ICOT Technical Report: TR-715

TR-715

Distributed Implementation of KL1 on the Multi-PSI

by
K. Nakajima (Mitsubishi)

November, 1991

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

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Distributed Implementation of KL1 on the Multi-PSI*

Katsuto Nakajima

Abstract

KL1, a committed choice language based on Flat GHC, was implemented on a distributed memory multi-processor, the Multi-PSI, which has up to 64 processing elements (PEs) connected by a message passing network.

The key issues for a distributed implementation of KL1 are: (1) how to reduce the amount of inter-PE communication, (2) how to achieve efficient intra-PE and inter-PE garbage collection, and (3) how to avoid making redundant copies of data objects over many processors.

The well-defined semantics of KL1 allows incremental intra-PE garbage collection by the Multiple Reference Bit (MRB) technique and incremental inter-PE garbage collection by the Weighted Export Counting (WEC) technique. A global structure management mechanism is also introduced to avoid making duplicate copies of large data objects such as program codes. The communication required for inter-PE process control is minimized by the Weighted Throw Counting (WTC) scheme.

The implementation has been completed and the performance of inter-PE communication was evaluated. Although the cost for inter-PE message handling is high, the effective overhead in the benchmark programs remains within 20% on 64 PEs.

1. Introduction

KL1 is a stream AND-parallel logic programming language based on Flat GHC [Ueda86a] [Chik88a]. It was designed in ICOT [Ueda90a] as an interface to fill the gap between the knowledge information processing software and the hardware of the parallel inference machine(PIM) [Goto88a]. It is not only a system description language but also an application-user language for the PIM.

*This chapter is an extended version of the paper titled "Distributed Implementation of KL1 on the Multi-PSI/V2" in Proceedings of International Conference on Logic Programming, 1989.
Section 7 was derived from the paper titled "Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI" by K. Nakajima and N. Ichiyoshi in Proceedings of International Conference on Parallel Processing Vol.I (Univeristy Park: The Pennsylvania State University Press, 1990), pp 613–4.

The Multi-PSI system [Taki88a, Nakj89a] was developed as a prototype machine for the PIM, having the purpose of serving as a testbed for implementing concurrent language KL1 on a scalable multiprocessor architecture. It is a distributed memory multiprocessor, whose processing elements (PEs) are the CPUs of the personal sequential inference (PSI) machine [Naks87a]. Up to 64 PEs are connected by an 8×8 mesh network with wormhole routing. The two-dimensional mesh network has a dense and simple implementation, and is scalable in that network node degree stays a constant (4) as the number of nodes increases. The PEs are microprogrammable and have an architecture suitable for executing logic programming languages efficiently.

A distributed KL1 implementation was developed on the machine. It is written in microcode for performance. The design rationale was to obtain a high overall performance, taking account of garbage collection overhead, and to decentralize management information for scalability.

Some performance measurements have been done so far especially on the inter-PE operations in the system, both in absolute terms (cost of primitive operations) and in relative terms (rate of communication overhead in non-trivial benchmark programs), which reveal the bottlenecks in the performance of inter-PE operations.

2. GHC and KL1

2.1 GHC and Flat GHC

GHC (Guarded Horn Clauses) is called a committed choice AND-parallel languages. A GHC program is made up of a collection of guarded horn clauses, whose form is:

$$\underbrace{H :- G_1, \ldots, G_m}_{\text{guard}} \mid \underbrace{B_1, \ldots, B_n}_{\text{body}}. \ (m > 0,\ n > 0)$$

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

The guard part can be considered as test. If there are alternative clauses, their guard parts may be tested concurrently (OR-parallel). However, only one clause can commit even if more than one clauses can meet the test condition. The execution of the rest of the clauses are canceled. The caller goal is reduced to the body goals of the committed clause. These body goals are executed concurrently (AND-parallel). Body goals may be a unification goal of the form "$term_1 = term_2$," which may perform a binding to a variable. Body goals are, otherwise, a user-defined goal which represents the rest of the work. They communicate each other through their common variables which often represent a *stream* in the form of D-list.

In the guard part, binding to the caller variable is not allowed and the attempt of the binding causes a clause suspension. If there is no clause to commit and at least one clause is suspended, the predicate call itself suspends. It serves as a synchronization mechanism between GHC goals.

In Flat GHC, only predefined predicates are allowed as guard goals. This restriction does not decrease the expressive power of the language as the guard goals are a kind of auxiliary conditions of the clause[Ueda90a], while it makes the implementation much easier and more efficient.

2.2 KL1

GHC or Flat GHC is a clearly defined *concurrent* language. KL1 was born from them as a practical and efficient language for the parallel inference machine. Therefore, it is called a *parallel* language. KL1 is provided with the following meta-programming functions for describing not only application programs but also operating systems.

(1) Shōen mechanism: A shōen is a meta-logical unit for controlling and monitoring KL1 goals. It corresponds to a "task" or a "process" in ordinary operating systems. A shōen consists of all the goals which descended from the given initial goal of the shōen.

Shōen has a pair of input and output streams for interfacing with the outside. The input stream named the *control stream* is used to start, stop or abort the execution of the shōen from outside. Special events that occurred inside a shōen, such as termination of all goals, a failure or an exception, are reported on the output stream named the *report stream*. Shōen can be nested to form a tree-like structure (shōen tree) whose leaves are KL1 goals.

(2) Resource management: The upper limit of the resource that can be consumed by a shōen can be specified by a control message. This mechanism prevents programs with bugs to go on running wastefully such as in an erroneous infinite loop. Resource shortage is notified on the report stream, and additional amount of resource can be selectively given. Currently, resource is measured by the number of reductions.

(3) Priority pragma $(\ldots, B@priority(Prio), \ldots)$: Scheduling by using *priority* contributes to efficient problem solving. The shōen has a priority range and each goal inside it can have an individual priority within this range. The priority is specified by a priority pragma with a relative value in the allowed range.

(4) Throw goal pragma $(\ldots, B@processor(PE), \ldots)$: A throw goal pragma in the source program denotes load distribution. It also contributes to efficient execution on a multi-processor machine.

The priority and processor pragmas are merely guidelines for language implementations and may not strictly be obeyed. The control such as an abortion in a shōen may not be immediately performed. As these functions do not affect the correctness of the programs and are expected only for the efficiency, distributed implementation becomes much easier.

3. Architecture of the Multi-PSI

3.1 Processing Elements

The processing element (PE) of the Multi-PSI is the CPU of the sequential inference machine, the PSI-II [Naks87a]. It is a 40-bit (8-bit for tag, 32-bit for data) CISC processor controlled by the horizontal micro-instruction. It enables a flexible implementation suited for incrementally enhancing the performance and adding various functions. The cycle time is 200 ns. It has up to 16 Mwords of local memory and a 4-Kword direct-map cache memory.

