamic load balancing is employed [Furuichi *et al.* 1990].

- **Bestpath:** A 160 × 160 grid graph is given together with non-negative edge costs. The program determines the lowest cost path from a given vertex to all vertices of the graph by performing a distributed shortest path algorithm [Wada and Ichiyoshi 1990]. The vertices are represented by KL1 processes, and they exchange shortest path information along the edges. 25,600 small processes work cooperatively.

Message & Reduction Profile: Table 5 shows the execution time, the reduction and message rates, etc. [Nakajima and Ichiyoshi 1990]. Average time of one reduction in a PE is an inverse of the KRPS value. 25 μs (127 steps) in Pentomino, and 43 μs (214 steps) in Bestpath. They are almost the grain size of concurrent processes in a PE. The message sending rates on 64 PEs are: one message per 88 reductions in Pentomino, and one per 6 reductions in Bestpath.

The average network traffic was repoted in [Nakajima and Ichiyoshi 1990], calculated from these figures. Relative to the 10 Mbyte/s network channel bandwidth, the average traffic on a channel is very small: 0.08% (Pentomino) and 0.3% (Bestpath) of the bandwidth.

Communication Overhead: Profiling data of processor execution has been measured on the two benchmark programs [Nakajima 1992]. The execution time is broken down into the four categories in Figure 10: computing time (reduction operations), message handling time, cache-miss penalty, and idling time. The average of all PEs are shown in the bar graph. The resultant speed-up is also shown with the ideal one.

Two-level dynamic load distribution is used in Pentomino. Several thousands small processes are distributed to 64 PEs in 4.35 seconds adaptively. The graph shows low communication overhead and good speedup. The degradation of processor workrate in 64-PE execution is mainly caused by the latency of load feeding to PEs.

In Bestpath, 25,600 small processes are distributed statically on 64 PEs. They exchange messages to perform an distributed algorithm. The inter-PE communication and the cache-miss penalty degrade the performance because of the high communication rate and the large working set. As the number of PEs grows, the grid graph is divided into smaller blocks to keep the workrate high, and it makes the inter-PE communication rate higher. Best path includs speculative computation, which increases with the large number of PEs. It causes lower speedup than a calculated value from the processor workrate.

Measurements results in table 5 and Figure 10 show the actual communication rate and communication overhead. Programmers can use relatively large communication rate, one message per 6 reductions (measured in Bestpath), with non-large CPU overhead of approximately 15%. Considering a network load of 0.3% at that time, it is observed that CPU load (15% at that time) will limit the communication band width when communication rate increases. The language implementation, which supports the global name space on a distributed memory hardware, tends to increase the CPU load concerned with network communication.

6.2 Preliminary Measurements on the PIM

6.2.1 Single Processor Performance

Table 6.1 shows the single processor performance of PIM/m for four benchmarks. The table also includes the performance of Multi-PSI/V2 and the ratio of PIM/m and Multi-PSI/V2 (M/P-speedup).

M/P-speedup is 1.5 to 2.3 in average. Programs with large working set tends to show low M/P-speedup.

6.2.2 System Performance

Table 7,8 show the preliminary measurements of system performance on PIM/m. The benchmark program is Pentomino.

Speedup saturation in Table 7 is caused by small problem size. Better speedup (234 folds speedup with 256 processors) was attained with larger problem in Table 8. It is also surprising that the small problem (executed in 1.1 second) show 95 folds speedup, which uses the multi-level dynamic load distribution distributing several thousands of small processes. The facts shows an efficient language implementation suitable to handle a lot of small-grain processes with less overhead.

7 Conclusion

This paper described two subjects. One is an overview of the research and development on the parallel inference machine PIM and the language implementation of the kernel language KL1, a concurrent logic programming language.

The other is the clarification of the features and advantages of KL1 language, its parallel implementation, and the hardware architecture from the viewpoint that the features are suitable and may be indispensable for efficient parallel processing of *the dynamic and non-uniform problems* with large computation. Knowledge processing is included in the problem domain. These problems have not been covered by commercial parallel machines and their software systems that target the scientific computation. The PIM system focuses on this new domain of parallel processing.

