Constructing Parallel Programs Based on Rule Generators

Kiyoshi Akama
Information Initiative Center
Hokkaido University
Hokkaido, Japan
Email: akama@iic.hokudai.ac.jp

Ekawit Nantajeewarawat
Computer Science Program
Sirindhorn International Institute of Technology
Thammasat University
Pathumthani, Thailand
Email: ekawit@siit.tu.ac.th

Hidekatsu Koike
Faculty of Social Information
Sapporo Gakuin University
Hokkaido, Japan
Email: koike@sgu.ac.jp

***Abstract*—We propose a new architecture of parallel programs based on the master-worker model of parallel computing. In this architecture, computation is realized by rule application and rule generation. A master has a set of equivalent transformation rules (ET rules) and solves a problem by successively applying ET rules to definite clauses representing its computation state. A worker has a rule generator and makes computation by generating ET rules on demand based on run-time content of the master's computation states. A general scheme for constructing parallel programs based on rule-set generators and rule-generator generators is presented and a sufficient condition for the correctness of the scheme is established. Application of our framework to solving a constraint satisfaction problem is illustrated.**

***Keywords*-parallel computation; program correctness; rule generation; equivalent transformation**

I. Introduction

Constructing a correct parallel program that makes effective use of computing resources in a distributed environment is a nontrivial task. Major difficulties include how to strictly ensure the correctness of computation and how to obtain substantial efficiency gains, in particular under situations when the response time of distributed processes varies and is often unpredictable. Such situations happen commonly in distributed computing environments, owing not only to a large variety of possibly distributed tasks but also to other factors such as availability of computing resources and stability of communication channels, etc.

A. The Proposed Parallel Program Architecture

We propose a new architecture of parallel programs, where problem solving is carried out through rule application and rule generation. The architecture is based on the master-worker (or parent-child) model of parallel computing, where one process, referred to as a master (or a parent), solves a problem by distributing some tasks to other processes, referred to as workers (or children). Assume that a specification S is given. The master has a set of equivalent transformation rules (ET rules) with respect to S and its computation state is represented as a set of definite clauses. The master makes computation by successively applying ET rules to clauses in its state. A worker has a rule generator and makes computation by generating ET rules with respect to S. Having an initial set of ET rules, the master (i) selects an ET rule from its rule set and applies the rule, or (ii) sends an atom set to a worker as a request for rule generation, or (iii) receives an ET rule from a worker and adds it to the rule set. Based on the given information from the master (a selected atom set), a worker generates an ET rule using its rule generator, and returns the rule to the master.

B. Effectiveness of the Proposed Architecture

Using the proposed architecture, the correctness of computation can be guaranteed by combination of correct rules and correct rule generators. From atom sets observed at run-time, a master requests its workers to generate specialized rules on demand. Using specialized rules that are tailored to run-time content of computation states, substantial efficiency improvement of computation in the master process can be achieved, i.e., transformation steps can be reduced and computational explosion can be suppressed. It is difficult to predict which specialized rules should be generated beforehand, and generating all specialized rules in advance at compile time is impractical because there are usually far too many possible states and there are usually many possible specialized rules for each state, only a small part of which is really used. Distributing run-time rule generation to workers releases the master from the task of generating specialized rules, which can take much time even when specific patterns of target atoms are already determined.

The initial rule set of a master is prepared in such a way that it is sufficient for solving a problem, albeit not efficiently, without additional rules obtained from workers. As a consequence, delayed response of workers and communication failure do not affect the completion of a problem solving process. Rules obtained form workers contribute to computation speedup, rather than completion of computation. Since rules have a precise procedural semantics [1], the use of rules as messages returned from workers makes clear the meanings of the messages. A returned rule can be used any time no matter how the state of the master is changed during the course of master-worker interaction. No adjustment of returned rules is required even when the order of rule-generation requests and the order of the arrival of their corresponding returned rules do not coincide.

C. Paper Organization and Notation

The paper progresses from here as follows: Section II establishes a class of specifications considered herein. Section III describes parallel programs in our proposed framework. Section IV presents a scheme for construction of parallel programs based on rule-set generators and rule-generator generators, along with its correctness theorem. Section V provides methods for constructing rule-set generators and rule-generator generators. Section VI illustrates application of our framework to solving a constraint satisfaction problem. Section VII concludes the paper.

