自然言語処理 73-17
(1989. 6. 30)

トーラス型マルチプロセッサシステム上での Prolog並列処理手法

Parallel Processing of Prolog on a Torus Machine

真栄田 保　　川口 剛　　喜屋武 盛基

Tamotsu Maeda, Tsuyoshi Kawaguchi and Seiki Kyan

琉球大学

University of the Ryukyus

あらまし　トーラス型マルチプロセッサシステム上でのProlog並列処理手法を提案する。提案する手法では、すべてのＰＥが並列に木の探索を実行する。負荷分散のためにＰＥ間でノードの受け渡しを行なうことが必要になるが、本手法では、ノードのユニフィケーションに必要なすべての情報を通信するのではなく、探索木におけるノードの深さと左から数えて何番目かという情報のみを通信する。また、このような通信のみによってプロセッサ間の負荷分散がはかられるように次のような負荷分散手法が用いられる。各ＰＥは、まずホスト（計算機）からの指示によって次に解くノードを決定し、単独に一個のノードが与えられた時点から、自分自信の判断（深さ優先探索）または隣接ＰＥからの指示によって次に解くノードを決定する。全解探索の８クィーン問題のように、プログラム実行過程で生じるノード数が多い問題に対しては、ＰＥ数に近い加速指数が得られることが、シミュレーション結果から確かめられる。

Abstract Torus is one of the most promising interconnection topology for multiprocessor systems with a large number of processors. This paper presents a parallel execution scheme of Prolog programs on a torus machine. In the scheme, the node data used in communications is distinguished from that used in unification operations. The former consists of only two bytes. Moreover the scheme uses two types of procedures for load-balacing among processors. One of them is performed by global communication of the host with processors, and the other is executed using local communication between adjacent processors. Simulation experiments show that the speedup rate close to the number of processors can be achieved by the proposed scheme if the number of nodes in the search tree is sufficiently large as compared with the number of processors.

1. Introduction

Polog is a logic programming language which has recently received considerable attention for its application in the field of artificial intelligence. The implementation of Prolog programs on multiprocessor systems is important from the viewpoint of the computational efficiency.

Multiprocessor systems can be classified into two categories, tightly coupled systems and loosely coupled systems. About the implementation of Prolog programs on tightly coupled systems, Aida et al.[1] have reported an experimental result on a prototype machine and Kai et al.[2] have presented an efficient scheme for the load-balancing among processors. But it is difficult to construct tightly coupled systems with a large number of processors.

While loosely coupled systems overcome this difficulty, the choice of interconnection topology is critical for these systems. Miura et al.[3] have presented a scheme for the implementation of Prolog programs on tree machines. On the other hand, we have shown in [4] that torus machines are more efficient than tree machines for the parallel execution of the branch-and-bound method. (In this method, a given problem is solved by being partitioned into subproblems as in

the execution process of Prolog programs.)

In this paper we present a parallel execution scheme of Prolog programs on a torus machine. Although OR-, AND- and Stream-parallelisms can be used for the execution of Prolog, OR-parallelism is considered to be the most efficent [1]~[3]. Thus our scheme uses OR-parallelism as those presented in [1]~[3].

2. Prolog

A Polog program consists of a set of facts, a set of rules and a question. Fig.1 shows an example which is similar to that given by Conery et al.[5]. In this figure, constants are lowercase and variable are capitalized. Line(1) is a rule that X is a grandfather of Z if X is a father of Y and Y is a father of Z. There are two rules in this example. Line(3) represents a fact that curt is the father of elain, and lines (4) to (8) also denote facts. Finally, line(9) is a question that means "Who is a grandchild of sam?".

The Prolog program is executed in the following manner. First the question (9) is unified to (1) and is replaced by "f(sam,Y), f(Y,G)". The next goal "f(sam,Y)" is unified to (4) and Y is binded to larry. Further "f(larry,G)" is matched to (5) and G is binded to den. Thus we have a solution that den is a grandchild of sam. After backtracking, we have another solution "G=doug".