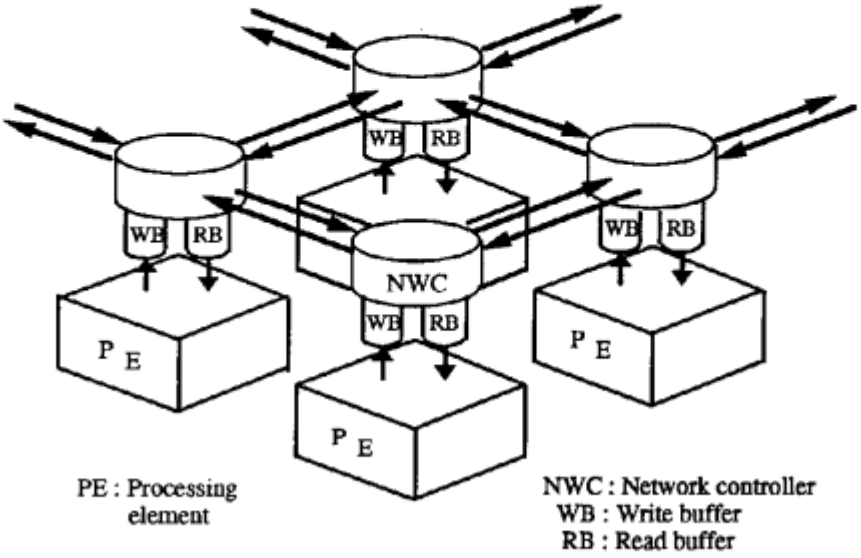

Figure 1: Processor Inter-connections of the Multi-PSI System

3.2 Network Controller

Each PE is paired with a specially designed network controller to support message passing communication between PEs. The network controller has five pairs of input/output channels connected to the four adjacent network nodes and to the PE of the node (Figure 1). Each channel consists of 11 bits, 9 bits for data, one for a parity and one for a busy acknowledge signal of the opposite direction.

The cycle time is 200 ns and the bandwidth of each channel is 5 Mbytes/s. Each output channel has a 48-byte buffer (*Output Buffer*) to retain messages when the destination node is busy. Each input and output channel for the PE of the node has a 4-Kbyte buffer (*Write Buffer* and *Read Buffer*) to reduce disturbance of processing in the PE. As soon as a complete message (from a message header to a tail) is written by the PE in the Write Buffer, it is shipped out by the controller unless the transmitting channel is busy. When a complete message is taken in the Read Buffer by the controller, the PE is informed by an interrupt signal.

The controller has the wormhole routing capability. It can route messages according to the PE number in the message header (Figure 2). A software-defined table called the *path table* is looked up to determine transmission direction. A fixed routing strategy called *prioritized coordinate ordering* is adopted. It means to transmit a message along the *x-coordinate* until the distance in the coordinate becomes zero, then to transmit along the *y-coordinate*. As long as messages are taken into the destination PE, network deadlocks never arise because there can be no cyclic chain of messages in transit blocking one another.

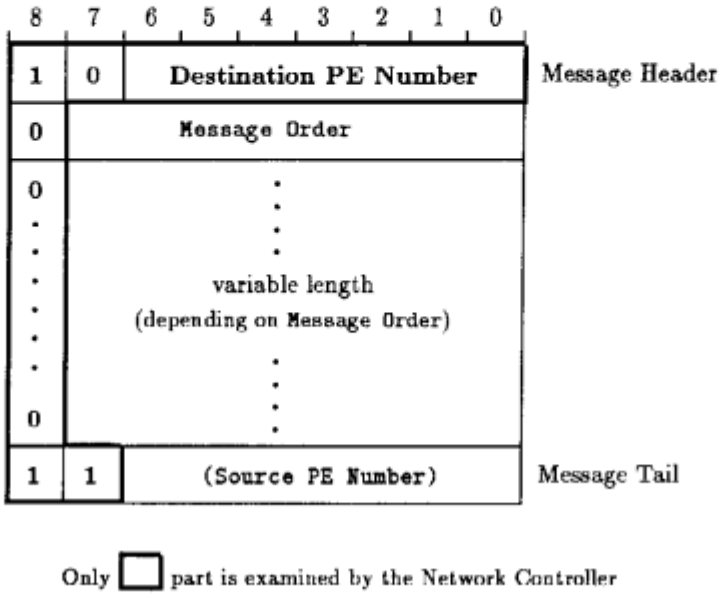

Figure 2: Message Packet Format

3.3 Messages Handling

The read/write of *Read Buffer* and *Write Buffer* are done by microcode. A low level microcoded routine in the PE is responsible for handling messages to and from the network controller. It performs composition and decomposition of message packets such as handling message header and tail (Figure 2), and arranging a 32-bit data in the PE to/from four byte serial data in the Write/Read Buffer.

The micro routine is invoked by an interrupt when the message tail is queued in the Read Buffer. It moves the complete message to a large memory area, called the *Read Packet Buffer*. The aim is to prevent network deadlock by propagating the processing delay to other nodes along the network path. The messages in the Read Packet Buffer are decoded at a slit of reductions. On sending a message, if the routine cannot find enough room to store the whole message in the Write Buffer, it will wait until more room becomes available.

4. Intra-PE Processing

4.1 Execution Model of KL1 goals

KL1 programs are compiled into the sequence of WAM [Warr83a]-like abstract machine instructions named KL1-B [Kimu87a]. It is a register-based instruction set and serves as an efficient interface between the language and machine architecture.

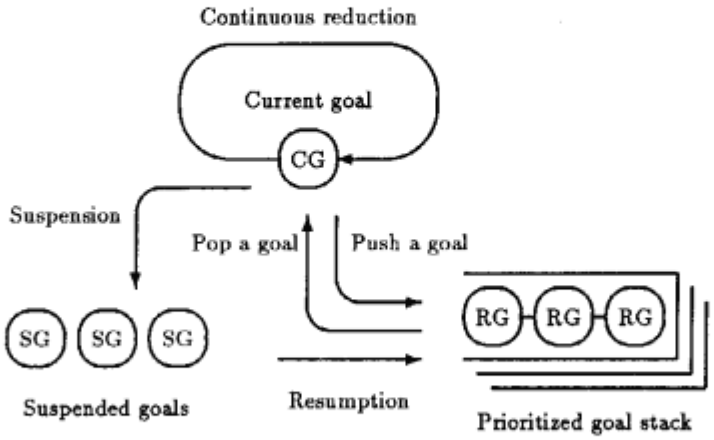

Figure 3: Goal State Transition

Goals are categorized as: (1) *ready goals* which are waiting for execution, (2) *current goal* which is being executed, or (3) *suspended goals* which are hooked on variables to be instantiated (Figure 3).

