International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:4, 2009

Specialization-based parallel Processing without Memo-trees

Hidemi Ogasawara, Kiyoshi Akama, and Hiroshi Mabuchi

Abstract—The purpose of this paper is to propose a framework for constructing correct parallel processing programs based on Equivalent Transformation Framework (ETF). ETF regards computation as In the framework, a problem's domain knowledge and a query are described in definite clauses, and computation is regarded as transformation of the definite clauses. Its meaning is defined by a model of the set of definite clauses, and the transformation rules generated must preserve meaning. We have proposed a parallel processing method based on "specialization", a part of operation in the transformations, which resembles substitution in logic programming. The method requires "Memo-tree", a history of specialization to maintain correctness. In this paper we proposes the new method for the specialization-base parallel processing without Memo-tree.

Keywords—Parallel processing, Program correctness, Equivalent transformation, Specializer generation rule

I. Introduction

As distributed-memory parallel processing computers (DMPC), such as PC clusters, have become more popular, parallel processing programs which run efficiently in such environment have been in great demand. Even it is not easy task to guarantee correctness of a sequential processing program, construction of a parallel processing program which achieves efficiency and correctness is more difficult, because of the fact that the program should control not only computation in each computational node whose computational state is independent from each other, but also interaction among the nodes. One way to deal with this problem is to implement a correct algorithm which has developed so far to a given problem, but development of new algorithm might be required for new problem domain or new computational environment. We have proposed a new general framework for constructing parallel processing programs which are correct and efficient [1], [2].

Our framework is based on Equivalent Transformation Framework (ETF) [3] which regards computation as transformation of logical formulae. This framework consists of the following three steps:

- define a problem
- construct Equivalent Transformation Rule (ETR)
- construct a program

In the first step, a given problem is defined in the two set of definite clauses: domain knowledge of the problem, and a query in the domain. Each ETR constructed in the second step is a rule to transform the query clause. The equivalent transformation (ET) defined by an ETR preserves "meaning" of the query and the domain knowledge, i.e. a model of the query before the transformation is equal to the queries after the transformation. Given the ETRs and a computational resource, e.g. single core CPU computer, multi-core CPU computer, or DMPC, the program that executes ET defined by the ETRs efficiently in the given resources is constructed. Our framework is one instance of ETF where DMPC is given as the computational resource.

From the cause of the difficulty in parallel processing programming mentioned above, it is important to decide the following two factors: computation in each node and communication among the nodes. Transformation of the query by an ETR application consists of two operation: specialization and body atom replacement. The specialization is generalized notion of substitution in logic programming (LP), and replace a term in the query with another term. The body atom replacement corresponds to resolution in LP. In the parallel processing system proposed by Akama et al.[1] and Ogasawara et al.[2], computations of information about specialisations are processes in parallel, called Specialization-based Parallel Processing (SBPP). The studies targeted DMPC architecture, and proposed computation executed as follows by one master role process and worker role processes.

- The master controls transformation of the query clause
- The master sends a part of the body atoms in the query to a worker requests partial information about specialization for the query clause
- The worker computes the requested the specialization information and replies it to the master
- The master transforms the query clause based on the received information

The difficulty in this process is correctness of the transformation executed in the last step. The specialization information has been computed from the body atoms in the query clause at the first step, Within the latency after sending the body atoms before receiving the substitution information, other transformations would be applied on the query in the master, and the body atoms might be changed, or even be removed when the specialization information is received. The studies proposed the adjusting algorithm of the specialization information received by specialization history within the latency. For this adjustment, the system stores a specialization log called "Memo-tree".

In this paper, we propose a lightweight adjusting algorithm that does not use the Memo-tree.

II. Equivalent Transformation Framework

A. Specialization System

A substitution in LP is defined by a set of bindings of variables, and application of a substitution on an atom is associated with a mapping on a set of atoms. "Specialization"

is the generalized notion of this substitution, and it defines a partial mapping on a set of atoms. Like a most general unifier in LP, two terms will be unified by a specializer.

