ICOT Technical Report: TR-531

TR-531

Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI

by
K. Nakajima & N. Ichiyoshi

February, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI

Katsuto Nakajima†

Nobuyuki Ichiyoshi‡

†: Mitsubishi Electric Corporation

‡: Institute for New Generation Computer Technology

Abstract

The Japanese fifth generation computer project has the target of building a highly parallel inference machine (PIM) on which to construct large-scale knowledge information systems. We developed a prototype, the Multi-PSI, for the purpose of providing a practical tool for research and development of parallel non-numeric software. It also served as a testbed for implementation of concurrent logic language KL1 on a loosely-coupled multiprocessor. The design rationale was to obtain a high overall performance, taking account of garbage collection overhead, and to decentralize management information for scalability.

This paper gives the measurement results of inter-processor operations in the system, both in absolute terms (cost of primitive operations) and in relative terms (rate of communication overhead in benchmark programs), and determines the bottlenecks in the performance of inter-processor operations.

The measurements in the benchmark programs show that the bandwidth of the network hardware is much larger than the actual message traffic, and it is expected to remain so for larger configurations such as a 32×32 mesh network (1,024 processors).

1 Introduction

The Japanese fifth generation computer project has the target of building a highly parallel inference machine (PIM) on which to construct large-scale knowledge information systems. We developed a prototype machine, the Multi-PSI system [Taki 88, Nakajima 89], for the purpose of providing a practical tool for research and development of parallel non-numeric software. It also served as a testbed for implementation of concurrent logic language KL1 [Chikayama 88] on a loosely-coupled architecture.

The Multi-PSI is a non-shared memory multiprocessor, whose processing elements (PEs) are the CPUs of the personal sequential inference (PSI) machine [Nakashima 87]. There was an earlier version, the Multi-PSI/V1, but in this paper the name Multi-PSI represents only the Multi-PSI/V2.

Up to 64 PEs are connected by an 8×8 mesh network with automatic routing capability. The two-dimensional mesh network has a dense and simple implementation. It is also very scalable in that network node degree stays a constant (4) as the number of nodes increases. Thus, the Multi-PSI architecture expand to larger-scale configurations (for instance, more than 1,000 processors) with little modification.

A distributed KL1 implementation was developed on the machine. It is written in the 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.

This paper gives the measurement results of inter-processor operations in the system, both in absolute terms (cost of primitive operations) and in relative terms (rate of communication overhead in benchmark programs), and determines the bottlenecks in the performance of inter-processor operations.

Section 2 of the paper outlines the Multi-PSI hardware, the concurrent logic language KL1 and its parallel implementation, Section 3 reports the evaluation results, and Section 4 concludes the paper.

2 Overview of the Multi-PSI System

2.1 Hardware

2.1.1 Processing Elements

The processing element (PE) of the Multi-PSI is the CPU of the PSI-II [Nakashima 87]. 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 nsec. It has up to 16 Mwords of local memory and a 4-Kword direct-map cache memory.

2.1.2 Network Controller

Each PE is paired with a specially designed network controller to support message passing communication between PEs. It 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 nsec and the bandwidth of each channel is 5Mbytes/sec. A 48-byte buffer (*Output Buffer*) is installed at each output channel to retain messages when the destination node is busy. A 4-Kbyte buffer (*Write Buffer* and *Read Buffer*) is installed at each input and output channel for the PE of the node 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 as far as the transmitting channel is not 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 an automatic routing function. It can route messages according to the PE number in the message header. 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*

