PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A LOAD BALANCING MECHANISM FOR LARGE SCALE MULTIPROCESSOR SYSTEMS AND ITS IMPLEMENTATION

Yasutaka Takeda† Hiroshi Nakashima† Kanae Masuda†
Takashi Chikayama‡ Kazuo Taki‡

† Mitsubishi Electric Corporation
5-1-1,Ofuna,Kamakura 247,Japan

‡ ICOT Research Center
1-4-28,Mita,Minato-ku,Tokyo 108,Japan

ABSTRACT

In large scale multi-processor systems, the distance between processors should be taken into account by software to reduce the network traffic and the communication overhead. A load balancing method based on P^3 (Processing Power Plane) model is proposed to enable programmers to specify distributing computational load, keeping the locality of the computation. In this method, a process is allocated to a rectangle on a hypothetical processing power plane. The size of the rectangle represents the processing power given to the process, and the distance between rectangles represents the communication cost between them. This plane is divided to processors, and the region of the processor may be dynamically reshaped to alleviate imbalance on P^3. Mechanism for realization of the method has been implemented on the Multi-PSI/version 2, which is a parallel processing system with 64 processing elements connected to form a 2-dimensional mesh network. A packet transmission mechanism of the Multi-PSI/version 2 is described, which realizes the process distribution along with the balancing method.

1 INTRODUCTION

One of the most important research themes of the Japanese Fifth Generation Computer project is to build a parallel computer system PIM (Parallel Inference Machine) [Chikayama 1987, Sato et al. 1987] for high performance knowledge information processing. The concurrent logic programming language KL1 (Kernel Language Version 1), based on flat-GHC [Ueda 1985], should be implemented on the PIM [Ichiyoshi et al. 1987, Kimura et al. 1987]. The operating system PIMOS (Parallel Inference Machine Operating System), written in KL1, should be also developed on it [Chikayama 1987].

In large scale parallel machines, such as PIM, it is impossible to connect all the processors in constant overhead. Otherwise, any two processors must be connected with the cost as high as connecting the most distant two processors. On such machines, the software must be aware of the communication locality to achieve efficient computation.

On the other hand, load balancing is one of the most important and difficult problem to efficiently utilize the full processing power of large scale systems. In most numerical programs of high regularity, computational load of each process may be estimated beforehand and the load distribution may be given precisely in the program. However, in execution of more complicated programs with irregularity, such as knowledge processing systems, computational load is determined only at run time depending on intermediate results of computation.

Dyamic load balancing can be achieved by redistributing computation uniformly to the whole system. If this redistribution should not take communication locality into account, communication cost may increase significantly, and, in some cactses, no dynamic balancing ever may bring better result.

To solve this problem, a load balancing method is introduced, which utilizes the locality information specified by the software. In this method, the software specifies locality of computation on a hypothetical plane where computational power is uniformly distributed (called Processing Power Plane, or P^3). Dynamic load balancing is effected by changing how this plane is subdivided by physical processors.

The Multi-PSI/version 2, a loosely coupled multiprocessor system, has been developed as a prototype for establishing parallel software research and development environment [Taki 1986]. The Multi-PSI/version 2 is also used as an architectural evaluation model of the PIM. Mechanism for the load balancing method based on P^3 has been implemented on the Multi-PSI/version 2.

This paper is organized as follows: Section 2 describes P^3 model and the load balancing method; Sections 3 describes the implementation of the load balancing method for the Multi-PSI/version 2; Section 4 gives the conclusion and future works.

2 LOAD BALANCING METHOD

2.1 Model of the Processing Hardware

To consider communication locality, some model

of processing hardware with the notion of communication cost is reguired. It is not desirable, however, that the software should be aware of the physical stucture of the multi-processor system, such as the number of processors and inter-processor boundary. Therefore, an abstract model, which has some notion of countiguous distance, is required.

This notion of *distance* must have some correspondence with the physical distance in the hardware system. This leads to a quite naive model which almost directly corresponds to the physical structure of the system —to consider the system to be a cube in an n-dimensional Euclid space in which computing power is uniformly distributed. In multi-PSI/version 2, this n is 2 for arbitrarily extensible implementation, and this 2-dimensional cube (or plane) is called the *Processing Power Plane*, or $\mathbf{P^3}$, in short.

Any computation is located at some point in $\mathbf{P^3}$. The distance of two computations is modeled by the distance of two such points. This also is a linear approximation of the physical communication overhead.

2.2 Keeping Locality

To keep the locality of communication, the software must be aware of the distance between two computations. This might be realized by the following way.

- Programs should be organized so that processes requiring more communication fork later. That is, in the process tree structure, processes with more communication have their common ancestor process in lower levels.
- The initial process is given the whole $\mathbf{P^3}$ for its use.
- Each process is given some sub-rectangle of the rectangle given to its parent.