In this implementation, each PE has its own goal stack. Scheduling is performed on each PE individually. A reduction cycle at each PE is as follows. When a current goal calls a predicate, the guard parts of the clauses for the predicate are tested. In spite of the language allowing to evaluate alternative clauses simultaneously, guard unification is performed one by one for efficiency reason. Non busy-wait suspension is employed. That is, if the predicate call suspends, the goal is hooked on the variable(s) that caused the suspension. If no clause commits and there is no suspended clause, a *failure* is reported on the report stream of the shōen with the goal information. In either case, another goal is popped from the goal stack to be evaluated. If one of the clauses commits, all the user-defined body goals, except for the unification goals and the leftmost user goal in the clause, are pushed to the goal stacks according to the priority pragmas attached to the goals. Unification goals are executed immediately. The leftmost user goal is chosen as the next one to be evaluated unless the goal stack of the higher priority has a goal. When a clause without body goals commits, another goal is popped from the goal stack.

Every reduction cycle, request for (non-incremental) GC and message arrivals from the network are checked, and processed if necessary. This timing, called the *slit check*, is most suitable for switching the process because the PE is free from goal contexts.

4.2 KL1-B Execution

The KL1-B instructions, including approximately one hundred instructions for predefined predicates, are directly interpreted by the microcode to attain high execution speed.

The microcode of the PE can perform various functions in parallel, such as tag insertion, two-way or multi-way branching on a tag, specially prepared counter and flag operation, ALU operation and memory access operation. The arguments for the

Figure 4: References in the MRB Scheme

unification are fetched on the registers every reduction cycle. Control information, such as the priority and the shōen resource, is cached on the registers as long as the execution context remains unchanged.

4.3 Local Goal Scheduling

If the goal priority should be kept strictly on a multiprocessor, only one prioritized goal stack must be managed in the system. However, the access contention on such a global resource leads to serious performance degradation.

As KL1 priority pragma is a guideline for efficiency, each PE may have each prioritized goal stack, at the sacrifice of scheduling strictness. Even with such a local scheduling, it is known to be efficient enough to control the execution in most cases. However, the maldistribution of high priority goals over PEs is possible and problematic. A dynamic load balancing by software is necessary in such a case.

4.4 Memory Management using MRB

As goals may not be executed in a last-in-first-out manner, the stack mechanism used in most Prolog implementations is not suitable for a KL1 implementation. Therefore, heap-based memory management must be used for flexible memory use, although memory reclamation is generally inefficient with this scheme. The time spent in GC may seriously affect the system performance. Thus, efficient GC is vital in a KL1 implementation.

Goal contexts are maintained in a *goal record*, which contains the predicate code address, arguments, priority and etc. Goal record can be reclaimed at popping from goal stack into a free goal record list. However, variable cells or record area for structured data are difficult to reclaim because they may be shared by two or more pointers. Incremental GC by reference counting is desirable because of its access locality and high hit rate of cache memory. However, in reference counting, each word cell must have a reference counter field for the whole memory space. In addition, the cost of updating the reference counter is high, because data objects must always be accessed.

Several methods were proposed to reduce these overheads relying on the fact that data objects are not used so many times, and most are used only once [Deut76a]. The *Multiple Reference Bit* (MRB) method was proposed [Chik87a] as an incremental GC method for concurrent logic programming languages.

The MRB method maintains one-bit information in pointers indicating whether the pointed data object has multiple references to it or not. This multiple reference information makes it possible to reclaim storage areas that are no longer used.

Figure 4 shows the data representation in the MRB scheme. A single-referenced object (a) and a multi-referenced object (b) can be distinguished by the MRB flag on the pointers, *off-MRB* by○ and *on-MRB* by● .

The MRB method has the following two advantages;

- By keeping MRB information in the pointers rather than in the pointed objects, no extra memory access is required for reference information maintenance.
- On updating an *off-MRB* array, destructive assignment can be performed.

As variables in logic languages like KL1 can not be overwritten, updated array should not be identical to its original one. Each update causes a copy of the whole array with one element modified. However, in the case of a single-referenced array, the original array can be reused as a new one by modifying the element destructively.

The MRB information is also used and maintained through the unification. When a unification consumes a reference path to a single-referenced data object, the storage area can be reclaimed after the unification. It is known that even with the one-bit counter, more than 60% of the garbage cells are collected in various benchmark programs. Collected cells are linked in the free lists to be reused. Several individual free lists for records of various sizes are prepared for them[1].

When records in a free list are exhausted, a pre-determined number of new records are created on the heap top and linked to the free list. In the current implementation, fragmentation among the free lists is resolved only by local (non-incremental) GC, because, free list handling operation is too frequent to employ fragment reconstruction techniques such as the buddy system [Know65a].

5. Inter-PE Processing

5.1 Implementation Issues for KL1

In this KL1 implementation the followings were the most important issues because it was aimed at a scalable system on a distributed memory multiprocessor.

Reducing Message Communication: Message passing communication is more expensive than communication on a shared memory. The communication delay is also large. Reducing the amount of inter-PE communication and maintaining quick responses are major subjects in the implementation.

Fewer and Local Garbage Collection: As stated in 4.4, GC is the key point in the implementation for concurrent languages like KL1. On a distributed memory multiprocessor, the degradation by GC should be considered more seriously, because naïve inter-PE GC might take time proportional to the length of the reference chains over many processors.

Efficient Data Management: KL1 has a property where data objects that have been instantiated once can be copied, while keeping the program logic. To allow local access, data shared by PEs should be copied. However, uncontrolled copying leads to unnecessary data transfer.

[1] In the current implementation, the sizes are from 1 to 8, 16, 32, 64, 128 and 256. A record over 256 words is allocated on the heap top.

5.2 Goal Distribution and Distributed Unification by Inter-PE Messages

5.2.1 Goal Distribution

A KL1 goal specified with a throw goal pragma ($B@processor(PE)$) is distributed to another PE. The goal information in the goal record such as predicate code, goal arguments and execution priority are encoded into `%throw_goal` message. If the goal argument is an atomic data, it is encoded directly. If it is a structured data such as vector or string, or uninstantiated variable, it is encoded as an *external pointer*. The predicate code address is encoded to a pair of an external pointer representing code module and an offset representing the code position in the module. The code module handling will be explained in 5.4.8.

At the destination PE, the message is decoded, and a goal record is composed and is put in the goal stack according to its priority.

5.2.2 Guard Unification

As binding to the caller variable is not allowed in the guard part, guard unification with an external pointer is suspended until its value is known.

If there is no other clause to commit, a `%read` message is injected to get the value of the external pointer to the PE which has the data. A `%read` message has two arguments: the external pointer to read and a return address where to be replied to. The later is also an external pointer pointing to the cell which substitutes the former external pointer and will be instantiated by the value returned by `%answer_value` message. The suspended goal is hooked on the cell until it is instantiated.