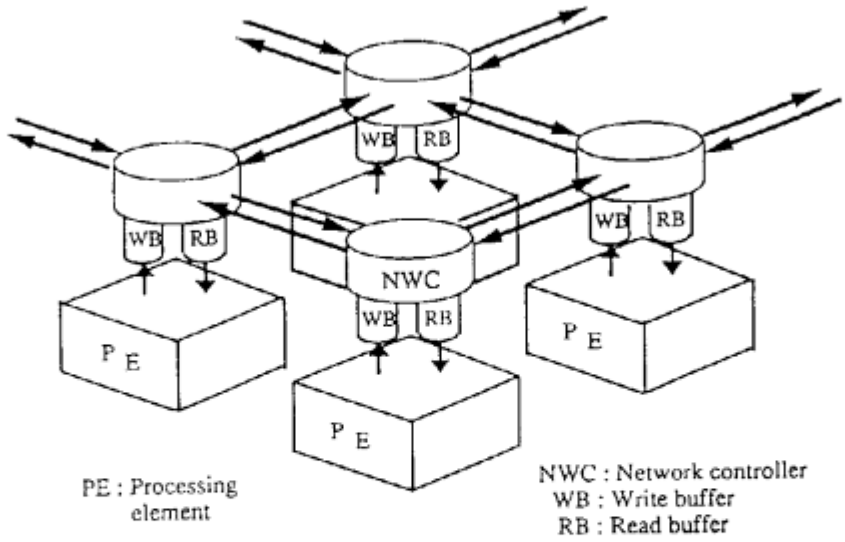

Figure 1: Processor Inter Connections of the Multi-PSI System

until the distance in the coordinate becomes zero, then to transmit along the *y-coordinate*. This prevents network deadlock as it is impossible to make a circle of nodes waiting for one another.[1]

2.2 Parallel Language KL1

KL1 (kernel language version 1) is a stream AND-parallel logic programming language based on Flat GHC[Ueda 86]. A KL1 program is made up of a collection of guarded horn clauses, whose form is:

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

where H is called the *head*, G_i the *guard goals*, and B_i the *body goals*. The vertical bar (|) is called the *commitment operator*. The guard part unification is to wait for the instantiations to variables (synchronization) and to test them. When the guard unification succeeds, the control proceeds beyond the commitment bar and the body goals are reduced concurrently, those of which may communicate each other through their common variables.

[1]Note that even with this strategy, a deadlock may occur if a PE fails to take in all the messages directed to it.

KL1 body goals can have *pragmas* as the meta-programming functions.

(1) Priority pragma ($\ldots, B@priority(Prio), \ldots$) : To specify scheduling priority, which contributes to efficient problem solving.

(2) Throw goal pragma ($\ldots, B@processor(PE), \ldots$) : To move a goal to another PE for load distribution.

2.3 Distributed KL1 Implementation

2.3.1 Execution Mechanisms in a PE

The following is the execution model of this implementation in a PE. Reduced KL1 body goals are stored in the current priority goal pool, or other priority pools if priority pragmas are specified. KL1 execution is done by choosing a goal from the highest priority pool. If the goal is waiting for a variable, the goal is hooked in the variable (non-busy wait) and would be stored back in the former goal pool when the variable is instantiated.

A KL1 program is compiled into KL1-B [Kimura 87] code which is similar to WAM [Warren 83] for Prolog. KL1-B code are interpreted by microcode of the PE. A list concatenation program in KL1 runs by 128 KRPS (Kilo Reduction Per Second) on a PE.

2.3.2 Inter-processor Processing Mechanisms

Inter-processor communication can be categorized into two, inter-processor data management and inter-processor goal management (control of the distributed computation). This paper will not describe the latter, which can be found in [Ichiyoshi 87], [Rokusawa 88], and [Nakajima 89].

The following is the outline of inter-processor data management. The details are in [Nakajima 89].

A KL1 goal with a throw goal pragma is shipped out to another PE by `%throw_goal` message. As the single assignment semantics of KL1 allows data copying, the atomic arguments of the goal are copied into the message. However, if the argument is a structure, it is encoded as an *external reference pointer* instead of copying it, because the goal might

not need the data. The structure elements are transferred lazily on request by %read messages. To generate an external pointer is called *exporting*, and to receive the external pointer is called *importing*. Unbound variables are always exported as an external pointers to retain the identity.

KL1 has a fine grain parallelism and is suitable for massively parallel computation, though its implementation tends to consume memory very rapidly. Efficient garbage collection (GC) is one of the most important issues for KL1 implementation. To collect all the garbage in the system, a *global GC* would be necessary, which must be performed by cooperation of all the PEs. However, the global GC has the serious disadvantage of forcing all PEs to stop program execution. It takes a very long time, up to the longest path of data references in the system multiplied by the network communication delay. Therefore, it is better to perform GC locally (*local GC*) as far as it can reclaim enough memory.