The execution process described above can be represented by the search tree shown in Fig.2, where each node denotes a sub-goal of the given goal (or in other words, a sub-question of the given question).

```
(1)gf(X,Z):-f(X,Y),f(Y,Z).
(2)gf(X,Z):-f(X,Y),m(Y,Z).
(3)f(curt,elain).
(4)f(sam,larry).
(5)f(larry,den).
(6)f(larry,doug).
(7)m(elain,john).
(8)m(peg,den).
(9)?:-gf(sam,G).
```

Fig.1 A Prolog Program

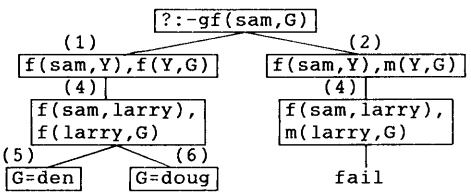

Fig.2 A search tree for the program shown in Fig.1.

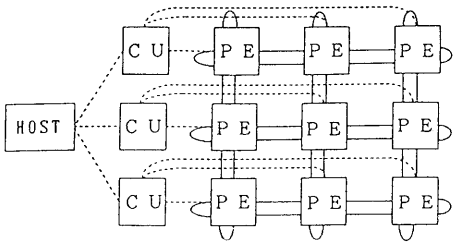

Fig.3 A torus machine.

3. Torus Machine

We present a torus machine shown in Fi.3 for parallel execution of Prolog programs. The machine consists of a host computer,m controllers and m^2 processors. We can view that m^2 processors are placed on intersections of m vertical lines and m horizontal lines. Thus unless otherwise speaking, processors are denoted by PE(i,j), $1 \leqq i \leqq m$ and $1 \leqq j \leqq m$, in this paper. Each processor has a processing unit and a memory unit. Further each processor is connected with four neighbors (PE(i,m) is connected to PE(i,1) for each i, $1 \leqq i \leqq m$, and PE(m,j) is connected to PE(1,j) for each j, $1 \leqq j \leqq m$.) A link connecting a pair of processors is called a local link.

Each controller is connected with m processors placed in a row of m×m array. Each controller receives a message from the host and sends it to all processors in the respective row. Messages from processors to the host are also transmitted by way of controllers. A link connecting a controller with the host is called a global link, and a link between a processor and a controller is also called so.

4. Parallel Execution of Prolog

4.1 Node Label and Path Table

An inference process of a Prolog program can be represented by a search tree as shown in Fig.4. In our scheme, m×m processors in a torus machine concurrently perform the unifications for the nodes in the search tree, and so each processor traces only one part of the search tree. Thus if a processor becomes idle, the processor needs to receive a new node from a neighbor.

A unification operation for a node v requires the data about a variable-binding-enviroment of v (that is, the data about unifications performed on the path from the start node to v). However if such data is directly transmitted between processors, the transmission time becomes very large.

(1)In our scheme, a label (depth(v),#(v))

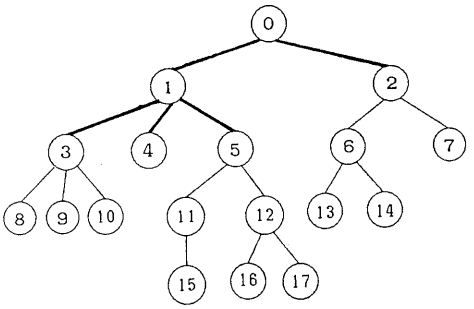

Fig.4 A search tree.

depth	0	1	2	3	4
1	1		1		
2	1		2		
3					
C_DEPTH→	2		2		

Fig.5 Path table TA.

is transmitted between processors, where depth(v) is the length of the unique path from the start node to v, and #(v) is the number assigned to v when all siblings of v are counted according to the "left to right" ordering. This label is called a node label of v. □

For example, assume a processor PE(i) has become idle after completing the process for node 4 in the search tree in Fig.4. Then PE(i) has all data about the subtree drawn by bold lines where nodes 0,1 and 4 have already been explored but the remaining nodes have not yet. Therefore if PE(i) receives a node label (2,1) from a neighbor, PE(i) judges the label to be node 3 and starts the processing for node 3.

It is assumed in the above discussion that the label of node 6 whose value is also (2,1) never comes to PE(i) from any of its neighbors. And such assumption really holds in our scheme.