The following notation holds thereafter. For any set A, $\mathit{pow}(A)$ denotes the power set of A. For any sets A and B, $\mathit{PartialMap}(A, B)$ denotes the set of all partial mappings from A to B.

II. SPECIFICATIONS

A specification provides background knowledge in a problem domain and defines a set of queries of interest. A specification considered in this paper is formulated using the concepts recalled below.

Assume that an alphabet Δ for first-order logic is given. Let $\mathcal{A}$ and $\mathcal{G}$ be the set of all first-order atomic formulas (atoms) and that of all ground atoms, respectively, on Δ. Let $\mathcal{S}$ denote the set of all substitutions on Δ. A *definite clause* C on Δ is an expression of the form $a \leftarrow \mathit{Bs}$, where $a \in \mathcal{A}$ and Bs is a (possibly empty) finite subset of $\mathcal{A}$. The set $\{a\} \cup \mathit{Bs}$ is denoted by $\mathit{atoms}(C)$; a is called the *head* of C, denoted by $\mathit{head}(C)$; Bs is called the *body* of C, denoted by $\mathit{body}(C)$; and each element of $\mathit{body}(C)$ is called a *body atom* of C. When $\mathit{body}(C) = \varnothing$, C is called a *unit clause*. The set notation is used in the right-hand side of C so as to stress that the order of atoms in $\mathit{body}(C)$ is not important. However, for the sake of simplicity, set braces enclosing body atoms are often omitted; e.g., a definite clause $a \leftarrow \{b_1, \ldots, b_n\}$ is often written as $a \leftarrow b_1, \ldots, b_n$.

A *declarative description* on Δ is a set of definite clauses.[1] The meaning of a declarative description is defined as follows: Given a definite clause C and a set $G \subseteq \mathcal{G}$, let

$$T(C, G) = \{\mathit{head}(C\theta) \mid (\theta \in \mathcal{S})\ \&\ (\mathit{atoms}(C\theta) \subseteq \mathcal{G})\ \&\ (\mathit{body}(C\theta) \subseteq G)\}.$$

Let D be a declarative description. A mapping T_D on $\mathit{pow}(\mathcal{G})$ is defined by $T_D(G) = \bigcup_{C \in D} T(C, G)$ for any $G \subseteq \mathcal{G}$. The meaning of D, denoted by $\mathcal{M}(D)$, is then defined as the set $\bigcup_{n=1}^{\infty} T_D^n(\varnothing)$, where $T_D^1(\varnothing) = T_D(\varnothing)$ and $T_D^n(\varnothing) = T_D(T_D^{n-1}(\varnothing))$ for each $n > 1$.

[1]We call a set of definite clauses a *declarative description* in order to emphasize that no procedural meaning of it is considered.

A *specification* on Δ is a pair $S = \langle D, Q \rangle$, where D is a declarative description, representing background knowledge, and Q is set of atoms, representing queries. For any $q \in Q$, the answer to q with respect to S is defined as the set

$$\mathcal{M}(D) \cap \mathit{rep}(q),$$

where for any atom a, $\mathit{rep}(a)$ is the set of all ground instances of a. Let SPEC be the set of all such specifications.

III. PARALLEL PROGRAMS

A. Rules and Rule Generators

A *transformation rule* (for short, *rule*) is a relation on definite-clause sets. A rule r transforms a set Cs of definite clauses into another set Cs' of definite clauses if $\langle \mathit{Cs}, \mathit{Cs}' \rangle \in r$. Let RULE be the set of all such rules.

Assume that a set R of rules is given. To compute the answer to a query q using R, a singleton definite-clause set

$$\mathit{Cs}_0 = \{\phi(q) \leftarrow q\}$$

is constructed, where ϕ is a bijective mapping that associates with each atom $a \in \mathcal{A}$ an atom obtained from a by replacing its predicate symbol with a new predicate symbol. Then a transformation sequence

$$[\mathit{Cs}_0, \mathit{Cs}_1, \ldots, \mathit{Cs}_m]$$

is constructed such that (i) for each i, $\langle \mathit{Cs}_i, \mathit{Cs}_{i+1} \rangle \in r$ for some rule $r \in R$, and (ii) Cs_m is a set of unit clauses.