A specialization system on a set $\mathcal{A}$ is defined by 4-tuples, $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$, where $\mathcal{A}$ is called a set of objects, $\mathcal{G}$ is an interpretation domain, and $\mathcal{S}$ is a set of specializations. $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ satisfies the following conditions.

1) $\mu : \mathcal{S} \Rightarrow partial\_map(\mathcal{A})$
2) $\forall s_1, s_2 \in \mathcal{S}, \exists s \in \mathcal{S} : \mu(s) = \mu(s_2) \circ \mu(s_1)$
3) $\exists s \in \mathcal{S}, \forall a \in A : \mu(s)(a) = a$
4) $\mathcal{G} \subseteq \mathcal{A}$

The first condition is that a specialization s provides a partial mapping on $\mathcal{A}$. This partial mapping defines the effects of application of the specialization. The second condition is the existence of the composition of any two specializations.

For example, substitution in LP is defined as a specialization system as follows. Consider a set of atoms $\mathcal{A}$, a set of ground atoms $\mathcal{G}$, and a set of bindings $\mathcal{S}$. Substitution of terms is then defined by $\mu(\theta)$ where $\theta \in \mathcal{S}$. Note that this mapping is not on terms. $\mu(\theta)$ maps an atom to another atom by replacing a variable in the atom with a term. By redefining $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$, an extended substitution, like substitution on structured variable used in Constraint Logic Programming, becomes available.

B. Equivalent Transformation

Based on a set of atoms defined by a specialization system, a class of definite clause is defined. In ETF, a problem is defined by a set of definite clauses. Let $\mathcal{P}$ be a set of definite clauses. $\mathcal{M}(\mathcal{P})$, the model of $\mathcal{P}$, is a set of head atoms of a clause whose head atoms are grounded, and whose body atom set is empty or a subset of $\mathcal{M}(\mathcal{P})$. Equivalent Transformation theory is a computational theory that defines computation as a transformation of $\mathcal{P}$. Equivalency in ET means that a model of $\mathcal{P}$ before and after a transformation is equivalent.

To solve a problem in ETF, $\mathcal{P}$ is divided into a domain knowledge part of the problem and queries of the problem. The query is represented by the special definite clause called "ans clause," whose predicate of the head atom is "ans." The answer of the problem is provided as the grounded ans clauses by transforming them. ET Rule (ETR) defines these transformations, which have the following pattern:

$$\mathsf{atom\_list1}, \{\mathsf{condition\_list}\} \Rightarrow \{\mathsf{execute\_list}\}, \mathsf{atom\_list2}.$$

The left side of "⇒" is called the *head*, and the right side, the *body*. Some rule, called a multiple-body rule, has multiple bodies. The "atom_list1," is called *head atoms*, and "atom_list2," *body atoms*. Both are lists of atoms. The "condition_list" and "execute_list" are lists of atoms that are user-defined predicates or built-in predicates.

An ETR is applied to an ans clause in the following steps.

(a) *The rule head check:* Test the existence of atoms that match atom_list1, and execute {condition_list}. If the test or the execution fails, the rule is not applied.
(b) *The ans clause specialization:* Execute {execute_list} of the rule. First, the ans clause is specialized by the specialization that is computed by step (a). The execution of {execute_list} then specializes the ans clause. If the execution fails, then the ans clause is deleted.
(c) *Replacing body atoms in the ans clause:* Replace the atoms in the ans clause that match the atom_list1 in step (a) with atom_list2.

If a multiple-body rule is matched in step (a), the ans clause is copied and each body is applied to each ans clause in steps (b) and (c).