Using the above allocation mechanism, two processes are always allocated inside the rectangle area allotted to their latest common ancestor, which gives a certain lower bound of locality.

Another probably feasible method of keeping locality is to allocate processes to where the required data are. When, for example, the generator consists of many processes scattered all around $\mathbf{P^3}$, and each such leaf process generates some substructure of the generated data, then the generated data (probably with some tree-like structure, corresponding to the generator processes' tree structure) will also be scattered all around $\mathbf{P^3}$. In such cases, the consumer processes should be allocated using the same strategy as the generator processes, so that they are allocated at the same point on $\mathbf{P^3}$ with the data generated by the corresponding generator process.

2.3 Load Balancing on $\mathbf{P^3}$

For the *logical* balancing of the computational load, the programmer and/or the language processing system are responsible. To achieve this, the programmer and/or the language processor should have at least vague knowledge of how much computation (relatively) is needed for a certain process.

One notation proposed for Prolog-like languages is something like:

$$p :- \leftarrow(2 \times q),\ \rightarrow r.$$

By the above specification, the subplane given to the predicate p will be subdivided for q and r as shown in Figure 1b. This specifies that the subgoal q should be allotted twice as large subplane as the subgoal r.

Arrows before the body goals specify which way the subdivision should be when body goals are subdivided again by their reduction in turn. By reductions using clauses such as:

$$q :- \leftarrow(3 \times s),\ \rightarrow t.$$

and

$$r :- \leftarrow u,\ \rightarrow v.$$

rectangles for q and r are subdivided again as shown in Figure 1c.

It would be sometimes impossible to specify such load balancing information statically on the source code. It might be possible, however, in certain kinds of programs, to guess the amount of computation required at runtime from the given data (sizes of argument structures, for example). In such cases, the subdivision parameters can be computed depending on arguments.

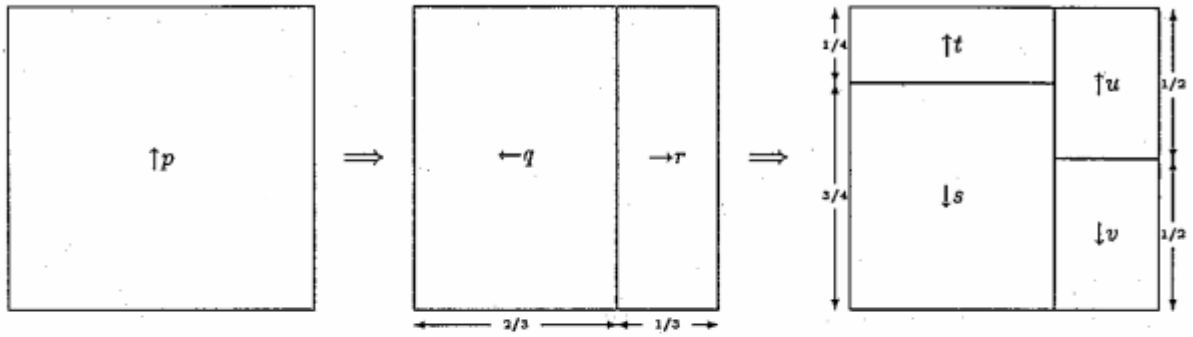
2.4 Load Balancing on the Physical Level

$\mathbf{P^3}$ must somehow be covered by the physical processors. When the whole computation starts, all the processors in the system should be responsible for the same amount of $\mathbf{P^3}$ as in Figure 2a. However, as perfect load balancing on $\mathbf{P^3}$ for complicated algorithms seems to be impossible, it is quite likely that some area of $\mathbf{P^3}$ becomes quite dense in computation and some area quite sparse (Figure 2b).

This could be improved (at least partially) by changing the size of the region in $\mathbf{P^3}$ each processor is responsible for. Processors responsible for dense area should narrower their territories and those for sparse areas should widen them (Figure 2c).

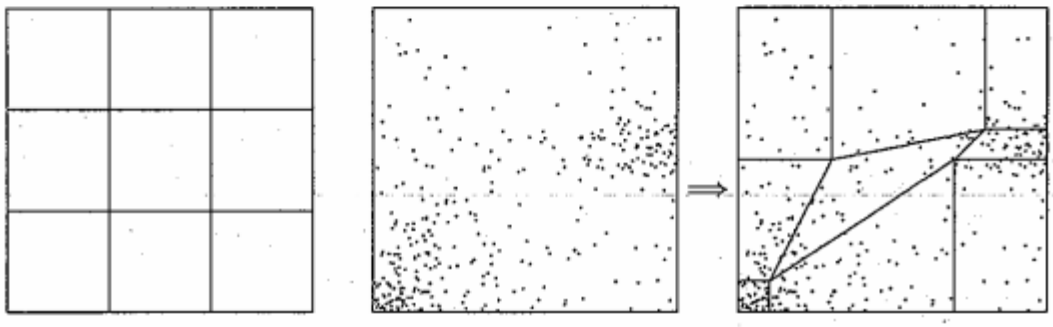
Assume that a process is allocated quite close to some processor boundary and communicating frequently with another process allocated on the opposite side of the boundary. Although these two processes are almost adjacent on $\mathbf{P^3}$, communication cost with such a process will be much higher than the cost with other processes, not as close on $\mathbf{P^3}$ but are allocated incidentally on the same processor. Such processes can be the bottleneck of the whole computation. To avoid such cases, some randomness might be desired in mapping $\mathbf{P^3}$ to physical processors.