To perform local GC, we employed two level address spaces; the higher one is for external pointers, the lower one is for inside PE. All the exported data must be known by the PE so that they are not reclaimed as garbage. For this purpose, the *export table* keeps internal addresses of all the exported data in a PE. All the external pointers point to the entries of the table from outside the PE(Figure 2). The external pointers are represented in the form $< n, e >$, where n is the exporting PE number and e is the entry position in the export table. This is the address (notation) in the higher level address space. This address is not affected (changed) by local GCs. The export table is the translation table from the $< n, e >$ form to the internal address of the exported data. The *export hash table* is also installed to translate from internal address to external one to re-export the same data, which prevents unnecessary duplication of the data between local memories.

To reclaim garbage cells pointed to by the export table, the entries of the table must be deleted when they become garbage. A reference counting scheme based on the weighted reference counting (WRC) [Watson 87] is used for the incremental GC of the entries. This is called the *weighted export counting (WEC)* method [Ichiyoshi 88].

In this scheme, a weighted count of a positive integer, called `WEC`, is kept on the both

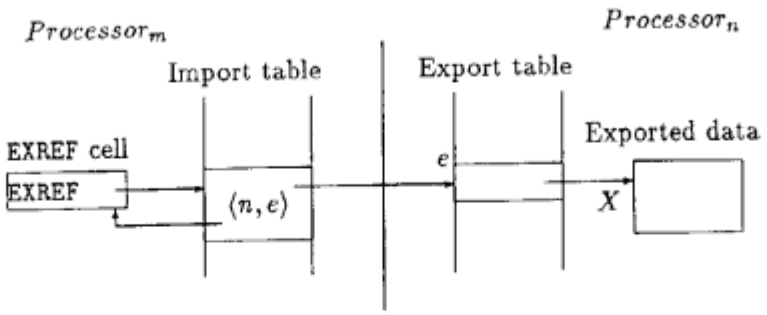

Figure 2: External Reference and Export/Import Table

export and import sides. The system operates keeping the sum of the WEC in the messages in transit and at the importing PEs equals to the WEC at the exporting PE. This scheme has the following advantages:

- When an imported pointer is copied to another PE, the WEC is split. No message is sent to maintain the WEC of the exporting PE.
- There is no racing problem.

To keep WEC on the importing side, we have the *import table*(Figure 2). The WEC of the imported pointer which is no longer used known by the intra-processor incremental GC using MRB [Chikayama 87] or a local GC is returned to the exporting PE by a `%release` message. We have also the *import hash table* to translate from an external address to the internal address of the cell representing the external pointer for re-importing.

2.4 Inter-processor Messages

2.4.1 Messages Handling

A low level microcoded routine in the PE is responsible for basic handling of the messages to and from the network controller. It performs composition and decomposition of message packets such as handling message header and tail, and arranging a 32-bit data in the processor to/from four byte serial data in the network controller. The operations in this routine are independent from the implementation of KL1.

When a series of message bytes reach to the node, they are queued in the Read Buffer by the network hardware. The micro routine is invoked by an interrupt when the message

Table 1: Inter-processor Messages (part)

Message	Note
%throw_goal	KL1 goal distribution
%read	request to read the value of an external pointer
%answer_value	response to a %read message
%unify	request to unify with an external pointer and an argument
%release	return WEC value of an external pointer

tail is queued. 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.

2.4.2 Summary of Inter-processor Messages

Table 1 and the followings summarize the typical inter-processor messages, some of which have already been introduced in the previous sections.

- A %throw message transfers a KL1 goal with its arguments.
- A %read message is sent to the exporting PE when a guard unification is attempted with an imported external pointer. A %read message has two arguments: the external pointer address to read and a new external pointer pointing to the external pointer cell where to be replied to.
- An %answer_value message is immediately replied to the %read message if the contents of the exported 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 received %read message is

suspended by hooking it in the variable. The arguments of %answer_value are the destination to reply and the value to reply.

- A %unify message is to request a body unification of two arguments, one of which is an external pointer exported by the PE where the message is directed to.
- A %release message carries WEC of an external pointer to return to the exporting PE. %Release messages are issued at intra-processor incremental GCs and local GCs.

3 Measurements and Evaluation

The major concern in evaluating loosely coupled multiprocessors would be the cost of inter-processor communication, the overhead by it, the traffic of the connection network and the workrate of each processors. This section describes the evaluation results of them. The Multi-PSI systems with 1, 4, 8, 16, 32 and 64 PEs were used for the measurements.