At the PE where `%read` is received, an `%answer_value` message is immediately replied for the `%read` message if the contents of the externally referenced data is an instantiated value. If the value is an structured data, the surface level (that is, the elements of the array or list) are encoded. If the elements are structured data, they are encoded as external pointers. If the exported data is still an uninstantiated (unbound) variable, the reply is suspended by hooking the received `%read` message on the variable. The arguments of `%answer_value` are the destination and the value to reply.

5.2.3 Body Unification

In body unification between an unbound variable and another, the variable can be instantiated with the other.

A body unification with an external pointer and an instantiated data is encoded into `%unify` message, which corresponds to a write operation in this case. The actual operation is shifted to the PE (exporting PE) which has the data cell of the external pointer.

A body unification between an external pointer and a variable falls in two cases: (1) binding the variable with the external pointer or (2) injecting `%unify` message to shift the operation to the exporting PE. One of them is chosen according to the binding rules to prevent a loop of references over PEs. The binding order is determined with the two exporting PE numbers.

If a PE attempts to perform a body unification between two external pointers, it sends a `%unify` message to shift the work to one of the exporting PEs. The direction is also determined by the PE numbers.

Table 1 summarizes the typical inter-PE messages.

Table 1: Typical Inter-PE Messages

Message Order	Note
`%throw_goal(PE,Code,Args, Priority,etc.)`	Move the goal to specified PE
`%read(Ext,Return)`	Request to reply the value of an external pointer
`%answer_value(Val,Return)`	Respond to a `%read` message
`%unify(Ext,Val)`	Request to unify with an external pointer and an argument
`%terminated(Shoen)`	Report a local shoen termination in a PE
`%release(WEC)`	Return `WEC` value of an external pointer

A `%terminated` message reports a local shōen termination in a PE. A `%release` message carries GC information. They are explained in the following sections.

5.3 Distributed Goal Control

5.3.1 Creating KL1 Shōen

A shōen is created by the following predefined predicate *execute*/6:

$$execute(Goal, ControlStream, ReportStream, MinPrio, MaxPrio, Mask)$$

Goal is the initial goal to execute inside the shoen. $MinPrio$ and $MaxPrio$ specify the range of the execution priority allowed in the shōen. *ControlStream* is used to start, stop or abort the execution inside from outside the shōen. It is also used to supply execution resource. From *ReportStream*, the events inside the shōen such as exceptions, resource shortage and the termination of the execution can be observed. Shōen itself never fails even if one of the goals inside the shōen would fail. The failure is treated as an exception and reported in the *ReportStream*. $Mask$ is a bit-map to specify which exceptions should be reported.

Shōens can be nested. A child shōen is treated as one of the goals in the parent shōen.

5.3.2 Shōen and Foster Parent

Goals which belongs to the same shōen are distributed over many PEs. Any event at each goal should be reported to the shōen staying at the PE where it is created. To reduce the message traffic towards a shōen, a cache technique is employed. When a goal is moved from the shōen PE (the PE that contains the shōen) to another, a *foster parent* is created on the PE to which the goal migrated [Ichi87a]. Only one foster parent is created for the shōen on each of the PEs which have goals belonging to the shōen (Figure 5).

The foster parents have the *shōen status* (*running, stopped or aborted*), the *child count* (the number of child goals created on the PE), and the cached resource information of the shōen. Goal termination is checked at the shōen only when one of the foster parents sends a `%terminated` message to report that its child count has reached zero.

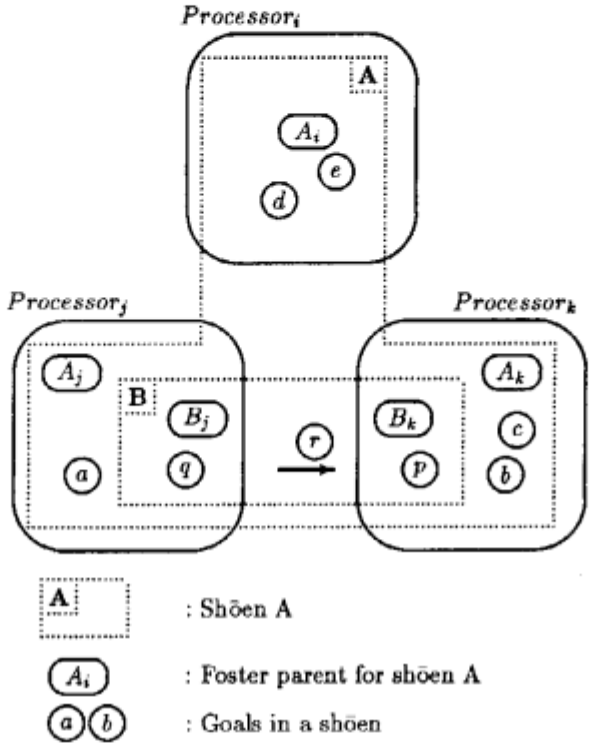

Figure 5: Shōen and Foster Parents

5.3.3 Termination Detection by WTC

The termination detection is one of the difficult problems in parallel computation systems, especially when messages may be in transit on the network as in the Multi-PSI. Even if all the foster parents report `%terminated`, the shōen is not necessarily terminated, because there are goals in transit.

One of the solutions for it is the *weighted throw counting* (WTC) scheme [Roku88a], which is an application of the Weighted Reference Counting (WRC) scheme [Wats87a]. In this scheme, each shōen manages the amount of the count called WTC. `%throw_goal` and `%unify` messages are always attached with some amount of WTC, and the foster parents accumulate their WTCs when receiving the messages. Foster parents can split it when they throw child goals or issue unify messages.

In this scheme, the following invariant is kept:

$$WTC_{shoen} = \sum(WTC_{fosterparent}) + \sum(WTC_{message})$$

where WTC_{shoen} is the WTC value maintained in the shōen which should be returned. $WTC_{fosterparent}$ is a WTC held in each existing foster parent for the shōne. $WTC_{message}$ is a WTC attached to a `%throw_goal`, `%unify` or `%terminated` message in transit in the network.

When all the goals in a foster parent terminate, the amount of WTC kept by the foster parent is returned to the shōen by `%terminated`. The shōen can determine the true termination of the child goals when all WTC is returned.

This scheme has the following features.

- WTC can be split locally (at each foster parent), without sending a message (to the shōen) to maintain the reference counting.
- No racing occurs in terms of checking zero count at the shōen.

5.4 Inter-PE Data Management

5.4.1 Copying Shared Data

When a goal is thrown to another PE, its arguments are also carried with it. If the argument is an atomic value, the value itself is sent with the goal. If it is an unbound variable, a pointer to the variable is created and carried. For a structure argument, there are three reasonable choices. One is to create and carry a pointer to the structure (0-level copying). The contents will be read when they are actually to be used in a unification. The second is to copy all the elements of the structure including all nested substructures (infinite-level copying). The third is to copy all the elements at the surface level (1-level copying).

In a distributed system like the Multi-PSI where the cost of the inter-PE reference is relatively high, it is better to copy data for later accesses in many cases. However, an infinite-level copying may cause unnecessary duplication because the passive or active unification for the structure might fail at any level in the destination PE.