A *rule generator* is a partial mapping from $\mathit{pow}(\mathcal{A})$ to RULE. When an atom set $A \subseteq \mathcal{A}$ is given as input, a rule generator yields a rule for transforming some definite clauses whose bodies contain instances of A.

B. Parallel Programs

Assume that

- a master has n workers $w_1, \ldots, w_n$,
- R_0 is a set of rules, and
- for each $i \in \{1, \ldots, n\}$, gen_i is a rule generator.

The pair $\langle R_0, [\mathit{gen}_1, \ldots, \mathit{gen}_n] \rangle$ determines a parallel procedure, consisting of $n + 1$ processes, which is described below. Assume that an input query $q \in \mathcal{A}$ is given.

- *The Master Process:* The master has a state $\langle \mathit{Cs}, R \rangle$, where Cs is a set of definite clauses and R is a set of rules, and it works as follows:
  - Initially, the master sets
    - $\mathit{Cs} = \{(\phi(q) \leftarrow q)\}$, and
    - $R = R_0$.
  - If Cs contains some non-unit clause, then the master performs one of the following operations nondeterministically whenever possible:
    - Select a non-unit clause C from Cs, select a rule from R, and update Cs by applying the selected rule to C.

  - $*$ Select an atom set A from the body of one clause in *Cs* and send it to a worker.
  - $*$ Receive a rule r from a worker and add r to R.
- If *Cs* contains only unit clauses, then the master outputs the set

$$\bigcup\{\mathit{rep}(\phi^{-1}(a)) \mid (a \leftarrow) \in \mathit{Cs}\}$$

as the computed answer.

- *A Worker Process:* A worker w_i has a rule generator gen_i and has a buffer storing atom sets received from the master. At any time, it performs the following operations sequentially:
  1) Select one atom set A from the buffer.
  2) Generate a rule $r = \mathit{gen}_i(A)$.
  3) Return r to the master.

With a design of more detailed control mechanism, the procedure thus obtained is converted into a program in a lower-level language, making optimization of state representation and memory usage. The lower-level implementation part is however outside the scope of this paper, and the pair $\langle R_0, [\mathit{gen}_1, \ldots, \mathit{gen}_n]\rangle$ is also regarded as a *parallel program*.

IV. A Parallel-Program Construction Scheme

A scheme for constructing parallel programs using rule-set generators and rule-generator generators is next described.

A. Rule-Set Generators and Rule-Generator Generators

First, a rule-set generator and a rule-generator generator are introduced:

- A *rule-set generator* generates a set of rules from a given specification. It is formalized as a mapping from Spec to $\mathit{pow}(\text{Rule})$.
- A *rule-generator generator* generates from an input specification a rule generator (which is a partial mapping from $\mathit{pow}(\mathcal{A})$ to Rule). It is formalized as a mapping from Spec to $\mathit{PartialMap}(\mathit{pow}(\mathcal{A}), \text{Rule})$.

Based on the concept of an equivalent transformation rule, the correctness of a rule-set generator and that of a rule-generator generator are defined:

- A rule r is an *equivalent transformation rule* (*ET rule*) with respect to a declarative description D iff for any $\langle \mathit{Cs}, \mathit{Cs}'\rangle \in r$, $\mathcal{M}(D \cup \mathit{Cs}) = \mathcal{M}(D \cup \mathit{Cs}')$.
- A rule-set generator *RSG* is *correct* iff for any specification $S = \langle D, Q\rangle \in$ Spec, every rule in $\mathit{RSG}(S)$ is an ET rule with respect to D.
- A rule-generator generator *RGG* is *correct* iff for any specification $S = \langle D, Q\rangle \in$ Spec and any $A \subseteq \mathcal{A}$, if $\mathit{RGG}(S)(A)$ is defined, then it is an ET rule with respect to D.