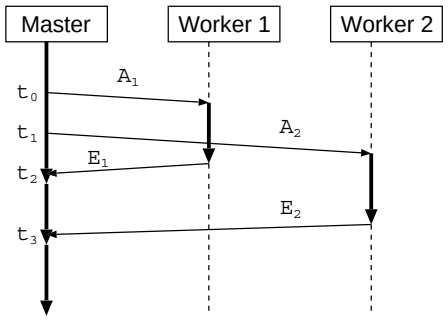

Fig. 1. Example of SBPP

III. Specialization-based Parallel Processing

A. SBPP with Memo-tree

The second step in the application process of ETR can be divided into the following two parts: computation of specialization to be applied and application of the specialization computed. The former part does not affect the ans clause, and can be executed in the other process. In SBPP, each worker computes this information in parallel. The master requests information about specialization by sending a set of body atom that will be match with ETR. After receiving the specialization information from a worker, it transforms the ans clause, i.e. specializes the ans clause, and replaces its body atoms.

For example, in Fig. 1 the master sends the body atom set A_1 to Worker 1 at t_0 and A_2 to Worker 2 at t_1, and receives the specialization information E_1 from Worker 1 at t_2 and E_2 from Worker 2 at t_3. E_1 is computed by Worker 1 from A_1, and E_2 by Worker 2 from A_2. The specialization based on E_1 is applied on the ans clause at t_2, and the other specialization based on E_2 is applied on the ans clause at t_3, which has been specialized at t_2. Note that application of a specialization based on the response message (E_2) on the ans clause does not guarantee correctness, because within the latency after sending A_2 before receiving E_2. A_2 might be transformed. Let a specialization applied within this latency be γ. E_2 is computed from A_2, but if γ was applied before t_3, a specialization computed from $A_2\gamma$ would not be E_2, and the specialization based on E_2 would not be correct.

To maintain correctness of computation, Akama et al.[1] and Ogasawara et al.[2] propose SBPP algorithm which uses "Memo-tree", a log which contains history of communication between the master and the workers and specialization applied on Master's ans clauses. In the system, the information about specialization is represented by "specializer", an atom with two arguments. It means the first argument should be

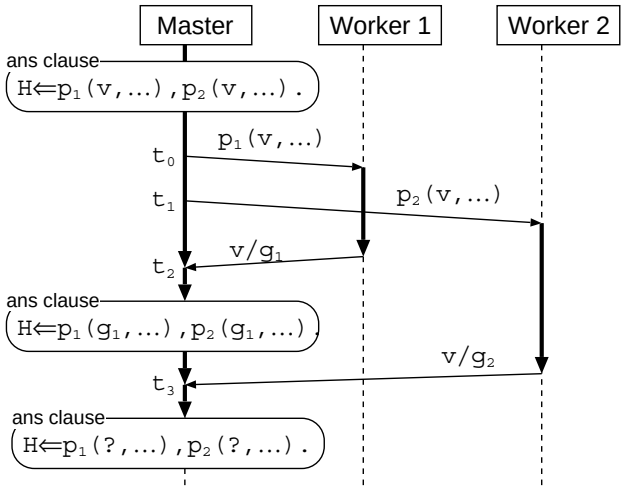

Fig. 2. Difficulty of SBPP

specialized to the second arguments. When the master receives a specializer E computed from a body atom that has been sent to Worker at time t, the master specializes E by a sequence of specializations that has been applied on the ans clause after t. Then it adds the specialized E in the body atoms, and applies the specialization represented by it. In this method maintenance of specialization history in the master is required, We introduce the other lighter algorithm.

B. SBPP without Memo-tree

Before introducing a new algorithm for SBPP, we indicates the difficulty of SBPP again using an example illustrated in Fig. 2. Assume the master has the following ans clause in the beginning:

$$H \Leftarrow p_1(v), p_2(v).$$

The master sends $p_1(v)$ to the worker 1 at t_0, and $p_2(v)$ to the worker 2 at t_1. The worker 1 computes v/g_1, a correct specialization for $p_1(v)$, and returns it to the master at t_2. The correctness of v/g_1 means that application of the specialization v/g_1 on an ans clause which includes $p_1(v)$ in its body preserves its meaning. After receiving v/g_1 at t_2, the master applies this specialization on the ans clause:

$$H \Leftarrow p_1(g_1), p_2(g_1).$$

While this specialization is executed, the worker 2 computes v/g_2, a correct specialization for $p_2(v)$, and returns it to the master at t_3. Because v in the ans clause has been specialized to g_1 when the master receives v/g_2, the master cannot specialize v any more. The former algorithm adjusts v/g_2 by the v/g_1 before the specialization based on v/g_2 is applied.