Locality is required also in the reallocation. If some centralized controller should be required for such



a. Initial state b. After reducing p c. After reducing q and r

Figure 1: Load Balancing by Subdividing P^3



a. Initial allocation b. Load distribution c. Reallocation

Figure 2: Load Balancing by Reallocation of Processors to P^3

an adjustment, much communication might be required for exchanging load balancing information. Thus, reallocation should be decided locally. A simplest way is by relocating the corner point shared by four adjacent processors, depending on the *loads* of the four processors. This method requires quite local communication only. Computation will be distributed to other processors in a diffusion manner.

2.5 Requirements for Algorithm Design

It is widely understood that algorithms good for sequential processors may not be as good for parallel processors. With the above processor model, what is important is not only the intrinsic parallelism, but also whether a certain algorithm has good nature in the following viewpoints.

Communication Locality: An algorithm is better when processes require less global communication. Some measurement resembling the notion of working set in sequential programming is required as an efficiency criterion.

Feasibility of Load Balancing: An algorithm is better when it is easier to predict required amount of computation for each subproblem. Some new measurement is required here again.

3 IMPLEMENTATION FOR THE Multi-PSI/version 2

This section describes an implementation of the load balancing method based on P^3 model for the Multi-PSI/version 2. In the load balancing method, throwing a goal is sending a packet to the processor element (PE) responsible for the point on P^3 where the goal is allocated. If the goal thrower PE knows the complete mapping from P^3 to PEs, it is quite easy to send the packet to the destination. This, however, requires global informatin of the mapping.

Therefore, we have developed the following;

(1) The packet transmission algorithm using only local information, that is, the region of each PE which relays the packet.

(2) The hardware mechanism to implement the algorithm using table look-up.

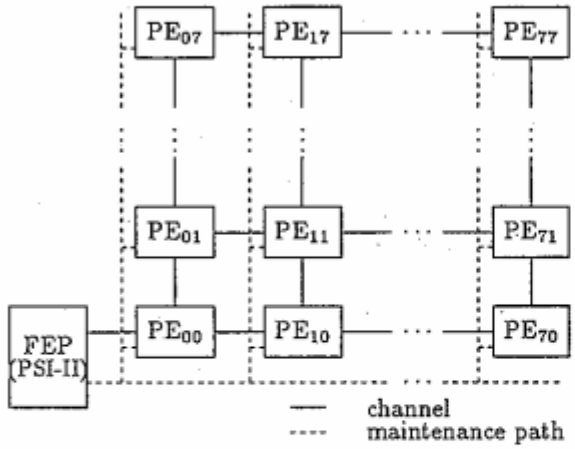


Figure 3: Multi-PSI/Version 2

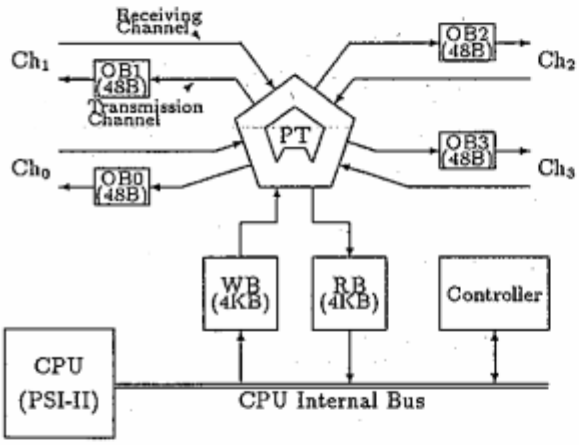


Figure 4: Connection Hardware

The following subsections describe the architecture of the Multi-PSI/version 2, and the algorithm and hardware mechanism for the goal throwing.

3.1 Multi-PSI/version 2 System

Figure 3 shows the block diagram of the Multi-PSI/version 2 system. Up to 64 processing elements (PE) are connected to form a 2-dimensional mesh inter-processor network. The network carries communication packets, such as the messages for the inter-processor unification and goal throwing. Each PE is a modified PSI-II CPU [7] with 16 M word local main memory. Its microprogrammed emulator of the KL1 machine instruction set achieves high performance of 200 KRPS (Reduction per Second).