B. A General Parallel-Program Construction Scheme

Construction of a correct rule-set generator and that of a correct rule-generator generator provide a general groundwork for constructing correct parallel programs from specifications, using the following parallel-program construction scheme.

1) Construct a correct rule-set generator *RSG*.
2) Construct correct rule-generator generators RGG_1, $\ldots, \mathit{RGG}_n$.
3) From a given a specification $S \in$ Spec, construct a parallel program $\langle R_0, [\mathit{gen}_1, \ldots, \mathit{gen}_n]\rangle$ as follows:
   - $R_0 = \mathit{RSG}(S)$.
   - For each $i \in \{1, \ldots, n\}$, $\mathit{gen}_i = \mathit{RGG}_i(S)$.

Even when only one rule-generator generator, say *RGG*, is constructed at Step 2, i.e., $\mathit{RGG}_1 = \cdots = \mathit{RGG}_n = \mathit{RGG}$, n workers can still be useful for parallel processing since there are many possible different atom sets to be distributed to workers. Given a specification S and different atom sets A_1, $\ldots, A_n$, workers with the same rule generator $\mathit{RGG}(S)$ may produce different ET rules $\mathit{RGG}(S)(A_1), \ldots, \mathit{RGG}(S)(A_n)$.

As a larger number of mutually independent rule-generator generators are available, a larger number of effective workers can be used, potentially yielding more efficient parallel computation. The number of effective workers is evaluated by the multiplication of (i) the number of mutually independent rule-generator generators and (ii) the number of atom sets to be sent to workers.

It is shown in [4] that the correctness of a rule-set generator and that of a rule-generator generator together provide a sufficient condition for a guarantee of the correctness of a resulting parallel program. More precisely:

Theorem 1: Suppose that *RSG* is a correct rule-set generator and $\mathit{RGG}_1, \ldots, \mathit{RGG}_n$ are correct rule-generator generators. Then for any $S \in$ Spec, the parallel program

$$\langle \mathit{RSG}(S), [\mathit{RGG}_1(S), \ldots, \mathit{RGG}_n(S)]\rangle$$

is correct with respect to S.

V. Constructing Rule-Set Generators and Rule-Generator Generators

There are many possible correct rule-set generators and many possible correct rule-generator generators. A large variety of correct parallel programs can thus be constructed using the scheme of Section IV. At present, several methods exist for constructing correct rule-set generators and correct rule-generator generators, some of which are described in this section.

A. Meta-Computation-Based Generators

Meta-computation [2], [3] is a general purpose method for generating ET rules from a specification. The method takes a declarative description D and an atom set $A \subseteq \mathcal{A}$ as input, and produces a nonempty output set of ET rules with respect to D for transforming some definite clauses whose bodies contain instances of A. When A is a singleton set containing the most general atom with respect to some predicate p,[2] the output rule set includes the general unfolding rule for p with respect to D.

Using meta-computation, a rule-set generator *RSG* is constructed as follows: For any $S = \langle D, Q\rangle \in$ SPEC,

- generate a set $\{A_1, \ldots, A_q\}$ of atom sets from Q,
- for each $i \in \{1, \ldots, q\}$, construct a rule set R_i from D and A_i by meta-computation, and
- produce $RSG(S) = R_1 \cup \cdots \cup R_q$.

Similarly, using meta-computation, a rule-generator generator *RGG* is constructed as follows: For any $S = \langle D, Q\rangle \in$ SPEC and any atom set $A \subseteq \mathcal{A}$,

- construct a rule set R from D and A by meta-computation,
- select a rule r from R, and
- produce $RGG(S)(A) = r$.