One of the major function of the adjustment is to identify a term in the master that corresponds a receiving specialization, e.g. to identify g_1 in the master that corresponds v/g_2 arriving from the worker 2 at t_3. The new algorithm solves this problem by introducing a "term list atom" which records this correspondence. A term list atom is $\mathrm{vl}(id, termlist)$ where id is a unique identifier and $termlist$ is a list of terms.

We explains application of term list atoms using the example of Fig.2. The clocks from t_0 to t_3 corresponds to the time series of interactions of the master and the workers in Fig. 2. "$[\mathtt{M} \rightarrow \mathtt{W}]$:" denotes the item following it is sent from the master to a worker, and "$[\mathtt{W} \rightarrow \mathtt{M}]$:" denotes its reverse.

t_0 Instead of sending the body atom $p_1(v)$ to the worker 1 in the original process flow of Fig.2, the master generates the unique identifier i_1, and creates a term list atom $\mathrm{vl}(i_1, \{v\})$ from the identifier and a list of variables in $p_1(v)$ that is to be sent to a worker. Then master adds this term list atom to the ans clause:

$$H \Leftarrow p_1(v), p_2(v), \mathrm{vl}(i_1, [v]).$$

Also the master sends the copies of the body atom and the term list atom:

$$[\mathtt{M} \rightarrow \mathtt{W}] : p_1(v_{c1}), \mathrm{vl}(i_1, [v_{c1}])$$

The "copy" of an atom is the same atom except variables in it are renamed to new unique variables.

t_1 The master generates the unique identifier i_2, and creates a term list atom $\mathrm{vl}(i_2, [v])$ from the identifier and a term list in $p_2(v)$. Then master adds this term list atom to the ans clause:

$$H \Leftarrow p_1(v), p_2(v), \mathrm{vl}(i_1, [v]), \mathrm{vl}(i_2, [v]).$$

Also the master sends the copies of the body atom and the term list atom:

$$[\mathtt{M} \rightarrow \mathtt{W}] : p_2(v_{c2}), \mathrm{vl}(i_2, [v_{c2}])$$

t_2 The worker 1 computes the specialization v_{c1}/g_1 from $p_1(v_{c1})$, instead of v/g_1. It returns a term list atom with the same identifier and the result of the specialization v_{c1}/g_1:

$$[\mathtt{W} \rightarrow \mathtt{M}] : \ \mathrm{vl}(i_1, [g_1])$$

The master receives it and extracts the term list atom with the same identifier from the ans clause:

$$\mathrm{vl}(i_1, [v])$$

By matching these two term list atoms, it becomes clear that the g_1 and v was the same term(v at t_0), and that should be unified. So the master applies the specialization v/g_1 on the ans clause:

$$H \Leftarrow p_1(g_1), p_2(g_1), \mathrm{vl}(i_2, [g_1]).$$

Note that $\mathrm{vl}(i_1, [v])$ is removed.

t_3 The worker 1 computes the specialization v_{c2}/g_2 from $p_2(v_{c2})$. It returns a term list atom:

$$[\mathtt{W} \rightarrow \mathtt{M}] : \ \mathrm{vl}(i_2, [g_2])$$

The master receives it and extracts the term list atom with the same identifier from the ans clause:

$$\mathrm{vl}(i_2, [g_1])$$

By matching these two term list atoms, it becomes clear that the g_2 and g_1 are specialized values for the same variable, v at t_1, and that should be unified. So the master computes the most general unifier θ of g_2 and g_1, and applies it on the ans clause:

$$H \Leftarrow p_1(g_1\theta), p_2(g_1\theta).$$

The main feature of this method is the identification method of a term in the ans clause and a term arriving from a worker, e.g. g_1 and g_2 in the above example. The two terms must be equal because both comes from the same variable in the former ans clause. We introduces eqaul atom to represent this equality constraint. In the following subsection, we describes the algorithm of the master and the workers, then we discuss this equality constraint based on specialization systems.