Figure 4 shows the block diagram of the connection hardware for one node of the network. It has four bidirectional channels to connect adjacent four PEs and two buffers to connect the CPU. Packet data

```
s1      e = one_of(nearest-edges(r,p));
s2      r = adjacent_region(r,e,R);
s3      while not(include(p,r))
s4        { E = nearest_edges(r,p);
s5          if member(e,E) then;
s6            { el = left_edges(e,r);
s7              if member(el,E);
s8                  then e=el;
s9                  else e=right_edges(e,r)};
s10         else  e = one_of(E);
s11         r = adjacent_region(r,e,R)};
s12     exit;
```

Figure 5: Packet Transmission Algorithm

is transferred in 10-bit parallel including a parity bit. The transmission rate is 5 Mbytes/sec for each direction. Several such channel pairs can be formed simultaneously except a transmission channel is required from multiple receiving channels. Main components of the connection hardware are as follows:

PT (Path Table) Each receiving channel has its own PT to determine the channel to transmit a packet. Channel controller looks up this table using the destination address of the packet, and connects receiving and transmission channels according to the PT data. The detail of the PT is described in section 3.4.

OB 0-3 (Output Buffer 0-3) Each output channel has a 48-byte FIFO output buffer to reduce the possibility of network choking.

RB (CPU Read Buffer) Connection hardware stores an arriving packet in this buffer. The interrupt is raised to the CPU to notice the packet arrival when the packet data is completely stored.

WB (CPU Write Buffer) CPU writes a packet data into this buffer and the packet will be transmitted when the packet data is completely stored.

3.2 Algorithm of Packet Transmission

Figure 5 shows the algorithm to transmit a packet to its destination PE in general. In this figure, the variables represent:

- `p` : The destination point of the packet.
- `R` : The set of the all regions of $\mathbf{P}^3$, each of which corresponds to a PE. They may be arbitrary polygons. No two regions have common area and regions in `R` covers the whole $\mathbf{P}^3$.
- `r` : The regions corresponding to the PEs which relay the packet. Its initial value is the goal

thrower, and the final value is the destination PE.

`e` : The edge of the region corresponding to the channel through which the packet is transmitted.

And functions return:

`member(e,S)` :
True if `e` is an element of the set `S`.

`one_of(S)` :
Arbitrary element of the set `S`.

`include(p,r)` :
True if the point `p` is included in the region `r`.

`nearest_edges(r,p)` :
Set of the edges of the region `r`, which are nearest to the point `p`.

`adjacent_region(r,e,R)` :
The region in the set R, sharing the edge `e` of the region `r`.

`left_edge(e,r)` :
The left neighbor edge of the edge `e` of the region `r`.

`right_edge(e,r)` :
The right neighbor edge of the edge `e` of the region `r`.

According to the algorithm, the distance between the region `r` and the point `p` should not increase in the packet transmission, and the packet transmission path never loops. Therefore, if the number of regions is finite, the algorithm must terminate, that is, the packet must reach the destination PE. The proof of the termination is given in the appendix.

The functions `include(r,p)` and `nearest_edges (r,p)` can be computed as follows;

(1) `include(r,p)` :
Let $v_0, v_1, \ldots, v_n$ be the vertices of the region `r`, ordered counterclockwise. If the region `r` is a convex polygon, function `include(r,p)` is true when:

$$0 \leq^{\forall} i \leq n, \quad (v_{i+1} - v_i) \cdot (v_i - p) \geq 0$$

where $p \cdot q$ is the inner product of p and q. If the region is a concave polygon, it can be partitioned into convex polygons.

(2) `nearest_edges(r,p)` :
Let d_i be the distance from the edge $\overline{v_i v_{i+1}}$ to the point p, d_i is calculated as follows;

if $[(v_{i+1} - v_i) \times (p - v_{i+1})] \cdot [(v_{i+1} - v_i) \times (p - v_i)] \leq 0$

then $d_i{}^2 = min((p - v_{i+1})^2, (p - v_i)^2)$

else $d_i{}^2 = ((v_{i+1} - v_i) \times (p - v_i))^2 / (v_{i+1} - v_i)^2$

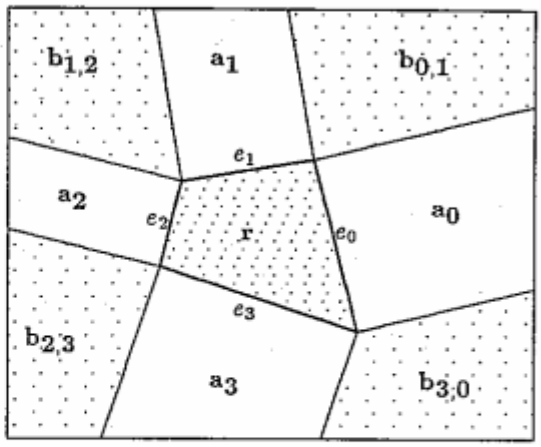


Figure 6: Geometric Method

where $p \times q$ is the outer product of p and q. The result of `nearest_edges(r,p)` is obtained through comparing distances of all the edges of the region `r`. The comparison of d_i and d_j is performed by calculating;