Following the policy of *on-demand* copying, the 0-level copying is done for the arguments of thrown goals and the 1-level copying for those of unifications (for the `Val` of `%answer_value` and `%unify` in Table 1). It is one of the design decisions to be evaluated. At least, if it is known that the element of a structure will be read sooner or later (such as the next element of a stream), it is better to copy the elements at one time as long as they are bound to a value. This is left as a future optimization.

5.4.2 Export and Import Tables

When a PE exhausts its heap memory, garbage must be collected. If the PE does not know whether a cell is referenced from other PEs or it is garbage, the PE cannot perform GC for its memory without cooperation by all PEs. Global GC, where all PEs perform GC at one time by exchanging marking messages and indicating the movements of object cells [Ali86a], can be a solution, but it is expected to be very time consuming.

In a large scale distributed memory multiprocessor, local GC, where the PE performs GC alone for its memory when exhausted, is desirable in terms of the system performance. It is possible if the object cells referenced from outside are represented by a kind of global ID for the external PEs. The garbage collecting PE needs only maintain the translation table according to the local object movements in a local GC. The table is called the *export table* (Figure 6). The object cells referenced from the external PEs are said to be *exported*, and on the referencing side, they are said to be *imported*. The global ID is represented in the form $< pe, entry >$, where *pe* is a PE

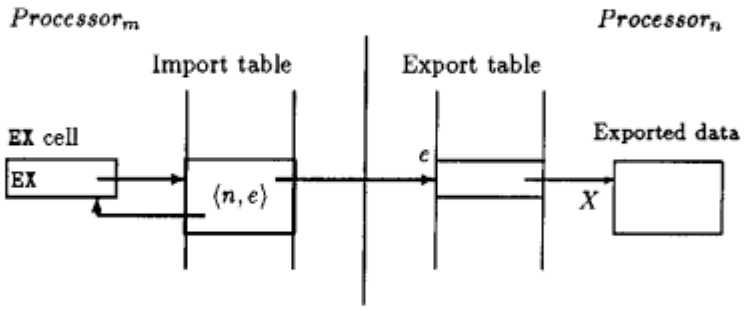

Figure 6: Export Table and Import Table

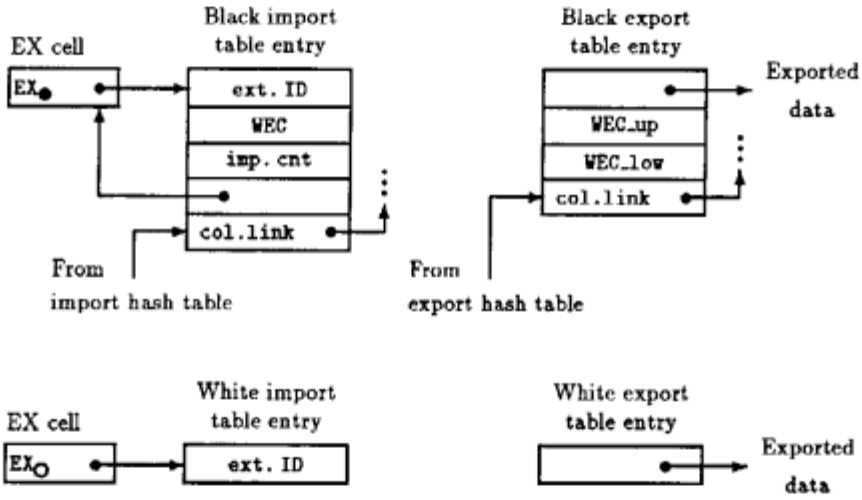

Figure 7: Export and Import Table Entries

number and *entry* is the entry position of the export table. The global ID is called the *external ID*.

5.4.3 Incremental Inter-PE GC by Weighted Export Counting

To reclaim the garbage cells pointed to by the export table, the entries of the table must be collected when they become garbage. The *weighted export counting* (WEC) method [Ichi88a] is employed to perform inter-PE incremental GC. This scheme is also based on the WRC principle.

An integer representing a WEC value is attached to an exported pointer and is stored in the *import table entry* (Figure 7). In this scheme, the total of WEC value of the imported or to be imported pointers is kept equal to the value at the exporting PE(Figure 8).

Inter-PE incremental GC is performed as follows.

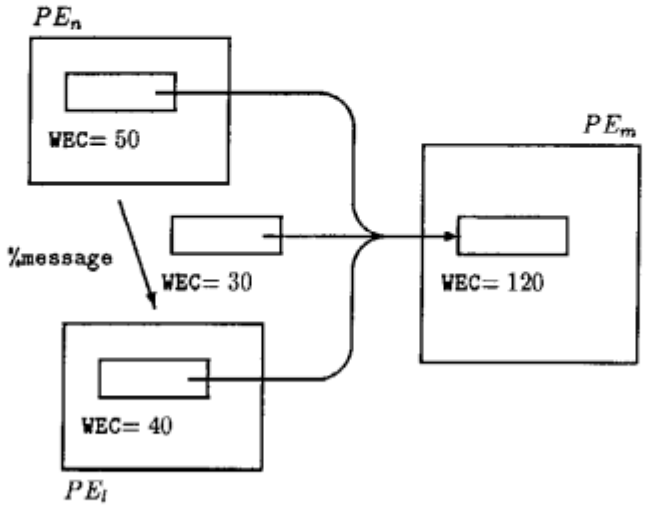

Figure 8: Weighted Reference Counting Scheme

Each entry for the imported pointers accumulates a WEC when the same pointer is imported again. The number of imports, the *import count*, is also counted. When the contents of the imported pointer are no longer necessary, a **%release** message is sent to the exporting PE to return the amount of the WEC. A **%release** message is sent when: (1) an instantiated value is returned by an **%answer_value** message in response to a **%read** message, (2) the import count reaches zero by incremental GC by using the MRB mechanism, or (3) an imported pointer is known to be garbage after a local GC in the importing PE (see 5.4.6).

When an export entry receives a **%release** message, the WEC in the entry is maintained. If it becomes zero, the entry is reclaimed. In this case, the exported object cell itself may also be reclaimed when the export table entry is known to be the single reference to the cell by the MRB mechanism.

5.4.4 Re-exporting

A variable may be exported to the same PE more than once. If the re-exported pointer to a variable is given with a different external ID, it cannot be determined that it refers to the same variable. Therefore, the importing PE may send **%read** messages twice, and if the object is a structure data, its copy is brought twice by **%answer_value**. This can be avoided by reusing the same export and import table entries. For this purpose, the *export hash table* and *import hash table* are provided on each side. The export hash table associates the exported object addresses with their external IDs, and the import hash table associates the imported external IDs and the import table entry.