3.1 Measurements of Primitive Inter-processor Operation Cost

3.1.1 Inter-processor Operation Cost and Analysis

Figure 3 shows the micro instruction steps obtained by microprogram traces. The time was calculated by multiplying the number of micro steps by 200 nsec (cycle time) and does not include the memory access overhead (cache miss penalty).

The costs of sending and receiving a %throw_goal message whose three arguments are an atom and two external pointers in a typical situation[2] are shown in Figure 3(a) and (b). It takes about 85μsec for sending such a goal and about 130μsec for receiving and storing it to a goal pool.

The time taken in the basic message handling routine is almost the same as that for Encode/decode KL1 term, etc., which consists of the steps for encoding/decoding the

[2]The PE receiving the message is assumed to already have the program code.

goal arguments according to their data types, for encoding/decoding other goal information such as code and priority, and for looking up the export/import table for the address translation. It takes 14% of the total cost of (b) to move a 65-byte from the Read Buffer to the Read Packet Buffer (Copy_to_RPKB).

As it takes only 13μsec (15% of the total cost of (a), 10% of (b)) to transmit a 65-byte message by the network controller, the large overhead exits both in the basic routine and the operation depending on the KL1 implementation almost half-and-half.

As shown in Table 2 in 3.2.2, the Pentomino program runs at about 40 KRPS (25μsec per reduction) on a PE. (a) and (b) are 3.4 to 5.2 times of that reduction cost.

Figure 3(c),(d),(e) and (f) describe the cost of sending and receiving a **%read** message requesting the contents of an external pointer and an **%answer_value** message answering the request. The returned data is a list whose CAR is an atomic data and the CDR is an external pointer.

It takes 25μsec to send a 14-byte **%read** message and 35μsec to receive it. It costs 42μsec to send a 24-byte **%answer_value** message and 80μsec to receive it. They are 1 to 1.7 times of the Pentomino reduction. A 14-byte or 24-byte message requires at least 2.8 or 4.8 μsec, which is 6 to 11% of the cost for sending/receiving each message. The make-up of the cost for these messages is quite similar to the **%throw_goal**.

3.1.2 Possible Tune-ups

Copy_to_RPKB can be omitted by decoding messages directly from the Read Buffer when there is enough room in the Read Buffer to get further messages. With this improvement, the cost for the **%throw_goal** can be reduced by about 13.5%. According to our recent estimation, the cost of sending it would be reduced by up to 15% and the cost of receiving it by up to 30 % (with the elimination of Copy_to_RPKB) by fully tuning up the microcode, mainly in the basic message handling routine. For **%read** and **%answer_value** messages, our estimations of the tuning up effect are also 15 to 30 % of the current version.

