The Evaluation of Parallel Inference Machines

Kouichi KUMON and Hiroyoshi HATAZAWA
Institute for New Generation Computer Technology
4-28, Mita 1-Chome, Minatoku, Tokyo 108, JAPAN
{kumon,f-hataza}@icot.or.jp

Abstract

In this paper, we study the performance of various implementations of KL1 processing systems, including the parallel inference machine (PIM) system and the KLIC system. PIM systems are built on dedicated parallel hardware developed in the FGCS project for efficient KL1 execution, while the KLIC system is KL1 ported to run on UNIX workstations. This difference makes the design policies of both systems quite different necessitating a quantitative analysis. KL1 is a parallel logic language running on hardware ranging from a single processor to several hundred processors. Thus performance measurements must cover this range.

Firstly we use a set of small benchmark programs for single processor performance. It includes a naive reverse program to compare and characterize those systems. And we take a close look at the execution of instructions in both the PIM KL1 processing system and the KLIC system. The PIM/p system is a convenient platform because both the conventional KL1 system and the sequential core of the KLIC system are available on it, and thus we can compare both implementations on the same base. We also compare both system using the `life` program, which shows quite a different aspect to the `append` execution.

Then, we evaluate the cluster performance for automatic load balancing. Some PIM models have a cluster structure, in which processors are connected by a common bus and share memory. In parallel processing in a cluster, automatic load balancing is available to reduce the burden of writing load distribution code. It is thus meaningful to investigate load balancing mechanisms and measure the speed-up in parallel performance. To do this, we make execution models for a very small program, examine the parallel performance, and estimate the distribution overhead in the cluster.

Finally, we investigate the distributed performance of the systems by using simple communication dependent benchmark programs. The PIM models use a dedicated network for communication between clusters/nodes. However as the KLIC system uses PVM for message passing primitives, we found that the performance of KLIC wholly depends on the PVM performance.

1 Introduction

ICOT has built five models of PIM, which have been used to develop application programs in the FGCS project. In the post-FGCS project, ICOT has been developing the KLIC system, which makes these application programs available on commercially-available workstations and parallel machines. These two KL1 processing systems differ in their design policies, and using the same scale for these two systems we can study their behavior.

In this paper, we measure and evaluate the performance of the five PIM models and the KLIC system with benchmark programs.

In Section 2, we briefly overview the five PIM models and the KLIC system, and in Section 3 we measure the single processor performance with a number of benchmark programs. In Section 4, we investigate the distribution strategy of the cluster-configured PIM and the characteristics of the strategy by making execution models. Of course, only cluster-configured PIMs are measured. In Section 5 we compare communication overhead inside a cluster and between clusters.

2 PIMs and KLIC

As we have already mentioned, the five PIM models have different architectures.

2.1 Configurations

First, we briefly introduce the features of each PIM model. PIM/m and the other PIMs (PIM/c, PIM/k, PIM/i and PIM/p) use the different base of implementation. PIM/m inherits its implementation from Multi-PSI, however, the other PIMs use the Virtual PIM[5] (VPIM) as a prototype KL1 processing system, but Multi-PSI and PIM/m use their own implementation for KL1 processing.

PIM/m[8] PIM/m is a revised model of the Multi-PSI machine, all processing elements are connected only by inter-processor networks; neither Multi-PSI nor PIM/m has shared memory. Memory can be safely accessed without implementing exclusive operations, resulting in faster system performance. Actually, the single processor performance of PIM/m is the fastest among the PIMs. PIM/m can have a maximum of 256PEs. The cycle time of the PIM/m is 65ns.

PIM/p[6] The PIM/p system has the largest configuration among the PIMs (up to 512PEs). Eight processors constitute a cluster using Illinois type snoop caches. The processor's cycle time is 80ns. The clusters are connected by a hyper-cube network with a worm-hole routing facility. On the PIM/p hardware, the KLIC system sequential core is also available. As we run the KLIC system on the bare hardware, very reliable measurements are available without the interference of interrups or virtual memory management. This makes a comparison between the conventional KL1 system and the KLIC system easier. We use the KLIC system version 1.511 and gcc-2.6.0 C-compiler customized to the PIM/p cross-compilation environment for measurement. We call the KLIC system on PIM/p as KLIC/p.

PIM/i[10] The processing element of PIM/i uses LIW instruction sets, and up to two instructions are executed simultaneously in every cycle. The cycle time is 160ns. The PIM/i also has a cluster structure from eight processors with write broadcast type snoop caches. PIM/i lacks network connection facilities, consequently an eight processor system is the maximum configuration.

PIM/k[10] In the PIM/k system, four PEs of PIM/k constitute a mini-cluster. Each PE has a first-level cache and shares a common bus, and four mini-clusters are connected by a secondary level bus with secondary caches. A global memory is attached to the secondary bus. Thus, all PEs of the PIM/k system can access any data directly by a global address.

PIM/c[7] The PIM/c cluster consists of eight processing elements, cluster controller and shared memory. The cycle time of the machine is 60ns. Totally 32 clusters are connected with a cross-bar network to reduce network latency.

KLIC[3][1] KLIC is not a hardware but a software implementation for the UNIX environment. A KL1 program is compiled into C source code, then a C-compiler compiles it to a load module. In the sequential performance measurement, we use both the Sparc Station 10[1] and the PIM/p hardware[2] as execution environments. In the distributed system measurement, we additionally use a SparcCenter 2000[3]. Currently KL1 features are not fully implemented on KLIC, and so some of the benchmark programs do not run on KLIC.