$$d_i{}^2 (v_{i+1} - v_i)^2 (v_{j+1} - v_j)^2 - d_j{}^2 (v_{j+1} - v_j)^2 (v_{i+1} - v_i)^2$$

Note that if the coordinates are represented in integer, these functions can be evaluated using only integer arithmetics.

3.3 Geometric Method of Packet Transmission

In the algorithm shown in 3.2, the region of each PE may be arbitrary polygon. In the actual implementation, however, the shape of the region is limited to a convex tetragon to avoid complicated calculation. Moreover, a geometric method greatly reduces the complexity to determine the transmission channel.

Figure 6 shows the geometric method to determine the transmission channel. In the figure, r is the region of a PE, and its edges e_0 to e_3 corresponds to the network channels. The outside area of the region r is subdivided into eight sub-areas a_0 to a_3 which share the edges with r, and $b_{0,1}$ to $b_{3,0}$ which shares vertices. The border of the sub-areas a_i and $b_{i,i\pm1}$ is the line perpendicular to the edge e_i, which is drawn from the vertex of the edge e_i and $e_{i\pm1}$. The packet transmission rule, shown in Table 1, is as follows;

rule 1: If the destination point of the packet is included in the region r, the packet has reached the destination.

rule 2: Otherwise, if the destination point is included in the sub-area a_i, the packet is transmitted to the channel corresponding to the edge e_i.

rule 3: Otherwise, if the destination point is included in the sub-area $b_{i,i+1}$, and neither of the edges e_i

Table 1: Packet Transmission Rule

receiving channel (edge)	sub-area								
	r	a_0	a_1	a_2	a_3	b_{01}	b_{12}	b_{23}	b_{30}
e_0	CPU	×	e_1	e_2	e_3	e_1	$e_{1/2}$	$e_{2/3}$	e_3
e_1	CPU	e_0	×	e_2	e_3	e_0	e_2	$e_{2/3}$	$e_{3/0}$
e_2	CPU	e_0	e_1	×	e_3	$e_{0/1}$	e_1	e_3	$e_{3/0}$
e_3	CPU	e_0	e_1	e_2	×	$e_{0/1}$	$e_{1/2}$	e_2	e_0

nor e_{i+1} corresponds to the received channl, the packet is transmitted to either of the channels corresponding to the edges e_i or e_{i+1}.

rule 4: Otherwise, the destination point is included in the sub-areas $b_{i,i+1}$, and either of the edge e_i or e_{i+1} corresponds to the received channel. If the edge e_i (or e_{i+1}) corresponds to the received channel, the packet is transmitted to the channel corresponding to the edge e_{i+1} (or e_i).

This rule is equivalent to the algorithm shown in Figure 5, because;

(1) **rule-1** is associated with the step `s3`.

(2) **rule-2** is associated with the step `s10`, because e_i is the unique element of the `nearest_edges(r,p)`, and it should not correspond to the received channel.

(3) **rule-3** is associated with the step `s10`.

(4) **rule-4** is associated with the steps `s8` and `s9`.

3.4 Hardware Support for the Packet Transmission

In the Multi-PSI/version 2 system, the geometric method to determine a packet transfer channel has been implemented using the Path Table (PT) lookup. The index of the PT is the coordinate of the destination point, which is represented by a pair of 7-bit integers. Each entry of the PT contains the channel number to which packets are transmitted, and is set up according to the method shown in 3.3 when the region is reshaped. Each receiving channel has its own PT for parallel switching of the packets. It also enables a packet to be transmitted to a different channel depending on the receiving channel.

The resolution of the coordinate for the software is not limited by the size of the PT. The hardware coordinate can be the top seven bits of the software coordinate which is represented by arbitrary precision integer (for example, a pair of 32-bit integers). A goal packet is sent to the PE responsible for the destination point represented by the hardware coordinate. Then, if the PE is not responsible for the point represented by the software coordinate, the packet is sent to the PE responsible for, by evaluating the functions shown in 3.2. In this case, the packet is not transmitted to the destination PE automatically. Each PE on the path evaluates the functions shown in 3.2, and transmits the packet to one of the adjacent PEs, using physical type packet (see below).