PIM is a distributed memory MIMD machine with a global view, connecting a maximum of 512 processors. It includes shared-memory substructures. Many component technologies have been developed that support efficient parallel processing on the target problem domain, especially on symbolic processing.

KL1 language also has very strong features for efficient programming and execution of the dynamic and non-uniform large problems. Major features are (1) small-grain concurrent processes, (2) implicit synchronization and communication, (3) separation of concurrency design and mapping (load allocation and scheduling), etc. They support highly concurrent programming with complex structures and support large flexibility for load balancing. The efficient language implementation made actual use of the language features possible. The PIM and KL1 system have realized a strong research and development environment for parallel software in that problem domain.

Measurements and evaluations showed a very low-cost language implementation for handling small-grain concurrent processes and their remote communications. Good speedup by parallel processing on benchmark programs was also reported. A lot of small-grain processes were handled during this processing. These results prove the efficiency and usefulness of the system to *the dynamic and non-uniform problems.*

Further measurement and evaluation is continuing, and the results of this will be reported soon. On the other hand, many problems of parallel software remain unsolved. Continuous research must be carried out to construct the real technology of large-scale parallel processing for *the dynamic and non-uniform problems* including the knowledge information processing in the 21st century. The parallel inference machine PIM and the KL1 language system will be utilized as the best research environment.

Acknowledgment

The R & D of PIM system have been carried out by researchers in the first research laboratory and cooperating companies, supported with valuable suggestions and helps by members of the second, seventh and the other ICOT laboratories and the PIM working group. The author would like to thank all of these people for their continuous efforts and cooperation.

References

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. of the Fourth Int. Conf. on Logic Programming*, 1987, pp.276–293.

[Chikayama 1992] T. Chikayama. Operating System PIMOS and Kernel Language KL1. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the Multi-PSI. In *Proc. of PPoPP'90*, pp. 50–59, 1990.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the*

Int. Conf. on Fifth Generation Computer Systems, ICOT, Tokyo, 1988, pp.208–229.

[Goto *et al.* 1989] A. Goto, A. Matsumoto and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *Proceedings of 16th Annual International Symposium on Computer Architecture*, pages 25 – 33, Jerusalem, Israel, 1989.

[Hirata *et al.* 1992] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase and K. Rokusawa. Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Ichiyoshi *et al.* 1987] N. Ichiyoshi, T. Miyazaki and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proceedings of Fourth International Conference on Logic Programming*, pages 257–275, University of Melbourne, MIT Press, 1987.

[Ichiyoshi *et al.* 1988] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *New Generation Computing*, Ohmsha Ltd. 1990, pp.159–177.

[Ichiyoshi 1989] N. Ichiyoshi. Parallel logic programming on the Multi-PSI. ICOT Technical Report TR-487, ICOT, 1989. (Presented at the Italian-Swedish-Japanese Workshop '90).

[Imai *et at.* 1991] A. Imai, K. Hirata and K. Taki. PIM Architecture and Implementations. In *Proc. of Fourth Franco Japansese Symposium*, ICOT, Rennes, France, 1991.

[Imai and Tick 1991] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *ICOT Technical Report*, TR-650, 1991. (To appear in IEEE Transactions on Parallel and Distributed Systems)

[Inamura *et al.* 1988] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Techiniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. of the North American Conf. on Logic Programming*, 1989, pp. 907–921 (also *ICOT Technical Report*, TR-466, 1989).

[Kimura and Chikayama 1987] Y. Kimura and T. Chika-yama. An Abstract KL1 Machine and its Instruction Set. In *Proc. of Symposium on Logic Programming*, 1987, pp.468–477.

[Kumon *et al.* 1992] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa and K. Hirano. Architecture and Implementation of PIM/p. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Masuda *et al.* 1988] Y. Masuda, Y. Ishizuka, Y. Iwayama, K. taki and E. Sugino. Preliminary Evaluation of the Connection Network for the Multi-PSI System. In *Proc. Europian Conference on Artificial Intelligence 1988 (ECAI-88)*, August 1988.