All PIMs except PIM/m use the Virtual PIM (VPIM)[5] written in PSL (PIM Specification descriptive Language) as their abstract processing system. This PSL representation is translated to each PIM architecture by the PSL compiler, and thus all VPIM based PIM use the same processing mechanisms.

2.2 PIM and KLIC Design Scheme

There is a major difference in the design scheme between the PIM processing systems and the KLIC system. Each PIM system uses dedicated processors designed for efficient execution of KL1 as follows:

- All data have both a value part and a tag part. Eight bits are used for the tag part, and thus, up to 256 different data types can be represented. One of the tag bits is used to represent the MRB status, 46 of the rest 128 types are used in the system. In the PIM processor, testing a tag value and jump on the result can be executed in one instruction using dedicated hardware.
- In the MRB management, propagation of the MRB status according to pointer dereferencing is needed. With a dedicated instruction, this MRB operation requires no additional overhead.
- KL1 language has no type declaration, and so dynamic data type checking is frequently needed and execution may differ depending on the types. Although there are many combinations of data type, actual execution is limited to a small portion of the combinations. Therefore, to reduce the static code size, it is helpful to provide a low-overhead subroutine-call mechanism. Usually, subroutines provide common operations of the KL1 processing system. The PIM/m system uses a micro-code architecture and the PIM/p system uses a macro-instruction mechanism for this purpose.
- The cluster-configured PIMs have an explicit locking/unlocking memory operation, unlike other conventional exclusive instructions such as compare-and-swap or test-and-set. This makes various exclusive operations feasible.

On the other hand, the main feature of the KLIC system is the pursuit of both portability and performance using a high-performance general-purpose processor and an optimizing C-compiler. Therefore, a tag data may use only the two bits which are not used for word addressing in byte-addressable machines. To check and jump on the tag data, an instruction sequence is needed, such as a logical-and followed by a comparison and a conditional jump instruction.

[1]The clock speed is 36MHz and the main memory size is 64M bytes.

[2]The clock speed is 12.5MHz and the main memory size is 256M bytes.

[3]The clock speed is 40MHz, and the memory size is 1G byte.

2.3 Garbage Collector

In the design of the garbage collector (GC), there is also a distinct difference between both systems. In all PIMs, the Multiple Reference Bit (MRB)[2] is employed to reduce memory consumption during reduction, and only when un-collected garbage fillup the heap, a copying type GC is invoked. Thus, if the MRB scheme can collect most of the garbage, the used heap area does not expand, hence, no copying-GC is invoked.

On the other hand, the KLIC system does not collect any garbage on the fly, because the tag bits have no room to contain an MRB bit, and without special hardware support to manipulate the MRB operations, the performance would be severely degraded. Consequently, all garbage must be collected by a stop and copy type GC. Unlike in PIM systems, the UNIX multiprocessing and virtual memory system prevent the KLIC system from using the physical memory directly. Hence, the KLIC system incrementally expands its heap when garbage cannot be sufficiently reclaimed. Thus, the final heap size and the number of GCs vary unexpectedly depending on execution conditions, such as input data. Therefore, performance of the KLIC system depends on the initial heap size. In the following measurements, we used the default initial heap size (= 24K words) for both the Sparc Station and the PIM/p KLIC system. In addition, we also measured the performance at 10M word heap settings in the PIM/p KLIC system. This value is almost equivalent to the available heap size in the conventional PIM/p KL1 processing system.

3 Single Processor Performance

In this section, we compare the single processor performance of various systems, using a set of benchmark programs.

3.1 Benchmark Programs

The following benchmark programs have been used to improve the performance of the PIM/p system, and so, the execution mechanisms of the programs are already well understood. A brief outline is given here:

Naive reverse Reverses a given list by repeatedly calling `append`, thus the performance of this program reflects the performance of `append` itself. The number of total reductions in this program is linear with the square of the size of input list while the number of active cells is proportional to the size.

Life A variation of a life-game program. This program consists of a 38 by 38 torus network. Each node connects to its adjacent four nodes and obtains their states on every cycle. Then the node calculates its next state and sends it to the four neighbors. This tight communication between adjacent nodes prevents each node from continuing to run more than two reductions. The source program size is about 200 lines.

Puz15 A solver for a well known 15-puzzle. The IDA* algorithm is used to find the minimum turn sequence to the initial pattern. The source program size is about 1,000 lines.

Pento A solver for a well known pieces packing puzzle. The source program size is about 1,000 lines. This program does not run on the KLIC system, because of the semantic difference of vector and functor in KLIC.

Bp100x100 A solver for finding the best path of a two dimensional mesh network. The distance between two adjacent nodes are randomly generated. The source program size is about 1,500 lines. This program does not run on the KLIC system, because KLIC does not support strings with 16-bit elements.

Waltz The program is a revised version for KL1 by E. Tick. The source program size is about 200 lines.

Nqueen N-queen solver. It generates all the solutions of N-queen programs. To investigate different search trees, it reproduces the current environment by using `append`, diagonal check is done by arithmetic calculation. The source program size is about 35 lines.