5.4.5 White and Black Exports

This external reference management with WEC has overhead in terms of maintaining both the WEC and import count, and of looking up the hash table to check the re-exporting. Fortunately, the MRB mechanism can be used to optimize them. To export

a single reference pointer at a low cost, a simplified pair of export and import tables, called the *white export* and *white import tables*, are used (Figure 7). The original tables are called the *black export* and *black import tables*. Pointers that was duplicated once will mostly be copied again later. In contrast, a single reference pointer is not likely to be duplicated after being exported. Thus, the white export and import tables do not have hash tables because the exported pointers are rarely exported again for the same reason.

The white import table can be considered as an import table for the pointers whose WEC and import count equals one, and its entries are released immediately when the imported pointers are collected by the MRB GC[2]. Their white export entries are also released only when the **%release** message is received. The effectiveness of this optimization depends heavily on the programs; however, the average characteristics are expected to be similar to that of the MRB inside a PE.

5.4.6 Local Garbage Collection

The current implementation of local GC is based on the conventional copying GC scheme. The garbage collector moves all data cells reachable from the prioritized goal stacks and the export tables to a new semi-space. After copying, valid entries in the import tables are swept. If unmarked entries are found, **%release** messages are sent to the exporting PEs to return their WECs.

The time spent in local GC can be a big factor in the total performance because PEs requesting a response from the garbage collecting PE have to wait for its termination[3]. One solution to improve the GC time is *generation GC* [Lieb83a][Nakj88a], which avoids moving long life objects at every GC. This is worth while investigating but not implemented yet.

5.4.7 Global Structure Management

In this export system, the external ID originates from the exporting PE. For example, if PE_B has a copy of the structure in PE_A, and PE_C has external references to both the original structure in PE_A and the copy in PE_B, their external IDs are not the same. PE_C will have two copies after reading them. If the structure is big and will live long, it is inefficient in terms of both the memory space and the data transfer overhead. In the worst case, copies are created at each imports if a pair of mutually linked structures are read alternately along the loop.

The *structure ID* solves this problem for such structures. It is a global ID attached to an instantiated structure. By using this, what was originally the same structure is duplicated at most once in a PE even if it is imported from different PEs more than once.

When a **%read** message is sent to a PE for a structure with a structure ID, only the ID is returned in the **%answer_value** message. If the **%read** sending PE receives only the ID, it looks up the *structure ID hash table* (Figure 9) with the ID to search for the structure address if the PE already has the structure. If it is not found, a **%read** message is sent again to copy it. Another hash table, the *structure address hash table*,

[2] If an imported pointer is copied, the MRB of both pointers, the original and its copy, are turned on so that the import table entry is not released when one of the pointers becomes garbage.

[3] Messages sent to the garbage collecting PE are buffered in the reserved memory area by the PE.

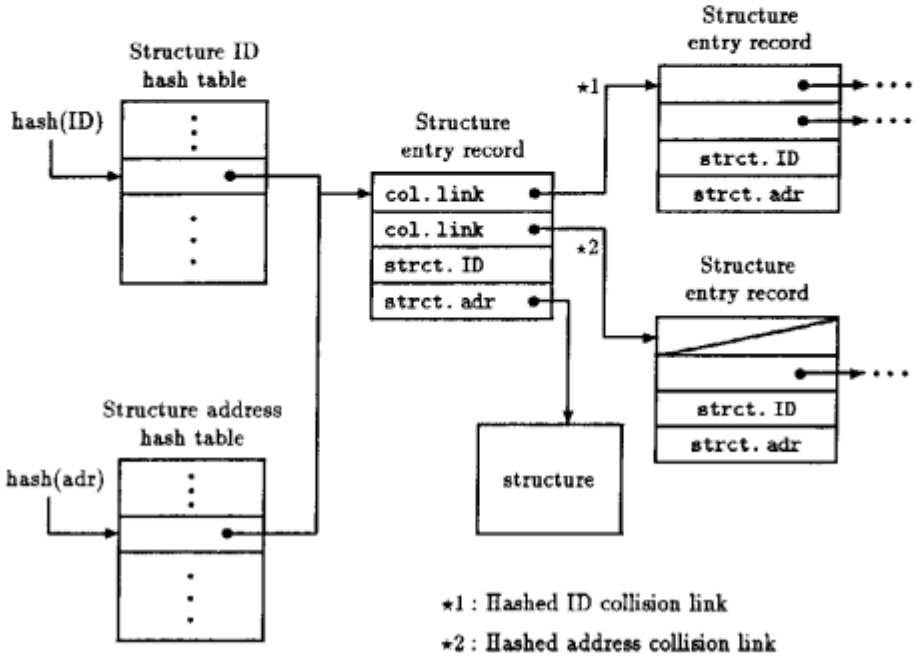

Figure 9: Structure Entry Records and Hash Tables

is used to get the structure ID from the structure address when the PE returns the ID instead of the structure itself in the `%answer_value` message.

The global structure management mechanism is used for the program code, because code pieces in a program are connected to each other and references to the same code piece can be imported from various PEs. The problem in this scheme is the collection of the garbage ID, which needs a kind of global GC scheme, and is not implemented yet.

5.4.8 Program Code Management

KL1 programs are described as a collection of *modules* which may contain several KL1 predicates. A module is the unit of compilation and is also used as the unit of code distribution to PEs.

The predicate calls within a module are represented by relative pointers. As they are constants in the module, they are free of maintenance when the module is moved in a local GC or is copied to send to other PEs. Only the inter-module predicate calls use absolute address pointers which require address maintenance in data transfer.

Absolute address pointers in a module are gathered at the top region in the module to reduce the size of sweeping for maintenance in local GC. The rest (and the greater part) of the module contains only atomic data, that is, KL1-B instructions (with the relative address operand if any) followed by their full word constant operands if any.

On a distributed memory multiprocessor with many PEs, an on-demand loading mechanism for the program code is essential to save the memory area in the system. The following is how this is realized.

When a goal is thrown to another PE, the code address for the goal is encoded as a tuple of <*module*, *offset*, *structure ID*>, where *module* is an external reference to the code module in the exporting PE, and *offset* is the code location in the module.

At the destination PE, *structure ID* is used to check whether the same module exists or not. If it does, the code address is calculated with *offset*. If it does not, a **%read** message is sent to the exporting PE. The received goal is hooked on a newly created variable which will receive the module.

6. Programming Environment Support

The Multi-PSI together with the parallel inference machine operating system, *PIMOS* [Chik88a], provides a programming environment for developing application programs of practical sizes. It offers various functions for parallel software debugging and its performance debugging. The KL1 implementation supports these functions so that the overhead is minimized.

6.1 Trace and Spy

The implementation supports the tracing and spying of goals. KL1 goals are colored with **normal** or **traced**. The reduction of a "traced" goal results in a *trace exception* with the information of all its subgoals to be forked. The monitor process for the shōen can report them and the user can specify which subgoals are to be traced next.