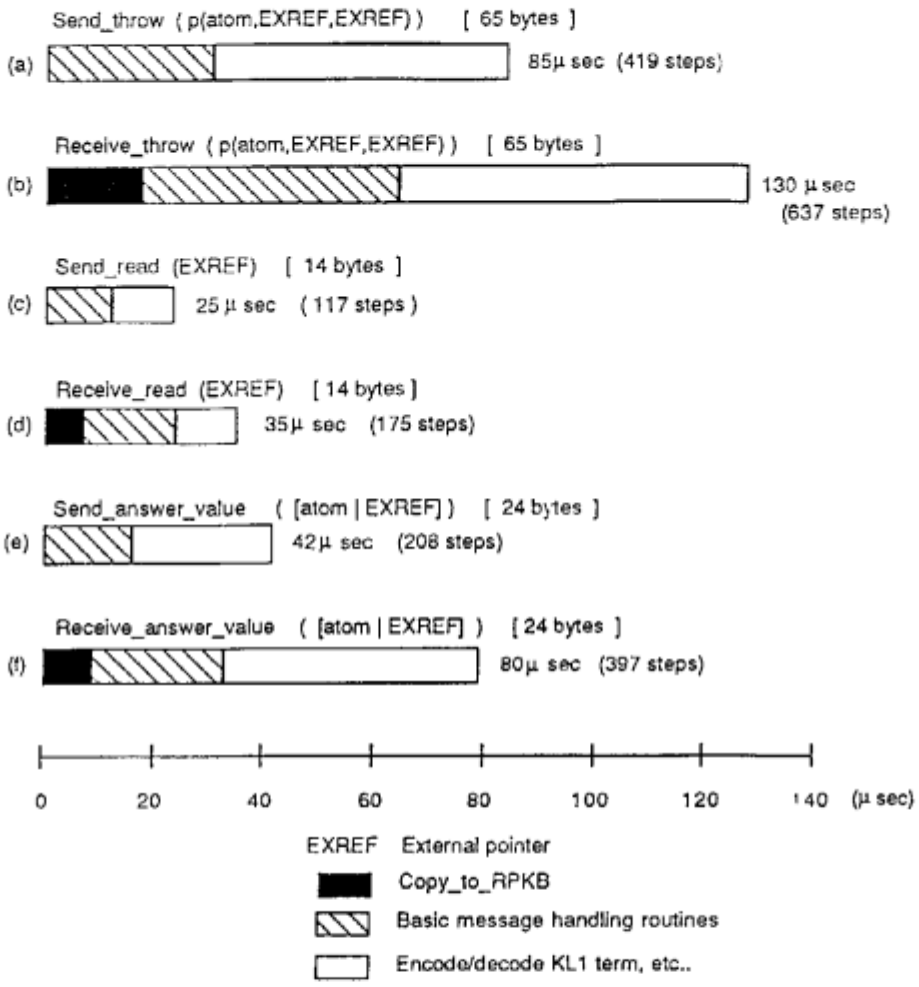

Figure 3: Message Handling Cost

3.2 Measurements of Inter-processor Communication by Benchmark Programs

The previous section gave the measurement results of individual inter-processor message handling costs. The network traffic and the impact of inter-processor communication in total execution time depend on programs that are run. We took measurements for two different types of benchmark programs.

3.2.1 Benchmark Programs

The followings are the two benchmark programs we used.

- **Pentomino:** A program to find out all solutions of a 5×8 packing piece puzzle. That is, it finds all the ways of packing a 5×8 rectangular box by ten various shaped pieces, each made up of four unit squares.
- **Bestpath:** A 160×160 grid graph is given together with randomly generated non-negative edge costs. A program determines the lowest cost path from a vertex to all other vertices of the graph (single-source shortest path problem).

The Pentomino program does an exhaustive search of an OR-tree of possible piece placements. One master PE starts at the root node (corresponding to the empty box) and searches the tree down to a certain fixed depth, and evenly distributes the subtrees below that depth to all PEs by a dynamic load balancing scheme [Furuichi 90]. When 64 PEs are used, they are divided into eight processor groups each consisting of eight PEs. At the higher level, the processor groups are balanced by distributing super-subtasks to them, and at the lower level, within each processor group, the load of the processors are balanced by distributing subtasks to them.

The Bestpath program generates the grid first, and then performs a distributed shortest path algorithm. The grid graph is divided into sixteen 40×40 superblocks, each of which is again divided into so many blocks as the Multi-PSI PEs and statically mapped onto them. For example, when 64 PEs are employed, each PE is assigned sixteen 5×5 blocks from the sixteen superblocks. The inter-vertex cost information packets in the

distributed algorithm is represented by goals to utilize the priority mechanism of the KL1 implementation. We took measurements of the second phase of the program.

3.2.2 Message Profile

Table 2 shows the total number of the messages by message type and their average length. The total number of reductions and the total reduction speed are also listed.

In Pentomino, the `%read`, `%answer_value` and `%release` messages are dominant. In Bestpath, the `%unify` and `%throw_goal` messages are also frequent.[3] The message sending rates in program execution on 64 PEs are 1/88 messages per reduction for Pentomino and 1/6 messages for Bestpath.

3.2.3 Communication Overhead

This section analyzes to what extent inter-processor communication overhead degrades the overall performance of the benchmark programs.

We break down the execution time in the following way. Each PE of the Multi-PSI has a built-in hardware for counting the number of steps executed at specified micro addresses. The PEs also have a calendar clock, all of which are simultaneously initialized when the system starts up. By using them and logging the time of entering and exiting from idle status at each PE, we can measure the following items.

(a) Total micro steps that are run outside of the idling loop (for each processor)

(b) Total micro steps that are run in message handling routines (for each processor)

(c) Total execution time

(d) Total idling time (for each processor)

The *workrate* of the PE is defined by $(c - d)/c$. The difference between c and $(a + b) \times cycletime$ ($cycletime$ = 200nsec) is due to cache miss penalty. $b \times cycletime$ is the communication overhead less cache penalty involved.

[3]This is because inter-vertex packets that cross processor boundaries are translated into goal throwing.