Zebra A constraint solver written by E. Tick. The source program size is about 200 lines.

Puzzle A solver for a 3-Dimensional pieces packing puzzle, written by E.Tick. The source program size is about 160 lines.

3.2 Measurement results

Firstly we show the `append` performance of each system. Historically, append execution speed has been frequently used to demonstrate the system performance. We will look at its detailed execution in Section3.3.

Table 1 shows the `append` Reductions Per Second, or RPS, in seven configurations. To observe the influence of cache miss-hit, we use 1,500 element and 5,000 element lists.

The heap size setting of the KLIC system directly affects the performance, thus it is difficult to state the KLIC performance. Too small a heap size decreases the performance due to GC overheads, too large a heap size also decreases performance due to cache miss hit penalty. Furthermore, UNIX related overheads, such as for virtual memory management or task switching, make the result obscure. By using the KLIC system on PIM/p, we can be free from these factors. Therefore, the KLIC/p system

Table 1: Append RPS on various systems

System	clock (ns)	nrev1500 (RPS×10^3)	nrev5000 (RPS×10^3)	Ratio
PIM/m	65	600	411	.69
PIM/p	80	305	281	.92
PIM/c	66	55	56	1.02
PIM/i	240	65	65	1.00
PIM/k	100	76	76	1.00
KLIC(1)	28	1,151	1,173	1.01
KLIC/p	80	225	238	1.06
KLIC/p(2)	80	405	386	0.95

(1) On SS10, time is measured by CPU time.

(2) The initial heap size is 10 mega word.

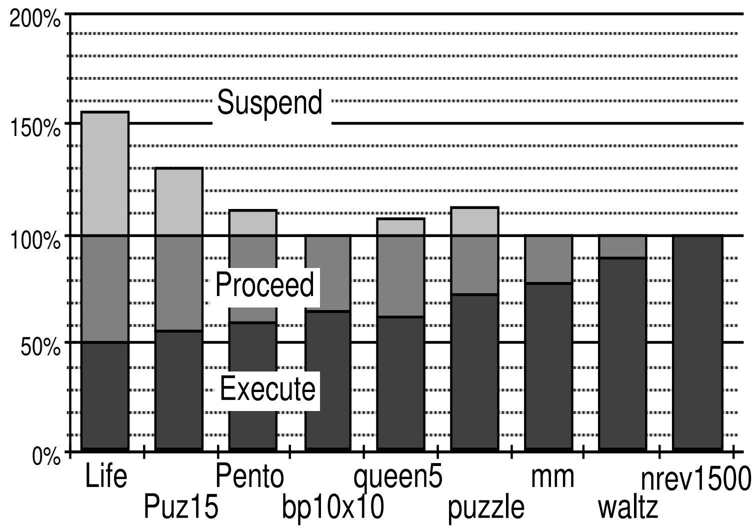

Figure 1: Dynamic Behavior of the Benchmark Programs

is better for observing the influence of heap settings on performance. We use a default heap size setting for the KLIC/p system, and also we use the size setting equal to the heap size of PIM/p system.

The distinct performance difference between short and long list reverse exists on the PIM/m system. The cache size of PIM/m is 4K words, it cannot hold 5,000 element lists, resulting in performance degradation.

Using `append` as a standard measure, we can measure the system performance to some extent. In KL1 processing, the execution patterns found in `append` can be viewed as stream creation and stream consumption. Consequently `append` employs some features of the KL1 language. But the other benchmark programs, which have more complicated structures and are closer to actual application programs, show different aspects of the KL1 processing systems.

To show the difference between `append` and the other programs, we classified the execution into three types, *Execute, Proceed* and *Suspension*. The ratios of these types are shown in Figure 1. All processing systems we measured in this experiment use last goal optimization, and thus *Execute* is the lightest operation, and *Proceed* is the next. Both *Execute* and *Proceed* actions are executed at the end of reduction, the portion over the 100% represents the suspension rate relative to the reduction counts. If a predicate body has more than one goal, like `append`, all reductions are executed by *Execute*. In KL1, suspension and resumption realize concurrency, which is one of the main features of KL1. The append RPS measurement does not reflect this feature at all, because `append` execution does not cause any suspensions/resumptions.

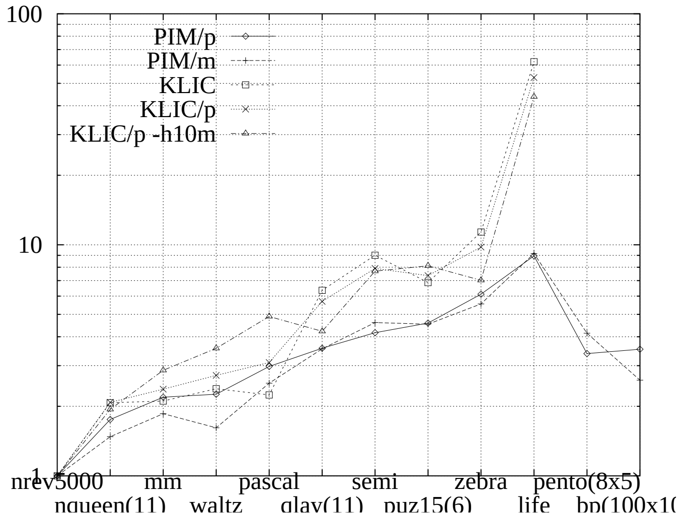

Figure 2: Normalized Reduction Time

In most of programs, one reduction requires a much longer time than `append`. Figure 2 shows the average reduction time of various programs normalized to the figures for `append`.