(2)Each processor has a table TA to store the path traced by the processor itself and paths traced by its neighbors. This table is called a path table. In the table, a path is denoted by a sequence of labels of nodes which lie on the path. Each row i($\geq$1) of the table corresponds to depth i in the search tree. Further column 0 corresponds to the processor itself, and column j($1 \leq j \leq 4$) corresponds to left, right, upper and lower neighbor respectively. In addition, each processor has C_DEPTH(j), $0 \leq j \leq 4$ which indicates the depth of the node latest processed in the last unification by the processor corresponding to column j. Using these, the path traced by neighbor j is given by (k,TA(k,j)), $1 \leq k \leq$ C_DEPTH(j). Further ACCEPT(j) shows whether neighbor j can accept node-requirements or not. □

For example, if some processor which currently explores a node in the search tree shown in Fig.4 has the table in Fig.5, the processor can find that its right neighbor processed node 4 in the last unification.

In our scheme, each processor has a queue to store the labels of the nodes generated in the past unifications.

(3)The queue, denoted by Q, has one entrance and two exits. A node label is inserted into the rear of Q. Morever when a processor sends a node label to a neighbor as a reply to its node-requirement, it sends the label placed in the front of Q. On the other hand, if a processor needs a node label for the next unification of itself, the processor removes a label from the rear of Q. □

4.2 Node Selection Rules of Processors

A processor is said to be "controlled" if it performs the node selection according to an instruction of the host at each branching point of the search tree, and otherwise it is said to be "uncontrolled". Once a processor changes from controlled to uncontrolled, it never turns back to controlled. Further an uncontrolled processor is one of an individual, a master and a slave. Each of individuals and masters selects a node according to depth-first search. Each slave selects a node by an instruction of its master.

Our scheme uses the following rules for master-slave relation.

(i)If a processor PE(i) is a master of another processor PE(j) (or PE(j) is a slave of PE(i)), they are adjacent each other.

(ii)If a processor is a master of a neighbor, it cannot become a slave of another neighbor.

(iii)If a processor is a slave of a neighbor, it cannot become a slave nor a master of another neighbor.

4.3 Outline of the Proposed Scheme

If at least one processor is busy, the host sends a repeat message to all processors by way of controllers, and processors start unifications for nodes. As the result, if a processor becomes idle, it sends an idle message to the host. Otherwise it sends a busy message to the host. This procedure continues until all processors become idle.

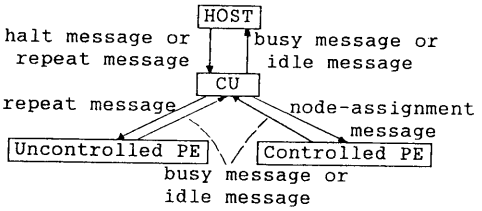

Fig.6 Messages used for global communications.

Initially, all processors are "controlled" and the host gives a start node (or a given goal) to all processors. After that, the following rules are recursively applied to controlled processors.

(i) If all controlled processors in $x(1 \leqq x \leqq m)$ rows perform the unification for the same node and $y(\geqq 0)$ children are generated from the node, these children are equally distributed into the x rows. (As the result, if $x \leqq y$ then $\lceil y/x \rceil$ or $\lfloor y/x \rfloor$ nodes are assigned to a row, and otherwise a node is assigned to $\lceil x/y \rceil$ or $\lfloor x/y \rfloor$ rows.)

(ii) Assume $s(\geqq 0)$ nodes are assigned to a row with $t(1 \leqq t \leqq m)$ controlled processors. If $s \geqq t$ these nodes are equally distributed into the t processors, which change from "controlled" to "individual". Otherwise, s-1 nodes are assigned one by one to the first s-1 processors and the remaining node is assigned to each of the last t-s+1 processors. And the first s-1 processors change from "controlled" to "individual".

An individual explores nodes according to depth-first search. That is, if an individual PE(i) has more than one nodes to be executed, it selects the node latest genereated in itself for the next unification. As mentioned in 4.1, the label of this node is placed in the rear of the queue of PE(i).

Moreover when PE(i) becomes idle, it requires a node to a neighbor PE(j). Then PE(j) returns the node label placed in the front of its queue if PE(i) has already explored the parent of the node (that is, PE(i) can immediately process the node). Otherwise, PE(j) makes PE(i) its slave. The function NODE_CHECK shown in Fig.12 is used for such decision. Moreover in the latter case, PE(i) follows the master PE(j) in the search tree (or in other words, a slave traces the same path as the master did).