B. Rule Generation Based on Common Specializers

Let $S = \langle D, Q\rangle \in$ SPEC and a singleton atom set $\{a\} \subseteq \mathcal{A}$ be given. Assume that $v_1, \ldots, v_q$ are all the variables that occur in a. A rule can be generated from S and $\{a\}$ based on common specializers as follows:

- Calculate the set $G = \{a\theta \mid (\theta \in \mathcal{S})\ \&\ (a\theta \in \mathcal{M}(D))\}$.
- For each $i \in \{1, \ldots, q\}$, find a common ground term for v_i with respect to G, i.e., find a ground term t_i such that for any $\rho \in \mathcal{S}$, if $a\rho \in G$, then $v_i\rho = t_i$.[3]
- Let E be the sequence of all equality atoms $=(v_i, t_i)$ such that t_i is a common ground term for v_i with respect to G.
- Assuming that the sequence E thus obtained is $[e_1, \ldots, e_{q'}]$, where $q' \leq q$, construct a rule

$$a \Rightarrow \{e_1, \ldots, e_{q'}\}, a.$$

This rule is applicable to a definite clause C if there exist $\theta \in \mathcal{S}$ and an atom $b \in body(C)$ such that θ contains only bindings for variables occurring in a and $a\theta = b$. When applied, the rule specializes C by the evaluation of the equality atoms $e_1\theta, \ldots, e_{q'}\theta$.[4]

VI. AN EXAMPLE

To illustrate application of our framework, a Pic-a-Pix puzzle[5] (Oekaki Logic or Paint by Numbers) of a fixed size $m \times n$ is used as an example problem.

[2]Given an m-ary predicate p, the most general atom with respect to p is $p(v_1, \ldots, v_m)$, where the v_i are mutually different variables.

[3]A common ground term for v_i with respect to G is unique if it exists.

[4]When $q' = 0$, the rule does not make clause specialization and it is not used in a master process.

[5]http://www.conceptispuzzles.com.

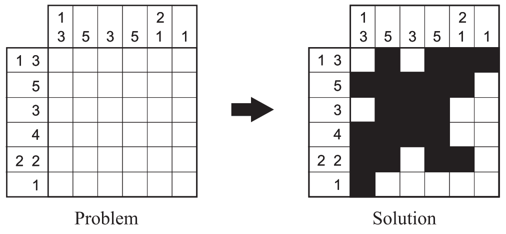

Figure 1. A Pic-a-Pix puzzle.

C_1: $pat([], *Z) \leftarrow zeros(*Z)$.
C_2: $pat([*a|*X], *Y) \leftarrow zeros(*Z), ones(*a, *A), pat(*X, *B), apps([*Z, *A, *B], *Y)$.
C_3: $zeros([0]) \leftarrow$.
C_4: $zeros([0|*X]) \leftarrow zeros(*X)$.
C_5: $ones(0, []) \leftarrow$.
C_6: $ones(*n, [1|*Y]) \leftarrow >(*n, 0), subtr(*n, 1, *m), ones(*m, *Y)$.
C_7: $apps([], []) \leftarrow$.
C_8: $apps([*a|*X], *Y) \leftarrow apps(*X, *Z), app(*a, *Z, *Y)$.
C_9: $app([], *Y, *Y) \leftarrow$.
C_{10}: $app([*a|*X], *Y, [*a|*Z]) \leftarrow app(*X, *Y, *Z)$.

Figure 2. Representing background knowledge for Pic-a-Pix puzzles.

A. Problem Representation and Formulating a Specification

First, consider the Pic-a-Pix puzzle of size 6×6 in Fig. 1. It consists of a blank grid and clues, i.e., block patterns, on the left of every row and on the top of every column, with the goal of painting blocks in each row and column so that their length and order correspond to the patterns and there is at least one empty square between adjacent blocks.

The puzzle in Fig. 1 is represented by a *Pic-a-Pix*-atom

$$\begin{aligned}&Pic\text{-}a\text{-}Pix([[1,3],[5],[3],[4],[2,2],[1]],\\ &\quad [[1,3],[5],[3],[5],[2,1],[1]],\\ &\quad *mat),\end{aligned}$$

where $*mat$ is a variable. Other 6×6 Pic-a-Pix puzzles are represented as *Pic-a-Pix*-atoms in a similar way. Let $Q_{6\times6}$ be the set of all such *Pic-a-Pix*-atoms. The specification for this class of puzzles, denoted by $S_{6\times6}$, is defined by

$$S_{6\times6} = \langle\{C_0\} \cup D_{\mathrm{LINK}} \cup D_{\mathrm{PIC}} \cup D_{\mathrm{BLT}}, Q_{6\times6}\rangle,$$

where C_0 is the definite clause

$$\begin{aligned}C_0:\ & Pic\text{-}a\text{-}Pix(*t, *y, *mat) \leftarrow\\ &\quad =(*t, [*t1, *t2, *t3, *t4, *t5, *t6]),\\ &\quad =(*y, [*y1, *y2, *y3, *y4, *y5, *y6]),\\ &\quad matrix(*t, *y, *mat), trans(*mat, *tam),\\ &\quad pairing(*t, *mat), pairing(*y, *tam),\end{aligned}$$