Table 3 shows the performance comparison of PIM/p and KLIC/p with a 10M word heap for various programs. We can see that KLIC has almost the same performance to the native KL1 processing system, except for the life program. We discuss the life program in Section 3.4.

3.3 Append Code Analysis

In this section, we study the key technique of the KLIC system to obtain its performance without any hardware support for the tag operations. To make the discussion more concrete, we use `append` execution trace on both PIM/p and KLIC/p.

Figure 3 and 4 show the pseudo C-code based on `append` execution trace on PIM/p and KLIC/p, respectively. In these figures, each line corresponds to a machine instruction. In the KLIC/p code, a total of 21 instructions are executed in the main cycle of the `append`. The number is same as that for a Sparc Station or a DEC Alpha, while the PIM/p KL1 system executes 28 instructions. To investigate the reason, we classify those instructions by function.

Table 2 shows the results. From these figures, we found:

- Over 20% of the total instructions are MRB related instructions.
- The locking/unlocking instructions, which are

```
append(x, y, z)
{
 top:
  if (is_ref(x)) {                        typechk
      t = x; x = deref(x);                deref
      if (is_undef(x)) goto susp;         typechk
      if (!is_mrb_on(x)) {                MRB(gc)
          *t = flist; flist = t;}         MRB(gc)
      if (is_ref(x)) goto ...; }          typechk
  if (!is_list(x)) goto ...;              typechk
  d = cdr(x);                             read(cdr)
  if (is_mrb_on(x)) new_list(x);          MRB(gc)
  if (empty(flist)) make_flist();         alloc
  new_w = flist; flist = *new_w;          alloc
  cdr(x) = new_w;                         write(cdr)
  *new_w = UNBOUND;                       MRB(penalty)
  {   /* mcall unify_bound(z, x) */     (mcall-penalty)
    if (is_ref(z)) {                      typechk
        work1 = z; z = deref(z);          deref
        if (!is_undef(z)) goto ...;}      typechk
    work2 = lock_read(work1);             exclusive
    if (is_mrb_on(work2)) {...};          MRB
    if (!is_undef(work2)) goto ...;       exclusive
    z = x;                                redundant
    unlock(*work1) = x;}                  write(unify)
  x = d;                                  MRB(penalty)
  z = new_w;                              move
  reds =                                  shoen
    reds -                                shoen
      2;                                  shoen
  if (reds > 0 && !slit_chk()) goto top; slitchk&loop
}
```

Figure 3: Pseudo C-code of `append` on PIM/p

```
append(x, y, z)
{
 top:
  tag = tagof(x);             typechk
  if (is_list_tag(tag))       typechk
    goto list;                typechk
                   delay_slot(redundant)
list:new_word = allocp;       alloc
  *allocp = allocp;           write(cdr)
  work = car(x);              read(car)
  *(allocp+1) = work;         write(car)
  new_cons = mkcons(allocp)   addtag
  tag = tagof(z)              typechk
  allocp += 2;                alloc
  if (!is_ref(tag)) goto ..;  typechk
  work = *z;                  deref
  if (z == work)              typechk
    goto L;                   typechk
L:*z = new_cons;              write(unify)
  x = cdr(x);                 read(cdr)
  z = new_word;               move
  work = heaplimit;           slitchk
  if (allocp < work)          slitchk
    goto top;                 slitchk
}
```

Figure 4: Pseudo C-code of `append` on KLIC/p

Table 2: Classification of Executed Instructions

Category	PIM/p		KLIC/p
Type-check	6	$<$	7
MRB	6	$\gg$	–
Write	2	$<$	3
SHOEN	3	$>$	–
Exclusive	2	$>$	–
Deref	2	$=$	2
Allocation	2	$=$	2
Read	1	$=$	1
Slit-check	1	$<$	3
Move	1	$>$	1
Other	2	$>$	–
Total	28	$>$	21

needed to support VPIM parallel execution in a cluster, form a rather small portion.

- The SHOEN management occupies a small but non-negligible portion.

- PIM/p checks the data tags 6 times with 6 instructions. However, KLIC/p checks only 3 times with 7 instructions.

The MRB GC[2] can prevent greedy memory consumption, however, it must pay the following penalties:

Dereference and Type-checking The PIM/p system needs only one instruction to check a tag, while the KLIC/p system requires two or more instructions. However, the total number of tag checking instructions is nearly the same as on the PIM/p system. That is because the variable cells must be kept outside the structure to enable the structure to be collected on the fly. In execution of `append` on the PIM/p, most of the cons structures have an undefined variable in the cdr part. Hence, one more dereference and another type checking are needed.

Extra Move Instruction In the `append` code, a cons structure is reclaimed prior to the use of its contents by the MRB GC. This requires the additional move instructions to backup the contents into a register.

Free List Management The collected cells are managed in a free list, and cells are always allocated from the list. Therefore an empty list checking instruction is demanded before every memory allocation.

GC Cost The cost of MRB GC is almost linear with heap consumption, and roughly linear with the number of reductions. However, the copying GC cost is linear with the number of active cells. The MRB GC performs better when the heap is almost full with active cells. Obviously, our benchmark programs do not meet this criteria, and so the MRB GC is not suitable.