C. Master/Worker Algorithm for SBPP

Let C be an ans clause in the master. $\mathrm{head}(C)$ denotes the head atom of C, and $\mathrm{body}(C)$ a set of body atoms of C. Let A be a set of atoms, and E be a specializer that represents specialization information. $\mathrm{gen}(A)$ denotes a specializer computed a set of atoms A, and $\mathrm{spec}(E)$ denotes a specialization represented by E. $C \oplus A$ denotes the following definite clause:

$$\mathrm{head}(C) \Leftarrow \mathrm{body}(C) \cup A.$$

and $C \ominus A$ denotes the following definite clause:

$$\mathrm{head}(C) \Leftarrow \mathrm{body}(C) - A.$$

Algorithm 1, 2 show the algorithms of Master and Worker's process. The next subsection discusses the unification of l_r and l_s in the master.

Algorithm 1 Master process

```
{Initialize}
C ← a given ans clause
while C is not grounded do
  {Send a body atom and a term list atom.}
  b ← a body atom in C
  w ← an idle worker
  if b and w exists then
    i ← a new unique identifier
    l ← a list of variables in b
    v_s ← vl(i, l)
    v'_s ← a copy of v_s
    Send b, v'_s to w
    C ← C ⊕ {v_s}.
  end if
  {Receive a response from a worker.}
  while a response from a worker exists do
    vl(i, l_r) ← a response
    vl(i, l_s) ← a vl atom in C with the identifier i
    C ← C ⊖ {vl(i, l_s)}.
    C ← C ⊕ {eqaul(l_s, l_r)}.
    Apply a ETR which solves eqaul(l_s, l_r).
  end while
end while
```

Algorithm 2 Worker process

```
loop
  Receive b, vl(i, l) from Master
  γ ← spec(gen(b))
  Send vl(i, l)γ to Master
end loop
```

D. Equality Constraint in Specialization System

The important part of the master's algorithm is the step which resolve an an equality constraint atom $\mathrm{eqaul}(l_s, l_r)$, i.e. computes θ which unifies l_r and l_s. Since l_r and l_s are lists of terms with equal length, the unification of them is achieved by unification of each element in l_r and l_s, and this unification process of two terms depends on a definition of equality of the terms. We discuss this equality processing based on Specialization System using Constraint Satisfaction Problems (CSP) as examples.

The Constraint Satisfaction Problem (CSP) is a class of problems defined by a set of variables, a domain of values, and a set of constraints. Its solution is a binding for all variables that satisfy the constraints. In ET, a query is defined by an ans clause with a head atom whose arguments are the variables in CSP, and a set of constraints for its body. In SBPP system to solve CSP, a constraint will be sent to a worker, and specialization for variables in the constraint will be sent to the master. Since variable operation, i.e. equality processing in SBPP, is important in solving CSP in SBPP, we illustrate application of Specialization System for CSP.

One example of CSP is the puzzle "OEKAKI-LOGIC (PIC-A-PIX)" [4] (Fig. 3). A cell on the board corresponds to a variable, and its domain of values is {black,white}. A constraint of the puzzle corresponds to one row or column. A list of numbers at end of a row (or column) describes a constraint that each sequence of black cells whose length corresponds each number is located in the row (or column) separated by one or more white cells from its neighbour black sequences. Because a variable in PIC-A-PIX is to be bound to black, white, or another variable, we define equality of the two terms in PIC-A-PIX, i.e. unification of the tow terms in the following three cases:

1) unify a variable and a constant
2) unify a variable and another variable
3) unify a constant and another constant

The final case will be failed if the two constants are different, like black and white. These operation on a variable is equal to LP's binding. Note that if a variable is not bound to black (or white), then it must be bound to white (or black).