The reduction of the coordinate resolution, shown above, can be performed recursively. There are two-level hardware coordinates, called *fine* and *coarse*. The fine coordinate is represented by a pair of 7-bit integers, and the coarse coordinate is the top some bits of the fine. The fine coordinate is used for the area including the border of the responsible region, and the coarse coordinate for other areas. This mechanism reduces the time to update the contents of the PT when regions are reshaped, because the number of set up operation for the *coarse area* is greatly less than that for the *fine area*.

Figure 7 shows the mechanism of the PT lookup. A packet is a sequence of the 9-bit byte data, the top bit of which indicates the packet head or tail. The first 2 bytes of the packet represent the destination. The bit 7 of the second byte indicates the type of the destination as follows;

Physical type: If the bit is off, the lowest 7 bits of the first byte represent the destination PE number. This type is used for the messages except those of goal throwing, such as inter-processor unification messages.

Logical type: If the bit is on, the lowest 7 bits of the first and second bytes represent the x and y coordinates of the P^3. This type is used for the goal throwing messages.

The PT address for the physical type packets is generated by concatenating the PE number and the constant 127. Thus, the P^3 coordinate system for the hardware contains 128 × 127 points.

Table 2 shows the format of the each entry of the PT. The top bit indicates whether the coordinate is coarse or fine. The PT is accessed with some lowest bits of both x and y coordinates masked out. If the top bit of the result indicates a coarse coordinate, the packet is transmitted to the channel or stored in the CPU buffer according to the lowest 2 bits of the result. If the top bit indicates a fine coordinate, the PT is accessed again by all bits of x and y coordinates.

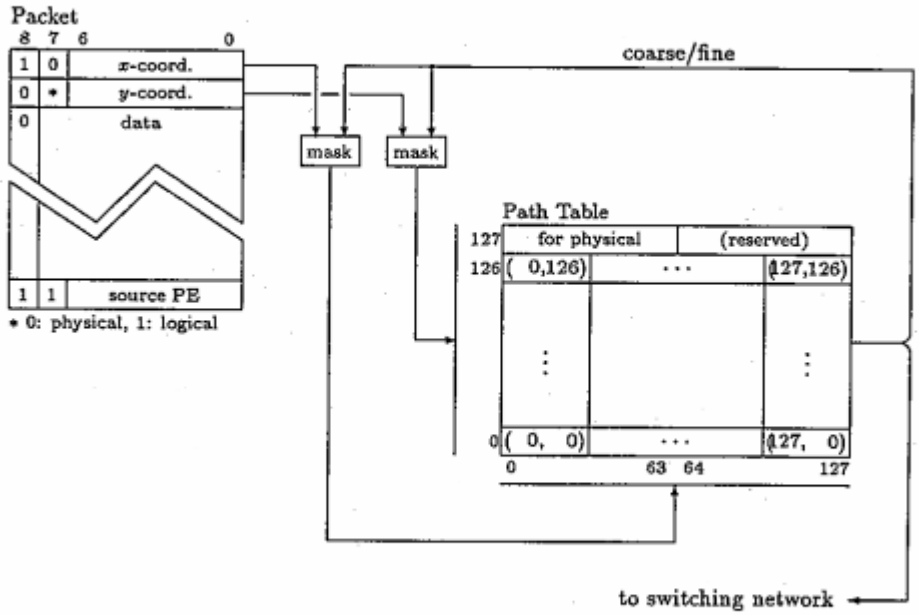


Figure 7: Path Table

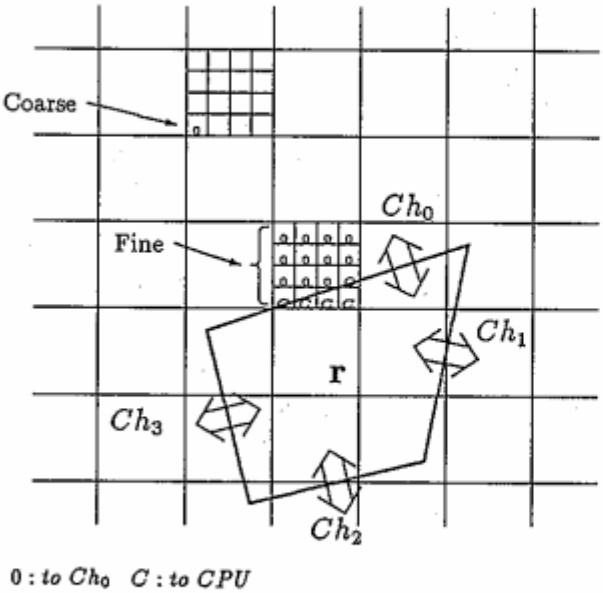


Figure 8: Coarse and Fine Coordinates

Table 2: Path Table Entry

b_2	b_1	b_0	
0	×	×	rough coordinate
1	×	×	fine coordinate
×	0	0	to CPU
×	0	1	to Ch_{i+1}
×	1	0	to Ch_{i+2}
×	1	1	to Ch_{i+3}

Thus, the PT has a nested structure as shown in Figure 8. In this figure, the coarse coordinates are two bits shorter than thefine coordinates.