From above observation, we conclude that the memory management scheme of KLIC is well suited for our benchmark programs.

3.4 Life Program Execution Analysis

The life program shows an interesting contrast with the append program. On the KLIC system, most programs run almost the same as, sometimes faster than, on the PIM system. The life program is an exception. The benchmark result shows that the KLIC/p system runs four times slower than the PIM system, and if the default heap size is used, the performance is down to 1/7 of the PIM performance. Even if we use KLIC on a Sparc Station, the performance is still inferior to the PIM/p KL1 system. This performance deterioration is explained as follows:

- Strong dependency among goals results in the very frequent suspensions. In the KLIC system, the goal records and suspension structure are allocated to the heap, and only a garbage collector can reclaim it, while the PIM systems reclaim goal records when it is dequeued from the goal queue.

  Usually, these structures are several times larger than a cons cell used for a stream. Frequent suspension accelerates heap consumption, and causes continual invocations of the garbage collector which severely reduces the performance. By looking into the performance of KLIC/p with default and 10 mega word heap size, we conclude that a small heap size may cut down performance to about half.

- KLIC expands frequently-executed code inline to obtain its performance while the less frequently executed code is stuffed into library routines. To make code small and simple, the body of the KLIC code has only one exception handler.

  Therefore, when an exception occurs such as a garbage collection request or a goal suspension, the library routine must analyze the reason for the exception, and then the corresponding processing begins. Usually, this makes the code smaller and execution faster, but these techniques make suspension much slower.

4 Cluster Performance

In this section, we argue the parallel execution performance of the cluster-configured PIM. Benchmarking of parallel processing on shared memory configuration or distributed memory configuration is very difficult. One reason is that the execution mechanism of the parallel environment differs for each system. Therefore, simple measurement of execution time may mislead the arguments.

Table 3: Relative Machine Performance

System	nrev5000	life	mm	pascal	semi
PIM/p	*44452*	*11458*	*13517*	*3546*	*10832*
PIM/p	1	1	1	1	1
PIM/m	1.46	1.43	1.72	1.73	1.32
PIM/c	0.20	0.07	0.05	0.07	0.07
PIM/k	0.27	0.39	0.38	0.38	0.34
PIM/i	0.23	0.30	0.26	0.29	0.16
KLIC(1)	4.17	0.60	4.33	5.54	1.93
KLIC/p	0.85	0.14	0.78	0.82	0.45
KLIC/p(2)	1.37	0.28	1.05	0.83	0.74
System	**nqueen(11)**	**qlay(11)**	**waltz**	**zebra**	**puz15(6)**
PIM/p	*28486*	*22144*	*9667*	*8782*	*131905*
PIM/p	1	1	1	1	1
PIM/m	1.74	1.47	2.04	1.61	1.48
PIM/c	0.07	0.12	0.05	0.05	0.05
PIM/k	0.36	0.36	0.36	0.39	0.45
PIM/i	0.27	0.26	0.31	0.12	0.13
KLIC(1)	3.53	2.34	3.95	2.25	2.78
KLIC/p	0.72	0.53	0.70	0.53	0.53
KLIC/p(2)	1.24	1.16	0.87	1.20	0.78

Italicized number shows absolute execution time on PIM/p in milli-second.

(1) On SS10, time is measured by CPU time.

(2) The initial heap size is 10 mega words.

To avoid the pitfall, first we try to build an execution model for a program, and then measure the parallel speedup of automatic load distribution on the PIM system.

4.1 Benchmark Program for Parallel Execution in Cluster

Good benchmark programs should cover a wide variety of application program parallelism. Parallelism is classified into three types:

OR parallelism: Processes consist of a process tree, and each process can be executed independently from the other processes. A program with this parallelism can easily obtain good parallel speedup even on loosely coupled parallel processors. The N-queen problem is a classic example.

Stream parallelism: Data generator processes and consumer processes are connected by streams, this parallelism is called as pipeline-parallelism also. The consumer is waiting until data is sent by a generator. It is generally difficult to obtain good performance on this type of parallel execution. Some resource control may be needed to prevent a generator runaway or a consumer data starvation. A typical example is a `sieve` prime number generator.

Parallel message passing parallelism: This type resembles stream parallelism, but dependency between generators and consumers may be bidirectional, resulting in efficient parallel execution more difficult. The `life` program is a simple example.

```
go(N):- N>0, N1:=N-1 | go(N1), go(N1).
go(N):- N=:=0        | true.
```

Figure 5: Simple Or-Parallel Program TP1

Unfortunately, making a general model available for all these types is a difficult task. Currently we have made a model for a simple and very small program. In the following discussion, we use the very small OR-parallel program TP1 shown in Figure 5. In the TP1 program, each goal spawns two child goals until the depth reaches N. This node hierarchy constitutes a binary tree in which a total of 2^{N+1} goals exist. In each branch, a goal can execute independently without any communication to goals in other branches.

First we make execution models of the program for PIM/k, PIM/c and PIM/p. Then, we show the TP1 benchmark results, and finally we estimate the distribution overhead of the PIM systems from the results.

4.2 Load Distribution Models

To discuss the parallel performance precisely, it is important to make an execution model of the program for a specific execution environment.