Another example is Sudoku [5] (Fig. 4). It resembles PIC-A-PIX, but the domain of values is 1 to 9, and a row (column, or box) is to a constraint. Sudoku's constraint is $\mathrm{alldiff}$ or $\mathrm{alldifferent}$, a constraint that each argument has different value from other variables [6]. In contrast to PIC-A-PIX, though a value of a variable will not be decided by one negative information about the variable (like "a variable X is not 6"), such

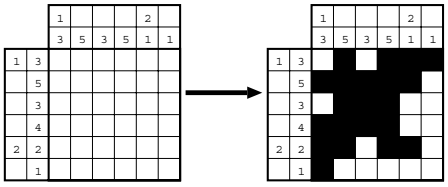

Fig. 3. PIC-A-PIX example

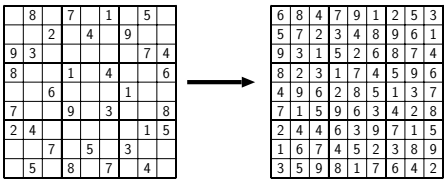

Fig. 4. Sudoku example

negative information plays important role in solving Sudoku. It is efficient and natural to extend a specialization system of PIC-A-PIX by introducing structures into a variable to carry additional information about its value. We call a variable without additional information "pure variable," and a variable with the additional information "candidate set variable." The candidate set variable contains a set of constants that may be a value of the variable, which is usually represented by a "member" predicate:

$$\mathrm{member}(*a, [*a|\_]) \leftarrow .$$
$$\mathrm{member}(*a, [\_| * X]) \leftarrow \mathrm{member}(*a, *X).$$

For example, assume a variable $*x$ will be a value 1, 2, or 3, i.e. "$\mathrm{member}(*x, [1,2,3])$". In a candidate set variable, this information is represented by "$*x : [1,2,3]$".

Equality of two terms in Sudoku is defined by the definition of member. Unification of two terms is categorized int the following 6 cases:

1) unify a pure variable to a constant
2) unify a pure variable to another pure variable
3) unify a constant to another constant
4) unify a pure variable to a candidate set variable
5) unify a candidate set variable to a constant
6) unify a candidate set variable to another candidate set variable

The first three cases are same as PIC-A-PIX. In the 4th case, the pure variable is bound to the candidate set variable. In the 5th case, if the constant is not a member of the candidate set, then this case fails, else the variable is bound to the constant. The final case is equal a set of three predicates. For example, equality of the following two variables

$$*x : [1,2,3]$$
$$*y : [2,3,4]$$

is equal to the equality of $\star\mathtt{x}$ and $\star\mathtt{y}$ with the following two constraints:

$$\mathrm{member}(*x, [1,2,3])$$
$$\mathrm{member}(*y, [2,3,4])$$

If the intersection of the both candidate set is not empty, then the variable are bound by the other unique variable with the intersection as its candidate set. If the intersection is empty, this case fails. In the above example, $*x$ and $*y$ are unified by the following specializer:

$$\{*x : [1,2,3]/*z : [2,3], *y : [2,3,4]/*z : [2,3]\}.$$

E. Example of SBPP

We explain the method using $\mathrm{alldiff}$ with 4 arguments. Assume that the following ans clause which has two $\mathrm{alldiff}$ atoms in its body exists in the master:

$$\begin{aligned}&\mathrm{ans}(*x : [1,2,3,4], *y : [1,2,3,4], *z : [1,2,3,4])\\ &\quad \Leftarrow \quad \mathrm{alldiff}(*x : [1,2,3,4], 3, *y : [1,2,3,4], 1),\\ &\qquad\quad \mathrm{alldiff}(*x : [1,2,3,4], 2, 1, *z : [1,2,3,4]).\end{aligned}$$