The spy function is also realized by coloring goals. The user can fork a goal with **spying** color with a spied predicate information (the name and arity of the predicate to be spied). This information is inherited to the child goals. If a subgoal for calling the spied predicate is created, a *spy exception* is raised reporting the parent goal information.

6.2 Deadlock Detection

One of the most common and distressful situations in debugging parallel programs is deadlock. Often, deadlock is known to the user only after all other executable goals have terminated, at which time there remain few clues to find the cause of the deadlock. The basic functions for deadlock should be to detect deadlocks and to show their causes.

In the KL1 implementation, deadlocks inside a PE are all detected at a local copying GC [Inam90a]. Typically, when a program falls inactive and seems to be deadlocked, the user will invoke the GC.

At the GC, the suspended goals which are not reachable from active goals are perpetually suspended. These goals are copied to the new semi-space and examined further by traversing the "causality graph"(goals and variables connected by reference and hook pointers) to find maximal goals in the causality. *Deadlock exceptions* are raised to the report stream for each maximal goal. This concise information usually helps locate the real cause of the deadlock.

The MRB scheme enables early deadlock detection. This is a common merit of reference counting scheme. If an MRB-off pointer that points to an unbound variable

with suspended goals hooked on it is found to be discarded at the commitment of a clause, the suspended goals are known to become perpetually suspended. Reporting a deadlock exception on the spot is generally more helpful than postmortem detection in a local GC, because the clause in execution may be precisely the cause of the deadlock.

6.3 Profiling

Two kinds of profiling facilities are provided. The *Shōen Profiler* counts the number of reductions of each predicates in the shōen. The control of profiling such as start, stop and collect data are done via the control stream of the shōen.

The *Processor Profiler* measures various dynamic characteristics at each processor. It records the time stamps of idling and local GCs, counts each inter-PE message frequency and the total time spent for message handling, records the user-defined events with time stamps, etc. A predefined predicate is used to start and stop the profiling.

7. Evaluation

This section shows the measurement results of the costs of inter-PE operations in the system. Actual communication overheads in two benchmark programs are also shown.

7.1 Cost of Communication Primitives

In typical KL1 programs, fine-grain processes (*goals*) communicate with each other via logical variables. In the Multi-PSI system, goal distribution is realized by `%throw_goal` message, inter-PE reading of values is realized by `%read` & `%answer_value` protocol.

Figure 10 shows the cost of handling those three messages at both sending and receiving PE. In Figure 10, `Copy_RPKB` stands for the time for copying a message packet from the hardware buffer to the software buffer when a message is received[4]. `Basic message handling routine` corresponds to the routine stated in 3.3, that is, putting a byte-serial message in the Write Buffer cutting from tagged words in encoding, or getting a byte-serial message from the Read Buffer and constructing tagged words for decoding.

`Encode/decode KL1 term, etc.` is for encoding internal KL1 term into a word sequence for export (by translating each element of the term to appropriate representation if it is a structured data and the encoding level is 1, see 5.4.1), or decoding a word sequence in a message packet form into the internal representation of KL1 term for import.

`Send_throw` (a) shows the cost of sending a 65 byte `%throw_goal` message for a three argument spawned goal. 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 to a goal stack.

The bar graphs (c), (d), (e) and (f) describe the cost of sending and receiving a `%read` message and `%answer_value` message. The returned data in this case is a list whose CAR is an atomic data and the CDR is an external pointer. `%read` and `%answer_`

[4] `Copy_RPKB` is executed when the hard buffer (Read Buffer) doesn't have much room to store further messages and the overflow is expected. `Copy_RPKB` is omitted most of the time.

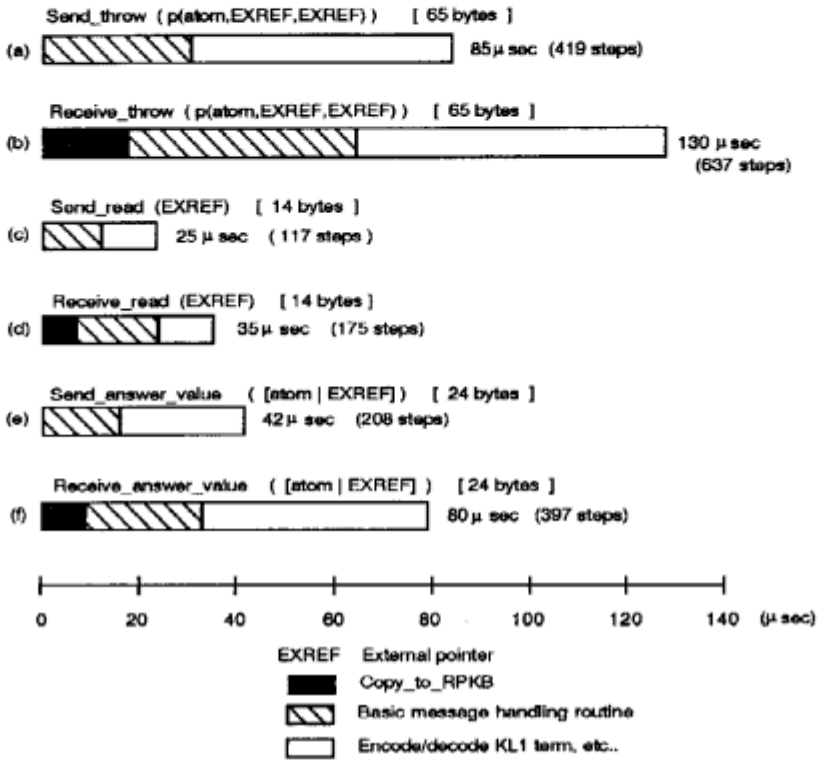

Figure 10: Message Handling Cost

value are the two most frequent messages in typical KL1 programs. The costs for **%unify** message, though not in the figure, are almost same as those of **%answer_value**.

In all operations, one third to half of the time is spent by **Basic message handling routine**. For example, it takes about 12 μs in **Basic message handling routine** for handling 14 bytes in (c). It is four times more than that for receiving them from the network channel. It proves that the hardware support for message composing/decomposing is not sufficient and forces the microcode to do much work.

Encode/decode KL1 term, etc. occupies more than half of the time and same thing can be said compared with the network bandwidth. Possible support hardware is for manipulating export/import tables and their hash tables. It can be a dedicated small processor.

7.2 Measurements of Benchmark Programs

7.2.1 Benchmark Programs

The followings are the two benchmark programs used here.

Table 2: 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

- **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 [Furu90a].
- **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. The vertices are represented by KL1 processes, and they exchange shortest path information along the edges.

7.2.2 Message & Reduction Profile

Table 2 shows the execution time, the reduction and message frequency, etc. 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 can be calculated from these figures. Relative to the 5 Mbyte/s network channel bandwidth, the average traffic on a channel is very small: 0.08% (Pentomino) and 0.3% (Bestpath) of the bandwidth.