Table 2: Message Frequency and Reductions

Pentomino (39.3 KRPS on 1 PE)			
message	4 PEs	16 PEs	64 PEs
`read`	10,408 (27.7 %)	24,746 (31.9 %)	30,736 (32.4 %)
`answer_value`	10,408 (27.7 %)	24,704 (31.9 %)	30,581 (32.3 %)
`release`	6,843 (18.2 %)	12,333 (15.9 %)	13,641 (14.4 %)
`unify`	2,379 (6.3 %)	7,504 (9.7 %)	11,129 (11.7 %)
`throw`	1,191 (3.2 %)	1,677 (2.2 %)	1,768 (1.9 %)
etc.	6,396 (16.9 %)	6,542 (8.4 %)	6,895 (7.3 %)
total	37,625 (100 %)	77,506 (100 %)	94,750 (100 %)
avg. msg length	21.1 bytes/msg	20.4 bytes/msg	20.2 bytes/msg
total time	54,634 msec	14,615 msec	4,345 msec
total reductions	8,317 K	8,332 K	8,340 K
reductions/sec	152.2 KRPS	570.1 KRPS	1,919.4 KRPS
reductions/msg	221	108	88
msg bytes/sec	14.5 K	108.1 K	440.5 K

Bestpath (23.4 KRPS on 1 PE)			
message	4 PEs	16 PEs	64 PEs
`read`	12,396 (27.5 %)	28,780 (27.8 %)	66,899 (27.7 %)
`answer_value`	10,156 (22.6 %)	23,980 (23.2 %)	56,980 (23.6 %)
`unify`	8,439 (18.8 %)	19,190 (18.6 %)	43,370 (18.0 %)
`release`	6,198 (13.8 %)	14,390 (13.9 %)	33,446 (13.9 %)
`throw`	6,198 (13.8 %)	14,390 (13.9 %)	33,446 (13.9 %)
etc.	1,618 (3.5 %)	2,694 (2.6 %)	7,263 (2.9 %)
total	45,005 (100 %)	103,424 (100 %)	241,404 (100 %)
avg. msg length	27.0 bytes/msg	27.2 bytes/msg	27.0 bytes/msg
total time	10,655 msec	4,062 msec	1,691 msec
total reductions	987.7 K	1213.6 K	1,505.2 K
reductions/sec	92.7 KRPS	298.8 KRPS	890.1 KRPS
reductions/msg	21.9	11.7	6.2
msg bytes/sec	114.0 K	692.5 K	3,854.3 K