For example, assume a processor having the table in Fig.5 has just required a node from the right neighbor. If a node label (2,3) is placed in the front of its queue, the processor sends the node label to the neighbor. On the other hand, if (3,2) is placed in the front of the queue, the processor makes the neighbor its slave. In the latter case, the neighbor backtracks to node 3 using its path table and starts to follow the master in the search tree.

```
procedure HOST
begin
  if a busy message or an idle message
  is sent from a controller then begin
    if it is a busy message then
      BUSY#:=BUSY#+1;
    else
      IDLE#:=IDLE#+1;
    if BUSY#+IDLE#=m then
      if BUSY#=0 then
        send a halt message to each
        controller;
      else begin
        send a repeat message to each
        controller without controlled
        processors;
        partition the set of rows hav-
        ing controlled processors into
        subsets Rj such that rows x and
        y are included in the same Rj
        iff controlled processors in
        rows x and y executed the same
        node in the last unification;
        for each Rj do
          for each row i included in Rj
          do
            let a and b denote the
            smallest and the largest
            indices of the nodes assig-
            ned to row i using rule (i)
            described in 4.3;
            send a repeat message with
            (a,b) to the controller of
            row i;
        BUSY#:=IDLE#=0;
      end;
  end;
end.
```

Fig.7 Procedure for the HOST.

After performing the unification for a node, a master sends an instruction to each of its slaves. If a master becomes idle, it breaks off a master-slave relaiton between each slave. Otherwise, the master does the following for each slave PE(j) : if the function NODE_CHECK returns true for PE(j), the master sends PE(j) the node label placed in the front of the queue and changes PE(j) to individual; otherwise the master keeps on a master-slave relation between PE(j).

4.4 Details of the Proposed Scheme

Fig.7 and 8 show the procedures for the host and a controller. Moreover each processor performs according to the process given in Fig.14, which uses the procedures indicated in Fig.9 to 13. To

```
procedure CONTROLLER
{let i be the index of this controller}
begin
  if a halt message is sent from the
  host then
    send a halt message to each
    processor in row i;
  if a repeat message is sent from the
  host then begin
    send a repeat message to each
    uncontrolled processor in row i;
    if CONTROLLED#>0 then
      for each controlled processor
      PE(i,j) do
        let a and b denote the smallest
        and the largest indices of the
        nodes assigned to PE(i,j) using
        rule (ii) described in 4.3, and
        let R be the role of PE(i,j)
        determined by rule (ii);
        send a node-assignment message
        (R,(a,b)) to PE(i,j);
  end;
  if a busy message or an idle message
  is sent from a processor then begin
    if it is a busy message then
      if the message includes a pair of
      number (k,j) where k is the
      number of nodes and j is an index
      of a processor then begin
        NODE#:=k;
        CONTROLLED#:=m-j+1;
        BUSY#:=BUSY#+CONTROLLED#;
      end;
      else
        BUSY#:=BUSY#+1;
    else
      IDLE#:=IDLE#+1;
    if BUSY#+IDLE#=m then bgin
      if BUSY#=0 then
        send an idle message to the
        host;
      else
        if CONTROLLED#>0 then
          send a busy message with
          NODE# to the host;
        else
          send a busy message to the
          host;
      BUSY#:=IDLE#:=CONRTROLLED#:=0;
    end;
  end;
  if a solution is sent from a
  processor then
    send the solution to the host;
end;
```

Fig.8 procedure for a controller

help interpret these procedures, we will give several remarks below.

Each processor has the following fields in its memory.

A : a variable used for storing a label of a node for which the next unification is performed,

```
procedure SET_ROLE_AND_NODE
begin
  let (R,(a,b)) be a node-assignment
  message from controller and let (d(i),
  #(i)),1≦i≦k,be the labels placed in
  Q;
  ROLE:=R;
  if a=b then
    set A to (d(a),#(a)) and make Q
    empty;
  else
    set A to (d(a),#(a)) and leave
    only (d(i),#(i)),a<i≦b,in Q;
end;

procedure UNIFICATION
begin
  perform the unification operation for
  the node whose label is placed in A;
  if the node generates children then
    if ROLE≠slave then
      set A to the label of the left-
      most child and place the labels
      of the remaining children in the
      rear of Q;
    else begin {a solution is found or
    the unification fails}
      if a solution is found and ROLE=
      (controlled leader or master or
      individual) then
        send the solution to the
        controller;
      A:=φ;
    end;
end;
```

Fig.9 SET_ROLE_AND_NODE and UNIFICATION

Q : a queue used for storing labels of active nodes (remember (3) in 4.1),

ROLE : a variable keeping the role of the processor which is one of controlled leader, controlled nonleader, individual, master and slave,

SLAVESET : a queue used for storing slaves of the processor when ROLE=master.