D_{LINK} is a declarative description that provides the definitions of *matrix*, *trans*, and *pairing*, D_{PIC} consists of the definite clauses C_1–C_{10} in Fig. 2, and D_{BLT} provides the definitions of built-in predicates, such as $>$ and *subtr*.

$$
\begin{array}{ll}
C_0'\colon & \mathit{Pic\text{-}a\text{-}Pix}([*t1,*t2,*t3,*t4,*t5,*t6], \\
& \quad [*y1,*y2,*y3,*y4,*y5,*y6], \\
& \quad [[*a11,*a12,*a13,*a14,*a15,*a16], \\
& \quad\ [*a21,*a22,*a23,*a24,*a25,*a26], \\
& \quad\ [*a31,*a32,*a33,*a34,*a35,*a36], \\
& \quad\ [*a41,*a42,*a43,*a44,*a45,*a46], \\
& \quad\ [*a51,*a52,*a53,*a54,*a55,*a56], \\
& \quad\ [*a61,*a62,*a63,*a64,*a65,*a66]]) \\
& \leftarrow \\
& \mathit{pat}(*t1,[0,*a11,*a12,*a13,*a14,*a15,*a16,0]), \\
& \mathit{pat}(*t2,[0,*a21,*a22,*a23,*a24,*a25,*a26,0]), \\
& \mathit{pat}(*t3,[0,*a31,*a32,*a33,*a34,*a35,*a36,0]), \\
& \mathit{pat}(*t4,[0,*a41,*a42,*a43,*a44,*a45,*a46,0]), \\
& \mathit{pat}(*t5,[0,*a51,*a52,*a53,*a54,*a55,*a56,0]), \\
& \mathit{pat}(*t6,[0,*a61,*a62,*a63,*a64,*a65,*a66,0]), \\
& \mathit{pat}(*y1,[0,*a11,*a21,*a31,*a41,*a51,*a61,0]), \\
& \mathit{pat}(*y2,[0,*a12,*a22,*a32,*a42,*a52,*a62,0]), \\
& \mathit{pat}(*y3,[0,*a13,*a23,*a33,*a43,*a53,*a63,0]), \\
& \mathit{pat}(*y4,[0,*a14,*a24,*a34,*a44,*a54,*a64,0]), \\
& \mathit{pat}(*y5,[0,*a15,*a25,*a35,*a45,*a55,*a65,0]), \\
& \mathit{pat}(*y6,[0,*a16,*a26,*a36,*a46,*a56,*a66,0]).
\end{array}
$$

Figure 3. A clause obtained by unfolding C_0.

B. Determining the Forms of Exchange Information

The clause C_0 is unfolded using D_{LINK}, resulting in the clause C_0' in Fig. 3, which contains $6+6$ *pat*-atoms in its body. A copy of each of these *pat*-atoms is then added to the body of C_0'. The predicate *pat:c* is used for denoting a copy.[6] The clause thus obtained is

$$\hat{C}_0' : \quad \mathit{head}(C_0') \leftarrow \mathit{body}(C_0') \cup \mathit{copy}(\mathit{body}(C_0')),$$

where $\mathit{copy}(\mathit{body}(C_0'))$ is the set consisting of the 6+6 added *pat:c*-atoms. These *pat:c*-atoms are designated as messages that a master sends to workers. The original specification $S_{6\times 6}$ is then transformed into the specification

$$S_{6\times 6}' = \langle \{\hat{C}_0', C_1', C_2'\} \cup D_{\text{PIC}} \cup D_{\text{BLT}}), Q_{6\times 6}\rangle,$$

where C_1' and C_2' are the definite clauses obtained from C_1 and C_2, respectively, by replacing the predicate *pat* with the predicate *pat:c*.