1) Assume the master decides to send the first $alldiff$ atom to a worker with a term list atom. Let be generated by the master. The master generates a term list atom with the unique identifier i_1, and adds it in the ans clause:
$$\begin{aligned}&\mathrm{ans}(*x : [1,2,3,4], *y : [1,2,3,4], *z : [1,2,3,4])\\ &\quad \Leftarrow \quad \mathrm{alldiff}(*x : [1,2,3,4], 3, *y : [1,2,3,4], 1),\\ &\qquad\quad \mathrm{vl}(i_1, [*x : [1,2,3,4], *y : [1,2,3,4]]),\\ &\qquad\quad \mathrm{alldiff}(*x : [1,2,3,4], 2, 1, *z : [1,2,3,4]).\end{aligned}$$
The master copies the $\mathrm{alldiff}$ atom and the term list atom as follows:
$$\begin{aligned}[\mathtt{M} \rightarrow \mathtt{W}]: \quad &\mathrm{alldiff}(*y : [1,2,3,4], 3, *t : [1,2,3,4], 1),\\ &\mathrm{vl}(i_1, [*s : [1,2,3,4], *t : [1,2,3,4]])\end{aligned}$$
and sends it a worker.
2) Before receiving a response from the worker, the ans clause would be specialized by other rules, or responses from other workers, e.g. the master at t_2 in Fig. 2. Assume the following specialization γ is applied:
$$\gamma: \quad \{*x : [1,2,3,4]/*u : [3,4]\}$$
Then the ans clause becomes as follows:
$$\begin{aligned}&\mathrm{ans}(*u : [3,4], *y : [1,2,3,4], *z : [1,2,3,4])\\ &\quad \Leftarrow \quad \mathrm{alldiff}(*u : [3,4], 3, *y : [1,2,3,4], 1),\\ &\qquad\quad \mathrm{vl}(i_1, [*u : [3,4], *y : [1,2,3,4]]),\\ &\qquad\quad \mathrm{alldiff}(*u : [3,4], 2, 1, *z : [1,2,3,4]).\end{aligned}$$
3) Assume the master receives the following response from the worker:
$$[\mathtt{W} \rightarrow \mathtt{M}]: \quad \mathrm{vl}(i_1, [*v : [2,4], *w : [2,4]])$$
Then the master retrieves the following variable list atom which has the same identifier i_1;
$$\mathrm{vl}(i_1, [*u : [3,4], *y : [1,2,3,4]])$$
and removes this from the ans clause. To unify the two term lists, the master adds an equality constraint atom to the ans clause:
$$\begin{aligned}&\mathrm{ans}(*u : [3,4], *y : [1,2,3,4], *z : [1,2,3,4])\\ &\quad \Leftarrow \quad \mathrm{alldiff}(*u : [3,4], 3, *y : [1,2,3,4], 1),\\ &\qquad\quad \mathrm{eqaul}([*u : [3,4], *y : [1,2,3,4]],\\ &\qquad\qquad\qquad [*v : [2,4], *w : [2,4]]),\\ &\qquad\quad \mathrm{alldiff}(*u : [3,4], 2, 1, *z : [1,2,3,4]).\end{aligned}$$
4) To solve the equality of the two lists, i.e. the equality of $*v$ and $*u$, and of $*w$ and $*y$, the master applies ETRs, and specializes the ans clause by θ as the follows:

TABLE I
EXECUTION TIME OF SBPP SYSTEM IN SUDOKU SOLVING

Number of Workers	Mean (sec)	(SD)
1	25.255	(0.276)
2	24.303	(0.832)
3	22.829	(1.001)
4	22.279	(0.390)
5	23.577	(0.711)

θ: $\{*v : [2,4]/*m : [4], *u : [3,4]/*m : [4],$
$*w : [2,4]/*n : [2,4], *y : [1,2,3,4]/*n : [2,4]\}$
By this application, the ans clause becomes as follows:
$\mathrm{ans}(*m : [4], *n : [2,4], *z : [1,2,3,4])$
$\Leftarrow \quad \mathrm{alldiff}(*m : [4], 3, *n : [2,4], 1),$
$\mathrm{alldiff}(*m : [4], 2, 1, *z : [1,2,3,4]).$

F. Performance

The system is divided into the master and worker subsystems. Each system has two modules: the domain dependent module and the domain independent module. The domain independent module supports communication, process control and the domain dependent module management. Another important role of this module in the master is management of task allocation to each worker. The domain dependent module supports domain knowledge, i.e. its specialization system, ETR, gen and spec for each domain. Another important role of the master's module is to select atoms to be sent to a worker and its timing. The system is implemented in ETI, ETR interpreter. It uses OpenMPI library for process controls and communication between the master and the workers.