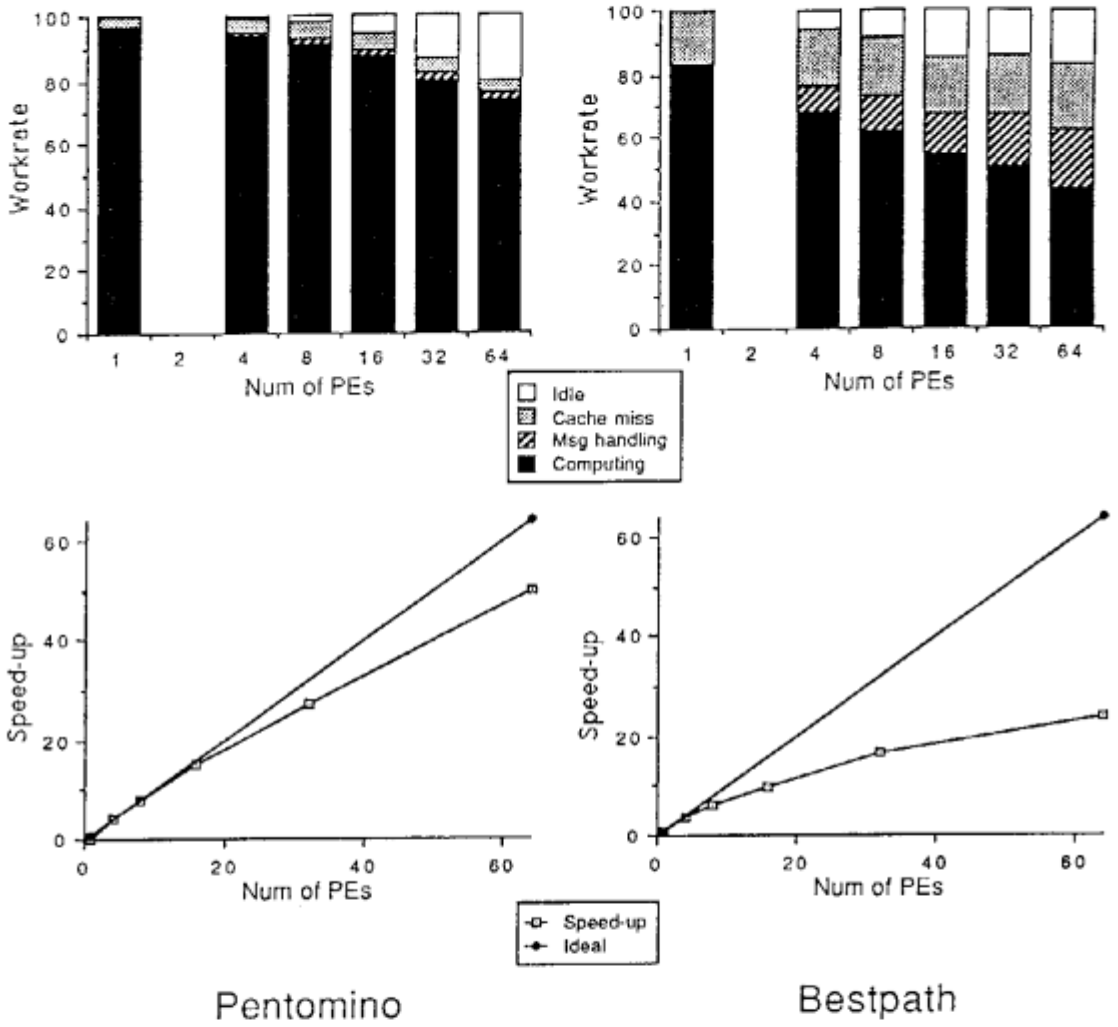

Figure 4: Workrate and Speed-up

The processor time breakdown is given in Figure 4. The breakdown in the bar graphs is into: computing ($a \times 200$ nsec), message handling ($b \times 200$ nsec), cache miss penalty ($c - (a + b) \times cycletime$) and idling times ($d$), averaged for all PEs.

In Pentomino, the overhead by the inter-processor communication and cache miss is very small and the speed-up degradation was mainly caused by idling time, which is small but still exists.

In Bestpath, though the idling ratio on 64 PEs are smaller than that of Pentomino, the computing ratio is rather low. Not only the overhead of the inter-processor communication, but the cache miss penalty is very large due to large working set. As the number of PEs grow, the percentage of communication time also grows. More precisely, the number of inter-processor vertex-to-vertex packet sending is expected to be proportional to the total length (in number of vertices) of the block boundaries, which is proportional to the square root of the number of PEs. This is supported by the total number of messages (Table 2) and by the measurement of communication time. This means, if we run the same program for a much larger graph, the relative communication overhead will decrease.

In summary, the performance degradation by communication overhead is small for the Pentomino program, and is expected to be so for OR-parallel type of problems. For the Bestpath program, the relative communication overhead grows as the graph is decomposed into smaller blocks, but it should decrease for larger graphs. Nonetheless, message handling mechanism must be optimized so that the graph can be decomposed into smaller blocks to increase processor utilization rate without large communication overhead.

3.3 Evaluation of Network Hardware

3.3.1 Network Traffic in Benchmark Programs

In Pentomino, since intra-processor-group messages are dominant, the average message traveling distance (the number of hops of the network channels) is estimated to be about 2. The average message traffic rate per channel on 64 PEs is estimated to be:

$$\frac{441 \text{ Kbytes} \times 2}{(\text{number of channels}) = 224} = 3.9\text{Kbytes/sec}^4$$

which is 0.08% of the 5Mbytes/sec bandwidth of a channel and extremely small. Message sending and receiving centers at each subtask- distributor PE in the processor groups, but the amount is eight times the average, which is still very small.

In Bestpath, the average hops is estimated at a little more than 1, since most inter-block messages are from one PE to an adjacent PE. Therefore the average message traffic is calculated as:

$$\frac{3,854 \text{ Kbytes} \times 1}{224} = 17.2 \text{ Kbytes/sec}$$

which is 0.3% of the channel bandwidth. As PEs are mapped statically to the block of the problem grid, there must be hot spots along the wavefront of short pass messages traversing the processor plane, which makes the channel busier by a factor of the square root of the number of PEs. Still, the real traffic rate would be very small.