7.2.3 Communication Overhead

By counting the number of executed steps and logging the time of entering and exiting from idle status at each PE, the execution time is broken down as follows; (1) the total executed steps for reductions (**Computing**) and (2) for message handling (**Msg handling**), both of which exclude cache miss penalty, (3) the total cache miss penalty (**Cache miss**), and (4) the total idling time (**Idle**).

Figure 11 shows the average of the above figures of all PEs and the resultant speed-up shown with the ideal one. In Pentomino, the overhead by the message handling and cache miss is very small and the speed-up degradation was mainly due to idling time.

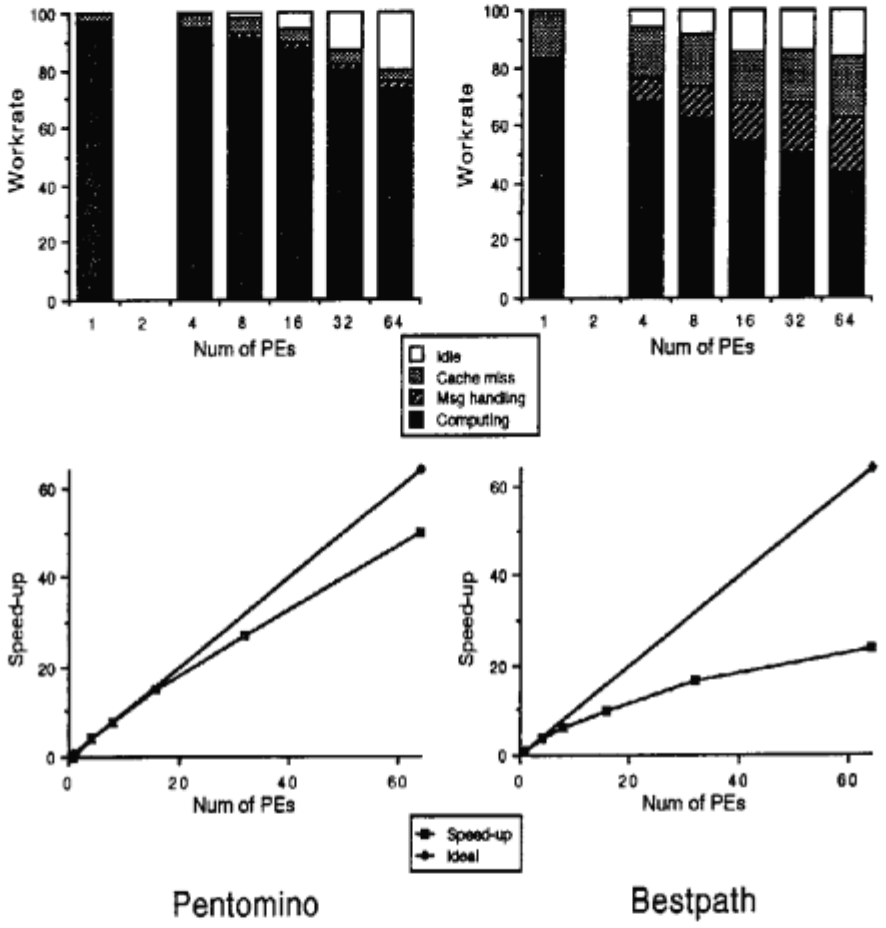

Figure 11: Decomposition of Processor Time and Speed-up

In Bestpath, though the idling ratio of 64 PEs is smaller than that of Pentomino, the computing ratio is rather low. Not only the overhead of the inter-PE communication, but the cache miss penalty is very large because of large working set. As the number of PEs grows, the grid graph is divided into smaller blocks (5×5-grid block for 64 PEs, 16 different blocks at each PE) to keep the workrate high, and it makes the percentage of communication time larger. The message frequency is expected to be proportional to the total length of the block boundaries, which is proportional to the square root of the number of PEs. This is supported by Table 2.

The cost of communication primitives is rather high compared with that for local reduction (25 μs at 40 KRPS). However, in two benchmark programs examined here,

performance degradation by communication overhead was small. The network traffic was very small relative to the hardware bandwidth. As the result, it is expected that the system can scale up to 1K ($2^5 \times 2^5$) PEs range without the network becoming the bottleneck in performance if the hardware performance of each element remains same as the Multi-PSI.

8. Conclusion

This chapter described the design issues and various techniques in implementing KL1 on a distributed memory multiprocessor, the Multi-PSI. Several evaluation results on the inter-PE communication with two benchmark programs were also shown.

In addition to the operating system, PIMOS, several application programs such as a protein sequence alignment program, a case-based legal reasoning system and LSI-CAD programs (a logic simulator, a cell placement program, and a routing program) have been developed and are now running on the Multi-PSI.

The Multi-PSI now has its successor, called PIM/m. The maximum configuration of the PIM/m is 256-PE (16×16 mesh). The PE of the PIM/m is more than twice faster and four times smaller than that of the Multi-PSI by utilizing CMOS VLSI technology. The evaluation result of the Multi-PSI network shows that the major part of inter-PE communication cost is that of the message handling by the processor rather than the network delay. Therefore, the network of the PIM/m was designed to have roughly the same performance as that of the Multi-PSI, while a hardware support for message composing/decomposing is added to the processor.

The KL1 implementation described here has been ported to the PIM/m, and will be used for future research on parallel programming and processing for large scale multiprocessors.

Acknowledgments

I would like to thank all the people at ICOT, Mitsubishi Electric and other related companies in the fifth generation computer project, who collaborated in the design and development of the Multi-PSI and its KL1 implementation.

References

[Ali86a] K. A. M. Ali and S. Haridi. Global Garbage Collection for Distributed Heap Storage Systems. International Journal of Parallel Programming, 15(5): 1986.

[Chik87a] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Chik88a] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Deut76a] L. P. Deutsch and D. G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. Commun. ACM, 19(9): 1976.

[Furu90a] M. Furuichi, N. Ichiyoshi and K. Taki. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[Goto88a] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Ichi87a] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Ichi88a] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Inam90a] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[Kimu87a] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[Know65a] K. C. Knowlton. A Fast Storage Allocator. Commun. ACM, 8(10): 1965.

[Lieb83a] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. Commun. ACM, 26(6): 1983.

[Nakj88a] K. Nakajima. Piling GC — Efficient Garbage Collection for AI Languages. In *Proceeding of the IFIP WG 10.3 Working Conference on Parallel Processing*, 1988.

[Nakj89a] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[Naks87a] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[Roku88a] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima.
An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. I, 1988.

[Taki88a] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI system. Programming of Future Generation Computers, Elsevier Science Publishers B.V. (North-Holland), 1988.

[Ueda86a] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[Ueda90a] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. The Computer Journal, Vol.33, No.6, 1990.

[Warr83a] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

[Wats87a] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proceedings of Parallel Architectures and Languages Europe*, June 1987.