As mentioned in 4.1, Q has one entrance and two exits. Thus we define the following functions: INSERT_REAR(Q,x) places x in the rear of Q; DELETE_REAR(Q) returns the element placed in the rear of Q and deletes it from Q; DELETE_FRONT(Q) returns the element placed in the front of Q and deletes it from Q.

Moreover the messages shown in Fig.6 are used for global commmunications. In addition, the following messages are used for local communications.

(1) A state message (α_s, β_s) is sent from a processor to each of its neighbors, where $\alpha_s \in$ {true,false} and β_s is a label of a node for which the next unification is

```
procedure SEND_INSTRUCTION_TO_SALVE
begin
  repeat
    if a request message is sent from
    slave j then begin
      if A=φ then
        M:=(individual,φ);
      else
        if NODE_CHECK(j)=true then
          M := ( individual, DELETE_
          FRONT(Q));
        else begin
          let (d,#) be the node label
          in A if C_DEPTH(0)=C_DEPTH(j),
          and otherwise let d=C_DEPTH(j)
          +1 and #=TA(d,0);
          M:=(salve,(d,#));
        end;
    end;
    send a relation message M to slave
    j;
  until (request messages are sent from
        all slaves in SLAVESET)
end;

procedure RECEIVE_INSTRUCTION_FROM_
          MASTER
begin
  send a request message to the master;
  wait until a relation message is
  sent from the master;
  if the message instructs the change
  to individual then
    ROLE:=individual;
  A:=node label in the message;
end;
```

Fig.10 SEND_INSTRUCTION_TO_SLAVE and RECIEVE_INSTRUCTION_FROM_MASTER.

performed. α_s is true if the processor can accept a node-requirement, and otherwise α_s is false (see procedure COMMUNICATION_WITH_NEIGHBORS)

(2) A node-requirement message is sent from an idle processor to a neighbor (see procedure NODE_REQUIREMENT).

(3) A relation message (α_R, β_R) is sent from a master to each of its slaves, where $\alpha_R \in$ {empty,individual,slave} and β_R is empty or a node label (see prcedure SEND_INSTRUCTION_ TO_SLAVE). The message is also used as a reply to a node-requirement message (see procedure RECEIVE_NODE_REQUIREMENT).

4.5 Example

Fig.15 gives an example of our scheme, in which the tree in Fig.4 is searched on a 3×3 torus machine. C, I, M and S denote a controlled processor, an individual, a master and a slave, respectively. Further "×" represents that the corresponding processor is idle, and "W" denotes that the corresponding slave is waiting an instruction from the

```
procedure COMMUNICATION_WITH_NEIGHBORS
begin
  repeat
    for each neighbor j do begin
      if a state message from j has
      already been stored in I/O buffer
      then begin
        let (ACC,(d,#)) be the state
        message;
        ACCEPT(j):=ACC;
        TA(d,j):=#;
        C_DEPTH(j):=d;
      end;
      if a state message from PE has
      not been sent to j then begin
        if A≠φ and ( ROLE=master or
        individual) then
          ACC:=true;
        else
          ACC:=false;
        (d,#):=node label in A;
        send a state message (ACC,(d,#))
        to j;
      end;
    end;
  until (all columns in the path table
  are renewed and state messages are
  sent to all neighbors)
end;
```

Fig.11 COMMUNICATIO_WITH_NEIGHBORS.

master about the node selection.

For example in the 3rd repetition, PE(1,2) becomes idle after executing node 4 and requires a node label to PE(1,1). PE(1,1) has the path table shown in Fig.5, and the label placed in the front of its queue is (3,2) which is the label of node 9. Therefore the function NODE_CHECK returns "false". Thus PE(1,2) becomes a slave of PE(1,1) and follows PE(1,1). (Node 3 is the first node that PE(1,1) has explored but PE(1,1) has not yet.)

5. Performance Evalution

Using simulation experiments, we estimated the performance of our scheme for parallel execution of Prolog programs on a torus machine. It was assumed in the experiments that

(i) the computation time needed by the unification operation is the same for each node, which is denoted by t_u,

(ii) the transmission time of the messages shown in 4.4 is so small that it can be disregarded as compared with t_u.