The processing system of cluster-structured PIMs derives from the same code, the VPIM code. However, performance tuning on each PIM system makes the system slightly different. Because of this modification to the execution mechanism, models must be built independently.

The load balancing method of the VPIM based processor is as follows:

Original VPIM Load Distribution In the PIM/c and the PIM/k implementation, each processor has one goal queue which is accessed only by the processor. Therefore, no exclusive operation is needed to manipulate it. The load distribution works as follows: When a processor becomes idle, it signals to the busy processors to send a goal. Then, a busy processor sends the goal next to the currently executing goal. Thus, when no idle processors exist in the cluster, no special operation for goal distribution is needed, resulting in a low overhead in single processor execution.

Modified VPIM Load Distribution In the PIM/p implementation, each processor has two queues; a local queue and a global queue. Enqueuing and dequeuing goals are assigned only to the local queue, which can contain up to two goals. Surplus goals are swapped out to the global queue, which requires expensive lock/unlock operation to protect from unexpected operations by the other processors. The local queue is prepared to reduce the overhead in accessing the global goal queue. When a processor becomes idle, it searches other global queues, and if it finds a non-empty queue, it takes a goal for itself.

Now we can consider how the above difference affects the parallel performance of TP1. We look at the two processor case to make the problem simple, although it is feasible to extend this argument to more than two processors.

In the following discussion, we assume the following hypothesis:

1. All goal reductions take the same time ($= T$).
2. The busy processor gives to the idle processor the next goal to execute in the goal queue which is managed as a LIFO[4]. The goal sent requires j reduction to reduce. Stochastic analysis shows that j is equal to the depth of the initial tree N.
3. The initial goal requires 2^{N+1} reductions to reduce. In goal transferring, average $j(= N)$ reductions task is moved from the busy processor to the idle processor, and both sender/receiver processors must execute transferring overhead o.[5]
4. During execution, both processors must always execute reductions or send/receive tasks. Thus, the sum of the reduction-time and the distribution-time is the same on both processors.

At first we analyze the PIM/c and the PIM/k implementation.

Execution Model for PIM/c and PIM/k The load distribution overhead must be paid by both sending and receiving processors.

Initially, the busy processor has a total of 2^{N+1} tasks. When a goal is transferred, $N \cdot T$ tasks are moved from the busy processor to the idle processor, and overhead o is added to the both processors. If task transfer occurs a total of d times, the task to be executed on both processor becomes:

$$\begin{array}{lrr} \texttt{Busy processor executes}: & 2^{N+1}T+ & d \cdot o - d \cdot N \cdot T \\ \texttt{Idle processor executes}: & & d \cdot o + d \cdot N \cdot T \end{array}$$

From assumption (4), these execution times must be the same, then we can solve for the number of distributions d and also obtain the theoretical speedup ratio as:

$$\text{speedup} = \frac{2}{1 + o/(N \cdot T)}$$

if $o > N \cdot T$, i.e. the goal transferring overhead is larger than the average task contained in the transferred goal, the *speedup* becomes less than 1.

[4]We should call it a goal stack instead of a goal queue, but historically we call it a goal queue.

[5]In the PIM/p system, the goal sending overhead is negligible for the sender.

Table 4: Load Distribution Costs in Cluster

	Distribution Costs(μs)	Equivalent Reductions
PIM/p	17∼22	3∼4
PIM/c	239∼275	14∼16
PIM/k	94∼116	9∼11

Execution Model for PIM/p In the PIM/p system, the goal distribution overhead for busy processors is very small. Thus, we can assume that the goal sending overhead is added to the idle processor. Using a similar calculation, we obtain the theoretical speedup ratio for PIM/p as:

$$\text{speedup} = 1 + \frac{1}{1 + o/(N \cdot T)}$$

The *speedup* is always greater than 1 regardless of the transferring overhead.

From the above two equations, we can say that if the distribution overhead becomes relatively large, the original VPIM method has inferior parallel performance to a single processor performance, while the modified VPIM method does not.

On the other hand, if the overhead is small enough, the original method will perform better because no preparation for parallel execution, such as global queuing with exclusive operations, is needed.

4.3 Cluster Performance Measurement

We measure the execution time of `TP1` with $N = 20$ on the three types of PIM. The result is shown in Figure 6. The speedup is normalized to the speed of one processor.

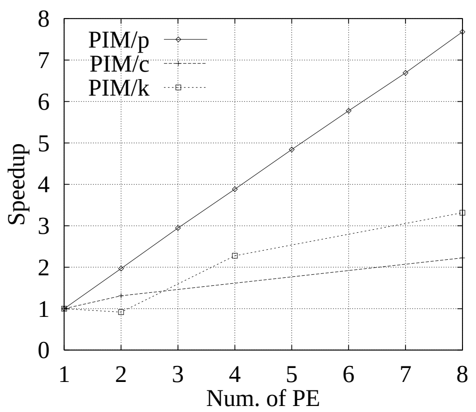

Figure 6: Parallel Speedup Ratio

Table 4 shows the estimated overhead time o in the above theoretical arguments calculated from the benchmark results. *Equivalent reduction* means the goal transfer overhead measured in units of reduction time. The PIM/p executes less expensive goal distributions for both absolute timing and relative timing. The reason is:

- The VPIM method uses interruptions to start task distribution, then the interrupted busy processor must save the current status into a goal record and then enqueue it. Usually, the interrupt handling takes several times longer than a reduction of `TP1`, resulting in large transfer overhead.
- The idle processor does not know which processor currently has surplus goals. Therefore, it interrupts all processors. However, to avoid a concentration of goals, only one processor is allowed to send a goal to the idle processor. Thus, unnecessary interrupt handling on other processor becomes a large distribution overhead.