This means the network hardware has far too much capacity than required, at least for the two benchmarks.

3.3.2 Estimation for Larger-Scale Networks

In this section, we give an estimation of the network traffic on a more scaled-up machine, for instance, up to 1,000 PEs. We assume to execute a fine grain parallel program, in which the average message frequency at each PE is 1/6 messages per reduction that is the same as in the Bestpath on 64 PEs. It corresponds to 27bytes×(890KRPS/64)/6 =63Kbytes/sec.

We also assume the inter-processor communications are performed randomly to take account of the defect of the two-dimension network topology, where the number of message traveling distance is almost proportional to the square root of the number of PEs.[5]

[4](number of channels)=$4 \cdot L \cdot (L-1) = 224$ where $L = \sqrt{\texttt{(number of PEs)}} = 8$.

[5]$2 \cdot (L-1) \cdot (L+1)/(3 \cdot L) = 21.3$ *where* $L = \sqrt{\texttt{(number of PEs)}} = 32$. (Assuming the processors are aligned in a square.)

This is a pessimistic assumption compared with the message hops in Pentomino and Best-path as described in 3.3.1. The total message frequency in the system is proportional to the number of PEs. The number of channels[6] that enlarges the network bandwidth of the system is proportional to the number of PEs. Thus, the average channel traffic is proportional to $(\sqrt{N} \cdot N)/N = \sqrt{N}$ where N is the number of PEs.

The average channel traffic for 1,024 PEs can be calculated as:

$$\frac{63 \text{ Kbytes} \times 1,024 \times 21.3}{3,968} = 346 \text{ Kbytes/sec}$$

which is 6.9% of the 5 Mbytes/sec bandwidth and is still small. However, we have to note that unless the locality of the inter-processor communication is considered or unless the hot spots in the network are avoided by program, the network could be saturated quite easily.

4 Conclusions

We developed the Multi-PSI system as a prototype parallel inference machine. The mesh-connected architecture makes it a scalable multiprocessor. Much effort was made in the KL1 implementation on the machine (1) to minimize the overhead of garbage collection, (2) to reduce the number of inter-processor messages, and (3) to decentralize information, so that the system might be efficient and scalable.

This paper gave the measurements of inter-processor operations in the system, such as the cost of primitive operations, the message frequency and the rate of communication overhead in benchmark programs.

The cost of one message handling is 25μsec to 130μsec, which is roughly 1 to 5 times that of a reduction at the PE performance of 40 KRPS. The workrate analysis in the parallel benchmark programs tells us that even with rather large cost of communication, the overhead for inter-processor communication can be kept within one third of the operating steps. It can be reduced by a third by optimizing some of the lowest-level routines.

[6] $4 \cdot L \cdot (L-1) = 3,968$ where $L = 32$.

The network traffic is very low and the network hardware is not limiting the system performance in the current system. The network hardware with the same bandwidth per channel is expected to be good enough in a much larger-scale system, even with more pessimistic inter-processor communication patterns.

This study shows that the Multi-PSI architecture is indeed scalable, and communication overhead is very small (for OR-parallel problems) or tolerable (for message-intensive programs, if inter-process locality in the program is preserved by the mapping on the machine). However, our insight into loosely-coupled multiprocessors and parallel programming on them is still very limited. We will need to experiment with more programs with different runtime characteristics, and conduct more detailed measurements and analysis.

Acknowledgments

We would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research laboratory, Dr. S. Uchida, for giving us the opportunity to conduct this research. We would also like to thank the researchers of ICOT and the cooperating companies, who have worked with us in designing and implementing the KL1 system on the Multi-PSI/V2. Lastly, We appreciate the offer of the benchmark programs to Mr. Furuichi and Ms. Wada.

References

[Chikayama 87] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Chikayama 88] 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.

[Furuichi 90] 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.

[Goto 88] 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.

[Ichiyoshi 87] 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.

[Ichiyoshi 88] 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.

[Kimura 87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[Nakajima 89] 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.

[Nakashima 87] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[Rokusawa 88] 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.

[Takeda 88] 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 88] 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.

[Ueda 86] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[Warren 83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

[Watson 87] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proceedings of Parallel Architectures and Languages Europe*, June 1987.