[Matsumoto *et al.* 1987] A. Matsumoto, T. Nakagawa, M. Sato, K. Nishida and A. Goto. Locally Parallel Cache Design Based on KL1 Memory Access Characteristics. ICOT Technical Report 327, 1987.

[Nakagawa *et al.* 1989] T. Nakagawa, A. Goto and T. Chikayama. Slit-Check Feature to Speed Up Interprocessor Software Interruption Handling. In *IPSJ SIG Reports*, 89-ARC-77-3, 1989 (*In Japanese*).

[Nakagawa *et al.* 1992] T. Nakagawa, N. Ido, T. Tarui, M. Asaie and M. Sugie. Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. of the Sixth Int. Conf. on Logic Programming*, 1989, pages 436–451.

[Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI. In *ICOT TR-531*, 1990.

[Nakajima 1992] K. Nakajima. Distributed Implementation of KL1 on the Multi-PSI. In *Implementation of Distributed Prolog*, edited by P. Kacsuk and M. Wise, John Wiley & Sons, Ltd., 1992.

[Nakashima and Nakajima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987, pp 104–113.

[Nakashima *et al.* 1992] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y. Inamura, S. Onishi and K. Masuda. Architecture and Implementation of PIM/m. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Nishida *et al.* 1990] K. Nishida, Y. Kimura, A. Matsumoto and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *Proc. of the Seventh Int. Conf. on Logic Programming*, 1990, pages 83–95.

[Nitta *et al.* 1992] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Onishi *et al.* 1990] S. Onishi, Y. Matsumoto, K. Nakajima and K.Taki. Evaluation of the KL1 Language System on the Multi-PSI. In *Proc. of Workshop on Parallel Implementation of Languages for Symbolic Computation*, July 30–31, 1990, Oregon, USA. Also ICOT TR-585.

[Rokusawa *et al.* 1988] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proc. of the 1988 Int. Conf. on Parallel Processing*, Vol. 1 Architecture, 1988,pp.18–22.

[Rokusawa and Ichiyoshi 1992] K. Rokusawa and N. Ichiyoshi. A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting. In *Proc. of Sixth Int. Parallel Processing Symposium*, IEEE, 1992.

[Sato *et al.* 1987] M. Sato, A. Goto, et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987.

[Sato and Goto 1988] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proc. of IFIP Working Conf. on Parallel Processing*, 1988, pp. 305–318.

[Sato *et al.* 1992] M. Sato, K. Takeda and T. Ohara. Design of the Parallel Inference Machine PIM/i Processor. In *Trans. of IPSJ*, Vol.33, No.3, 1992, pp. 278–287 (*In Japanese*).

[Shinogi *et al.* 1988] T. Shinogi, K. Kumon, A. Hattori, A. Goto, Y. Kimura and T. Chikayama. Macro-call Instruction for the Efficient KL1 Implementation on PIM. In *Proceedings of the International Conference on Fifth Generation Computing Systems 1988*, Tokyo, Japan, pages 953–961, 1988.

[Takagi and Nakase 1991] T. Takagi and A. Nakase, Evaluation of VPIM: A Distributed KL1 Implementation – Focusing on Inter-cluster Operations –, In *IPSJ SIG Reports*, 91-ARC-89-27, 1991 (*In Japanese*).

[Takeda *et al.* 1988] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Taki *et al.* 1984] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1984*, pp.398–409, Tokyo, Nov. 1984.

[Taki 1988] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI System. In *Programming of Future Generation Computers*, K.Fuchi and M.Nivat (Editors), pages 411–426, Elsevier Science Publishers B.V., North Holland, 1988.

[Uchida *et al.* 1988] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of The FGCS Project. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988*, pp.16–36, Tokyo, Nov. 1988.

[Uchida 1992] S. Uchida. Summary of the Parallel Inference Machine and its Basic Software. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

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

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

[Wada and Ichiyoshi 1990] K. Wada and N. Ichiyoshi. A study of mapping locally message exchanging algorithms on a loosely-coupled multiprocessor. ICOT Technical Report TR-587, 1990.

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

[Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proc. of Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, 1987, pp.432–443.