The performance of the system in solving a Sudoku problem (25×25 cells on the board) by two 4-core CPUs (Xeon W5590) is show in Table I. The problem is solved in one master with 1 to 5 workers. We measured execution time 10 times for each condition. The effect of parallel processing appears in 1 to 4 workers condition, though the effect seems to stop in 5 workers condition.

IV. CONCLUSION

In this paper, we discussed the parallel processing framework based on distributed computation of the specialization. We proposed the new implementation of SBPP. Our approach based on ETF has some similarities with parallel logic programming languages [7], [8], [9]. These languages are mainly based on AND/OR parallelism because the problems and computations in LP is represented by multiple independent goals and candidate clauses.

Though SBPP resembles AND parallelism, SBPP processes specialization information in parallel, and the use of the specialization in our parallel processing framework gives the advantage by extending the limitation of the binding in LP. The rich representation allowed by the specialization provides efficient communications between the master and the workers. These features of SBPP are appropriate to solving constraint satisfaction problems, which involves massive computations of variable bindings. SBPP's performance is presented using Sudoku, a CSP problem in which the specialization is more helpful than substitution in LP.

The advantage of using ETF as our framework for parallel processing programming is that it provides the base of the program generation from a problem definition while guaranteeing correctness. In the many parallel logic programming languages, the executable codes of parallel processing systems are to be written by programmers. The languages adopt the extra features for the programmers to control parallel processes easily, e.g. the guard part in LP to simplify OR-parallelism, the binding limitation of variables in the guard part, and the mode declarations for specification of input and output. Though these features resolve some complications in parallel processing programming, they may complicates examinations of the correctness of the programs, because they are not included in first-order logic that is the base of LP's correctness. Based on the correctness of SBPP with Memo-tree presented by Akama et al., we will construct a theory of the SBPP without Memo-tree.

REFERENCES

[1] K. Akama, E. Nantajeewarawat, and H. Ogasawara, "Generation of correct parallel programs based on specializer generation transformations," in *Proceedings of the 7th international conference on intelligent technologies*, 2006.
[2] H. Ogasawara, K. Akama, and H. Mabuchi, "Parallel processing framework based on distributed computation of specialization," *International Journal of Innovative Computing, Information and Control*, vol. 6, no. 5, pp. 2371–2381, 2010.
[3] K. Akama and E. Nantajeewarawat, "Formalization of the equivalent transformation computation models," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 10, no. 3, pp. 245–259, 2006.
[4] Conceptis-Limited. (2005) Pic-a-pix help. [Online]. Available: http://www.conceptispuzzles.com/online/pap/help.htm
[5] Nikoli. (2010) Sudoku outline. [Online]. Available: http://www.nikoli.co.jp/en/puzzles/sudoku/index_text.htm
[6] W.-J. van Hoeve and I. Katriel, "Global constraints," in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. ELSEVIER, 2006, ch. 6, pp. 169–208.
[7] J. C. de Kergommeaux, "Parallel logic programming systems," *ACM Computing Surveys*, vol. 26, no. 3, 1994.
[8] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey," *Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, 2001.
[9] B. Ramkumar and L. V. Kalé, "Machine independent and and or parallel execution of logic programs: Part i-the binding environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 170–180, 1994.

Hidemi Ogasawara: School of Information Science and Technology, Chukyo University, Toyota Aichi 470-0393, Japan.
e-mail: hidemi@sist.chukyo-u.ac.jp

Kiyoshi Akama: Information Initiative Center,Hokkaido University, Sapporo 060-0811, Japan, e-mail: akama@iic.hokodai.ac.jp

Hiroshi Mabuchi: Faculty of Software and Information Science, Iwate Prefectural University, Takizawa, Iwate 020-0193, Japan
e-mail: mabu@soft.iwate-pu.ac.jp