4 CONCLUSION AND FUTURE WORKS

One of the unique features of the Multi-PSI/ version 2 system is the load balancing method using $\mathbf{P^3}$. The abstract computation power and communication cost enable programmers to specify their strategy of the load distribution, not being aware of the physical implementation of their system. The imbalance of the load of the physical processors is corrected dynamically using local information of the load. The goal throwing mechanism supported by the hardware makes it possible to transfer messages automatically to the destination represented by $\mathbf{P^3}$ coordinates

without global information.

There are three problems remaining in this load balancing method. The first problem is concerning to the network deadlock. Since a packet may turn any direction at network nodes, it is impossible to employ deadlock-free routing methods which limit the turning direction. The deadlock-free network architecture with a packet buffer pool could have been employed [8], because the length of the longest path in the network is finite. In future, however, more efficient method should be required, because the number of the nodes will be significantly larger than 64.

The second problem is in packet transmission while the PT is being updated. The algorithm of the goal throwing is based on the fact that adjacent two PEs identically evaluate the relation between the destination point and the shared edge. If a packet is transmitted while the PT is updated, the evaluation of the shared edge may be different. This problem can be solved by stopping all packet transmission during update, but the solution may degrade the network performance.

The third problem is the measurement of the load. What the word "load" means is not quite clear. One possible method might be to give priority value to each process and consider the sum of such values of processes executed in unit time.

In addition to finding efficient solutions to the above problems, the following items must be studied.

Static Locality Analysis: Development of algorithms for automatically extracting locality information from programs where no explicit information is given. As cache or virtual memory systems does work with most of the programs which are written without even considering the existence of them, this study might be fruitful.

Parallel Algorithms: Development of parallel algorithms, considering communication locality and load balancing feasibility. This will probably lead to a set of algorithms quite different from sequential algorithms widely used currently. They might also be quite different from algorithms for parallel execution with the equal-distance assumption.

Acknowledgments

We would like to thank Dr. Shunichi Uchida, chief of the ICOT fourth laboratory, and the researchers in his laboratory for their valuable suggestions. The great efforts of the engineers at Mitsubishi Electric Corp. and Ooi Electric Co. realized our ideas into an actual machine.

References

[1] T. Chikayama. Parallel Inference System Researches in the FGCS Project. In *Proceeding of 4th Symposium on Logic Programming*, 1987.

[2] M. Sato et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceeding of 4th International Conference on Logic Programming*, 1987.

[3] K. Ueda. Guarded Horn Clauses. TR-103, ICOT, 1985.

[4] N. Ichiyoshi et al. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proceeding of 4th International Conference on Logic Programming*, 1987.

[5] Y. Kimura et al. An Abstract KL1 Machine and its Instruction Set. In *Proceeding of 4th Symposium on Logic Programming*, 1987.

[6] K. Taki. The parallel software research and development tool: Multi-PSI system. In *Proceeding of France-Japan Artificial Intelligence and Computer Science Symposium 86.*

[7] H. Nakashima et al. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proceeding of 4th Symposium on Logic Programming*, 1987.

[8] E. Raubold et al. A Method of Deadlock-Free Resource Allocation and Flow Control in Packet Networks. In *Proceeding of ICCC 1976.*

Appendix

A proof of the termination of the algorithm shown in Figure 5 is given. The principle of the proof is to show that there should not be the same region in the sequence of the regions $r_0, r_1, r_2, \ldots$, where;

(1) r_0 is the initial value of the variable r.

(2) $r_1, r_2, \ldots$ are the values of r asigned by the step s11.

That is,

$$\forall i > \forall j \geq 0, \quad ri \neq rj.$$

Lemma 1: Let $e_0, e_1, e_2, \ldots$ be the sequence of the edges determined by the steps s1,s8,s9 and s10,

$$\forall i \geq 0, \quad d(e_i, p) = d(r_i, p),$$

where $d(e_i, p)$ is distance from the edge e_i to the point p, and $d(r_i, p)$ is distance from the region r_i to point p.

Proof :

(1) If $i = 0$, it is clear that

$$d(e_0, p) = d(r_0, p),$$

because e_0 is the nearest edge of the region r_0.

(2) Suppose $d(e_k, p) = d(r_k, p)$,

(a) If $d(e_k, p) \neq d(r_{k+1}, p)$,

$$d(e_{k+1}, p) = d(r_{k+1}, p),$$

because the edge e_{k+1} is the nearest edge of the region r_{k+1} determined by the step s10.