Let T(n) be the computation time required to solve a problem by a torus machine with n processors where n is given by m^2 for an integer $m(\geq 1)$. Then the speedup rate RS(n) of a torus machine is defined by RS(n)=T(1)/T(n).

Fig.16 shows the results for the N-queen problem (N=6,7,8) and the example

```
procedure NODE_CHECK(j)
{it is assumed in this function that a
processor is required a node label from
a neighbor j, j=left, right, upper and
lower}
begin
  if Q=φ then
    return(false);
  else begin
    (d0,#0):=DELETE_FRONT(Q);
    if TA(1,0)=TA(1,j) then
      find the greatest index d such
      that TA(k,0)=TA(k,j) for all k
      from 1 to d;
    else
      d:=0;
    if d0=d+1 then
      return(true);
    else
      return(flase);
  end;
end;

procedure NODE_REQUIREMENT
begin
  find a neighbor j with the highest
  priority for which ACCEPT(j)=true;
  if such neighbor j exists then begin
    send a node-requirement message to
    j;
    wait a relation message returuned
    from j;
    if the message instructs the change
    to slave then begin
      ROLE:=slave;
      find d such that TA(k,0)=TA(k,j)
      for all k, 1≦k≦d, and TA(d+1,0)
      ≠TA(d+1,j);
      A:=(d+1,TA(d+1,j)); {A is set to
      the label of the first node that
      neighbor j has explored but PE
      itself has not yet}
    end;
    else
      A:=node label in the message;
  end;
end;
```

Fig.12 NODE_CHECK and NODE_REQUIREMENT.

```
procedure RECEIVE_NODE_REQUIREMENT
begin
  if a node requirement message is sent
  from a neighbor j then
    if NODE_CHECK(j)=true then
      send a relation message (φ,
      DELETE_FRONT(Q)) to j;
    else begin
      send a relation message (slave,
      φ) to j;
      ROLE:=master;
      INSERT_REAR(SLAVESET);
    end;
end;
```

Fig.13 RECEIVE_NODE_REQUIREMENT.

```
procedure PE
begin
  if a halt message is sent from the
  controller then
    halt;
  if a repeat message or a node-
  assignment message is sent from the
  controller then begin
    if ROLE=controlled leader or ROLE=
    controlled nonleader then
      SET_ROLE_AND_NODE;
    UNIFICATION;
    if ROLE = ( controlled leader or
    controlled nonleader) and A=φ then
      ROLE:=individual;
    if ROLE=slave then
      RECEIVE_INSTRUCTION_FROM_MASTER;
    if ROLE=master then begin
      SEND_INSTRUCTION_TO_SLAVE;
      if A=φ or SLAVESET=φ then
        ROLE=individual;
    end;
    COMMUNICATION_WITH_NEIGHBORS;
    if A=φ then begin
      NODE_REQUIREMENT;
      if A≠φ then
        send a busy message to the
        controller;
      else
        send an idle message to the
        controller;
    end;
    else begin
      if ROLE=controlled leader then
        send a busy message with a pair
        of numbers (k,j) to the
        controller where k denotes the
        number of nodes generated in
        UNIFICATION and j is the index
        of the processor;
      else
        send a busy message to the
        controller;
      repeat
        RECEIVE_NODE_REQUIREMENT;
      until (a repeat message is sent
      from the controller)
    end;
  end;
end;
```

Fig.14 Procedure for a processor.

used in reference [1], where an expression $2x \cdot \exp(x^2)+x^2((2x) \cdot \exp(x^2))$ is reduced to $2x(1+x^2) \cdot \exp(x^2)$ using a Prolog program.

Each result shown in Fig.16 is the one for the case when all solutions are searched, and in this case the speedup rate is always bounded by the number of processors. Further the number of nodes (sub-goals) in the search tree is 513 for the example in [1], and that number for 6-,7- and 8-queen problem is 7325, 31344 and 140892, respectively. It is found from Fig.16 that the speedup rate close

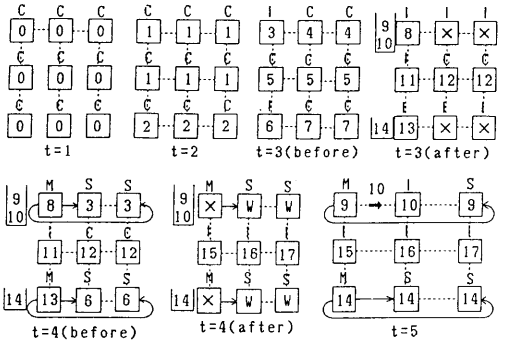

Fig.15 Parallel execution of the search tree shown in Fig.4.
(before: before performing the unifications
after : after performing the unifications)

to the number of processors can be achieved by our scheme if the number of nodes in the search tree is sufficiently large as compared with the number of processors.