If automatic distribution does not select appropriate goals to distribute, like the current PIM system, making the distribution overhead less expensive is the first thing, even though that may decrease single processor performance.

5 Distributed System Performance

The KL1 execution in a distributed parallel processor environment differs from the execution in a cluster, because goal destination in a program must be explicitly specified with *throw goal pragma*.

The major factors of the distributed system performance are:

- The time needed to throw goal
- The time to send/receive data
- The overhead for managing meta-program facilities such as a program termination detection

In a cluster, sending/receiving data is realized by passing the address pointer to the data between processors. On the other hand, in a distributed memory environment, the data to be sent is registered in an *export table* prior to transmission, and then the index of the table is sent. The receiver reads the number, and register it in an *import table* along with the sender processor ID. This translation enables independent garbage collection in clusters[4]. Usually, goal throwing is less frequent than data transfer, so operations on import/export tables become main overhead of inter-cluster communication.

In this section, we measure and compare the communication speed inside a cluster and between clusters.

5.1 Benchmark Programs

The time for the data transmission depends on the data structure to be transferred, as large data or complicated data needs more time.

In the processing of communication between clusters, a nested data structure is transferred lazily. When a receiver side tries to read imported data, a read-request is sent to the processor which exports the data, and then only one level of the data structure will be sent. Therefore, to measure the

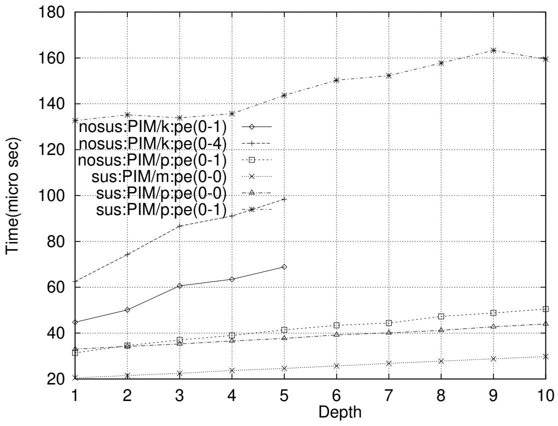

Figure 7: Inner Cluster Round-Trip Time vs. Data Depth

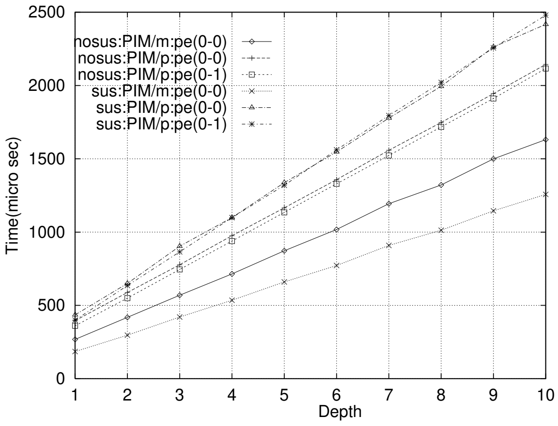

Figure 8: Inter Cluster Round-Trip Time vs. Data Depth

communication time, we have to measure the communication time dependence on the data structure. In this experiment, we measure the dependence on the data width and the data depth. For example, to measure the communication time for width five and depth five, we use `{1, 2, 3, 4, Next}` and `{{{{{Next}}}}}` , respectively. Here, `Next` is a variable to represent the next data to be sent. We measured the communication time for the data of width one to ten, and depth one to ten levels.

In the benchmark programs, we place two processes on specific processors, one is a sender and the other is a receiver. The sender process sends the data we have mentioned above and waits an acknowledgment from the receiver. The receiver process reads the data sent and sends an acknowledgment to the sender. The acknowledgment is always a one word vector.

5.2 Measurement Results on PIM

The partial results of the experiments are shown in Figure 7 which represents round-trip time of the specific depth in a cluster. Figure 8 shows that of the inter-cluster. The measurement results show that the execution time of the benchmark programs is almost linear with depth or length of the transferred data.

Thus, we can fit regression lines to the results. In a cluster of PIM/p, the round-trip time of a width one (and depth one) data takes 20.5μsec, it increases by 0.9μs or 1.0μs with the increase of the data width or depth, respectively. In inter-cluster communication, the round-trip time of a width one (and depth one) data takes about 182μsec, it increases by 8μs or 120μs with the increase of the data width or depth, respectively.

From these experiments, we can conclude the followings:

Inner/Inter-Cluster Communication The data transferring time depends on the data structure. In the case of small data transfer, the inter-cluster communication takes about 10 times longer than the communication within a cluster. If the depth of the transferred data is ten, inter-cluster communication takes about 100 times longer than within a cluster.

Influence of Data Structure All PIM systems except PIM/k take longer time to transfer deep structured data than to transfer a wide structured data, because all PIM models use the one-level data transfer scheme. In the PIM/k system, all processors are connected with common bus, the data transfer between mini-clusters is same as that in a mini-cluster. However, as all inter mini-cluster communication use the second-level common bus, it may become a bottleneck, which must be examined by other measurements.