(b) If $d(e_k,p) = d(r_{k+1},p)$, the edge e_k includes the nearest point p_n in region r_{k+1} to the point p. If p_n is not the vertex on e_k, shared by the region r_k and r_{k+1}, a line l from p_n to p passes through r_k or region r_{k+1}. If the line l passes r_k (or r_{k+1}), r_k (or r_{k+1}) must have a edge nearer than e_k. Therefore, the nearest point of r_{k+1} to the point p is one of the vertices of e_k, and one of the adjacent edges of e_k shares the nearest point. It is clear the adjacent edge sharing the nearest point is the nearest edge of r_k, and the edge is the value assigned to e by the steps s8 or s9. Thus,

$$d(e_{k+1},p) = d(r_{k+1},p).$$

Lemma 2:

$$^{\forall}i \geq 0, \quad d(r_i,p) \geq d(r_{i+1},p).$$

Proof: Since the edge e_i is shared by r_i and r_{i+1},

$$d(e_i,p) \geq d(r_{i+1},p).$$

Therefore, by **Lemma 1**,

$$d(r_i,p) = d(e_i,p) \geq d(r_{i+1},p).$$

Lemma 3: If

$$^{\forall}j >^{\forall} i \geq 0, \quad d(r_i,p) = d(r_{i+1},p) = \ldots = d(r_j,p),$$

the sequence of the regions $r_i, r_{i+1}, \ldots, r_j$ shares the vertex v, and

$$d(v,p) = d(ri,p).$$

Proof :

(1) If $j = i+1$, by **Lemma 1**,

$$d(e_i,p) = d(r_i,p) = d(r_j,p),$$

and by (2)(b) of the proof of **Lemma 1**, r_i and r_j shares the vertex v of e_i, and

$$d(v,p) = d(e_i,p) = d(r_i,p).$$

(2) If $j = i+2$, by **Lemma 1**,

$$d(r_i,p) = d(e_i,p) = d(r_{i+1},p)$$
$$= d(e_{i+1},p) = d(r_{i+2},p).$$

Therefore, e_i is a member of the set nearest_edges(r_{i+1},p). By (2)(b) of the proof of **Lemma 1**, e_i and e_{i+1} shares the vertex v, and

$$d(v,p) = d(e_i,p) = d(r_i,p).$$

(3) Suppose the lemma holds when $j \leq i+k$, but it does not when $j = i+k+1$. In this case, r_{i+k-2}, r_{i+k-1} and r_{i+k} shares the vertex v, r_{i+k-1}, r_{i+k} and r_{i+k+1} shares the vertex v', and $v \neq v'$. This causes the inconsistency that v and v' are different vertices of the edge e_{i+k-1}, and

$$d(e_{i+k-1},p) = d(v,p) = d(v',p).$$

Lemma 4: If

$$0 \leq^{\exists} i <^{\exists} j, \quad ri = rj,$$

$r_i, r_{i+1}, \ldots, r_j$ shares the vertex v, and v is included in (and not on the boarder of) the region R_{ij}, where R_{ij} is the union of $r_i, r_{i+1}, \ldots, r_j$.

Proof: By **Lemma 2**,

$$d(r_i,p) \geq d(r_{i+1},p) \geq d(r_{i+2},p) \geq \ldots \geq d(r_j,p),$$

and $r_i = r_j$. Thus,

$$d(r_i,p) = d(r_{i+1},p) = d(r_{i+2},p) = \ldots = d(r_j,p),$$

and $r_i, r_{i+1}, \ldots, r_j$ shares the vertex v (Lemma 3). Suppose the vertex v is on the border of R_{ij}. Let e be the edge of R_{ij} including v, and r_k be the region including e. Since e is a edge of the region R_{ij}, $e \neq e_{k-1}$ and $e \neq e_k$. By **Lemma 3**, e_{k-1} and e_k shares the vertex v, and by the steps s5 to s10, $e_{k-1} \neq e_k$. This causes the inconsistency that three edges of the region r_k share the vertex v.

Lemma 5:

$$^{\forall}j >^{\forall} i \geq 0, \quad r_i \neq r_j.$$

Proof: Suppose that there are i and j which satisfy

$$0 \leq i < j, \quad r_i = r_j.$$

By **Lemma 4**, the vertex v, shared by $r_i, r_{i+1}, \ldots, r_j$, is incleded in the region R_{ij}. And by **Lemma 3**,

$$d(v,p) = d(r_i,p) = d(r_{i+1},p) = \ldots = d(r_j,p).$$

If p is not included in R_{ij}, the line from p to v passes a region r_k, and $d(v,p) > d(r_k,p)$. This causes the incosistency that v is not the nearest point of r_k to the point p. And p is never included in R_{ij}, by the step s3.