We should notice that even if an ideal multiprocessor system were used for a problem in which all solutions are searched, RS(n) could not attain to n. Let $T_{OPT}(n)$ be the computation time to solve a problem by using an ideal multiprocessor system in which any amount of data can be transmitted between any pair of processors in an infinitely small time. And define $RS_{OPT}(n)$ by $T(1)/T_{OPT}(n)$. Under the assumptions (i) and (ii), we can compute $T_{OPT}(n)$ for each problem in Fig.16 using a result by Hu[6]. Fig.17 shows a comparison of RS(n) with $RS_{OPT}(n)$ for each of 6-queen and the example in [1].

6. Conclusion

A scheme has been presented for parallel execition of Prolog programs on a torus machine. When a communication of a node is needed for load-balacing among processors, a two-byte data about the node is transmitted. Moreover to obtain a good load-balacing by simply transmitting such data, centralized node-assignment and distributed node-assignment are used in the scheme.

The performance of the scheme was estimated using simulation experiments. Results for 8-queen problem showed that $RS(n) \geq 0.75n$ for $n \leq 50$ and $RS(n) \geq 0.51n$ for $n \leq 100$ where n is the number of processors and RS(n) denotes the speedup rate for the torus machine with n processors. Further it was shown that $RS(n)/RS_{OPT}(n) \geq 0.34$ for any $n \leq 100$ and for each of the problems used in simulation experiments, where $RS_{OPT}(n)$ denotes the speedup rate obtained by an ideal multiprocessor system with n

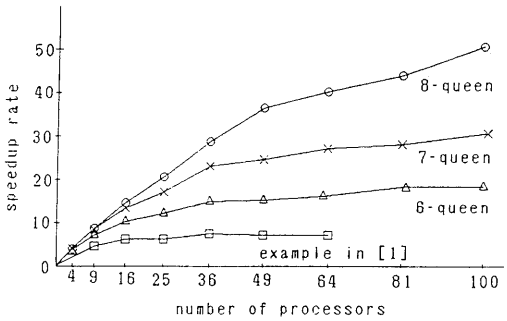

Fig.16 Performance of the scheme.

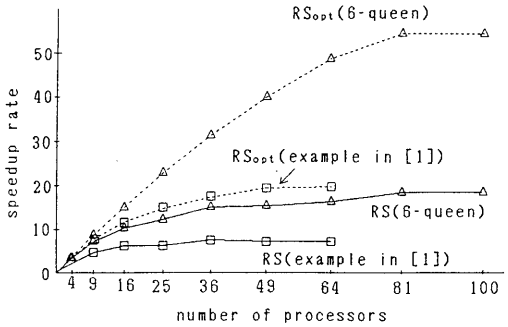

Fig.17 Comparison of RS(n) with $RS_{OPT}(n)$.

processors in which any amount of data can be transmitted between any pair of processors in an infinitely small time.

References

[1]H.Aida, H.Tanaka and T.Moto-oka : "On parallel processing system "Paralog"", Jour.of IPS of Japan, Vol.24 (1983), pp.830-837, in Japanese.

[2]M.Kai, K.Kobayashi and H.Kasahara : "An OR parallel processing scheme of Prolog using hierarchical pincers attack search", Jour.of IPS of Japan, Vol.29(1988), pp.647-655,in Japanese.

[3]H.Miura, M.Imai, M.Yamashita and T.Ibaraki: "Implementation of parallel Prolog on tree machines", Proc.of the 6th Fall Joint Computer Conference, Dallas, Texas (1986).

[4]T.Kawaguchi and T.Maeda : "A parallel branch-and-bound algorithm on a torus machine", Trans.of IEICE Japan, Vol. J72-D-I, in Japanese.

[5]J.S.Conery and D.F.Kibler : "Parallel interpretation of logic programs", Proc.of the ACM Conference on Functional Programming Languages and Computer Architecture, pp.163-170 (1981).

[6]T.C.Hu : "Parallel sequencing and assembly line problems", Operations Research, 9 ,6, pp.841-848 (1961).