5.3 Measurement Results on KLIC

Currently, an experimental version of KLIC parallel implementation on distributed memory machines is available[9]. We measure the same benchmark program on the KLIC-1.511 system for PVM, running on a SparcCenter. The experiment shows curious results. The round-trip time of depth N data requires $(20.1N + 9.8)$ milli-seconds. When we run the same program on Sparc Station 10 using four processes[6], The round-trip time of depth N data needs $(20.0N + 1.7)$ milli-seconds. Both experiments show nearly the same coefficients. To see bare PVM performance, we measured the performance using a C program, and we found that the delay of transmission is caused by PVM itself. The processing performance of KLIC system on SparcCenter is two to five times faster than that of PIM/m or PIM/p, however the communication performance of KLIC is almost one hundredth of the PIMs. PVM is not indented for use by response-time dependent program such as the distributed KLIC system. Therefore, the KLIC system needs other choices for the communication path without loss of portability.

We conclude that, in the communication dependent KL1 program, the PIM has superior performance to the current distributed KLIC system. Even if we use other communication libraries, we may not attain the same communication performance compared to the PIM networks. Therefore, the KLIC system needs a granularity control method to make the communication course grain and less dependent on the network response time.

[6]The distributed KLIC system needs $N + 2$ processes, while N is a number of reduction processes. In this case two processes are used for reduction, one process for an I/O and another for a termination detection.

6 Conclusion

In this report, we measured the performance of all PIM systems and the KLIC system, and analyze the performance quantitatively approach. In Section 3, single processor performance of the KLIC system is nearly equal to, sometimes superior to, that of dedicated machines when well behaved programs are executed.

The fact shows that the complicated operations of MRB suppress the PIM performance in these cases. However, when the active data fillup most of heap, the MRB scheme will have the advantage to a stop-and-collect GC.

And also, the goal reclamation scheme of PIM shows its advantage in programs with frequent suspensions. Therefore, the KLIC system may still have some room to improve the performance of such programs.

In Section 4, we measured the automatic load-balancing mechanism of the cluster-configured PIMs. We made execution model and analyzed the benchmarking results using it. From the execution model, we proved the modified VPIM load distribution method is superior to the original VPIM method. The analysis also shows that there should be some goal distribution strategies to achieve efficient load distribution.

We measured the distributed system performance in Section 5. From the measurement, the PIM system has superior performance to the current experimental KLIC system for communication dependent programs. The current distributed KLIC system performs communication several hundred times slower than the PIM systems. We think the KLIC system will be improved to some extent, however, it is difficult to fill up the gap between processing speed and the communication speed. Therefore, the KLIC system needs a granularity control method to make the communication course grain and less dependent on the network response time.

Through the measurement, we confirm that a bare hardware and a good software, environment, such as PIM/p, help us evaluate the architecture and the software implementation.

Acknowledgment

The research on evaluation of the PIM systems was carried out by the PIM/p group in the First Research Department at ICOT. The evaluation of PIM other than PIM/p was carried out by Katsumi Takahashi of Mitsubishi Electoric Corp., Toshiaki Tarui of Hitachi Ltd., Hiroshi Sakai of Toshiba Corp., Kenji Kato and all people who have engaged in the development and the measurement of the PIM systems.

The authors express special thanks to the Free Software Foundation for developing and distributing the GNU C-compiler, which enables the comparison between KLIC and the KL1 system on the PIM/p hardware.

Finally, the authors would like to express their gratitude to the PIME-TG members, who gave us helpful and fruitful comments.

References

[1] Takashi Chikayama, Teturo Fujise, and Daigo Sekita. A Portable and Efficient Implementation of KL1. In Manuel Hermenegildo and Jean Penjam, editors, *Proceedings of International Symposium on Programming Language Implementation and Logic Programming*, number 884 in Lecture Notes inComputer Science. Springer-Verlag, 1994.

[2] Takashi Chikayama and Yasunori Kimura. Multiple Reference Management in Flat GHC. In *Proc. of Fourth International Conference on Logic Programming*, pages 276–293. ICOT, 1988.

[3] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. A Portable Implementation of KL1. In *FGCS94*. ICOT, 1994.

[4] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A new external reference management and distributed unification for kl1. *New Generation Computing*, 7(2–3):159–177, 1990.

[5] ICOT 1st Laboratory. Tutorial on VPIM Implementaion(In Japanese). Technical Memorandom 1044, ICOT, 1991.

[6] K. Kumon et al. Architecture and Implementation of PIM/p. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 414–424. ICOT, June 1992.

[7] T. Nakagawa et al. Hardware implementation of dynamic load balancing in the parallel inference machine pim/c. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 723–730. ICOT, June 1992.

[8] H. Nakashima et al. Architecture and Implementation of PIM/m. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 425–435. ICOT, June 1992.

[9] K. Rokusawa, A. Nakase, T. Chikayama, T. Fujise, et al. Distributed Memory Implementation of KLIC. In *Workshop on Parallel Logic Programming and its Programming Environments*, number CIS-TR-94-04, pages 151–162. University of Oregon, 1994.

[10] Kazuo Taki. Parallel Inference Machine PIM. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 50–72. ICOT, June 1992.