C. Constructing a Rule Set Using Rule-Set Generators

Using a meta-computation-based rule-set generator, all rules in Fig. 4 except r_6 and also all rules in Fig. 5 are generated. The rule r_6 can be generated using another rule-set generator, based on a method of finding a common specialization from definite clauses. An unfolding rule corresponding to $\hat{C}_0'$ is also generated using a meta-computation-based rule-set generator.

The rules r_1–r_{17} are specialized rules; they are applicable to clauses whose bodies contain atoms having certain specific patterns. For example, r_1 (respectively, r_2) is applicable to any clause containing in its body a *pat*-atom the first argument of which is an empty (respectively, nonempty)

[6]For example, the copy of the first body atom of C_0' is $\mathit{pat{:}c}(*t1, [0, *a11, *a12, *a13, *a14, *a15, *a16, 0])$.

$$
\begin{array}{ll}
r_1\colon & \mathit{pat}([\,], *Z) \Rightarrow \mathit{zeros}(*Z). \\
r_2\colon & \mathit{pat}([*a|*X], *Y) \\
& \Rightarrow \mathit{zeros}(*Z), \mathit{ones}(*a, *A), \mathit{pat}(*X, *B), \mathit{apps}([*Z, *A, *B], *Y). \\
r_3\colon & \mathit{zeros}([\,]) \Rightarrow \{\mathit{false}\}. \\
r_4\colon & \mathit{zeros}([*a]) \Rightarrow \{=(*a, 0)\}. \\
r_5\colon & \mathit{zeros}([*a, *b|*X]) \Rightarrow \{=(*a, 0)\}, \mathit{zeros}([*b|*X]). \\
r_6\colon & \mathit{zeros}(*X), \{\mathit{pvar}(*X)\} \Rightarrow \{=(*X, [0|*Y])\}, \mathit{zeros}(*X). \\
r_7\colon & \mathit{ones}(0, *Y) \Rightarrow \{=(*Y, [\,])\}. \\
r_8\colon & \mathit{ones}(*n, *X), \{>(*n, 0)\} \\
& \Rightarrow \{=(*X, [1|*Y]), \mathit{subtr}(*n, 1, *m)\}, \mathit{ones}(*m, *Y). \\
r_9\colon & \mathit{apps}([\,], *Y) \Rightarrow \{=(*Y, [\,])\}. \\
r_{10}\colon & \mathit{apps}([*a|*X], *Y) \Rightarrow \mathit{apps}(*X, *Z), \mathit{app}(*a, *Z, *Y). \\
r_{11}\colon & \mathit{app}(*X, *Y, [\,]) \Rightarrow \{=(*X, [\,]), =(*Y, [\,])\}. \\
r_{12}\colon & \mathit{app}(*X, [\,], *Z) \Rightarrow \{=(*X, *Z)\}. \\
r_{13}\colon & \mathit{app}([\,], *Y, *Z) \Rightarrow \{=(*Y, *Z)\}. \\
r_{14}\colon & \mathit{app}([*a|*X], *Y, *Z) \Rightarrow \{=(*Z, [*a|*Z1])\}, \mathit{app}(*X, *Y, *Z1). \\
r_{15}\colon & \mathit{app}(*X, [*a|*Y], [*b|*Z]), \{\mathit{neq}(*a, *b)\} \\
& \Rightarrow \{=(*X, [*b|*X1])\}, \mathit{app}(*X1, [*a|*Y], *Z). \\
r_{16}\colon & \mathit{app}(*X, [*a1, *a2|*Y], [*b1, *b2|*Z]), \{\mathit{neq}(*a2, *b2)\} \\
& \Rightarrow \{=(*X, [*b1|*X1])\}, \mathit{app}(*X1, [*a1, *a2|*Y], [*b2|*Z]). \\
r_{17}\colon & \mathit{zeros}([*a|*X]), \mathit{app}(*X, [1|*M], [1|*R]) \\
& \Rightarrow \{=(*a, 0), =(*X, [\,]), =(*M, *R)\}. \\
r_{18}\colon & \mathit{app}(*X, *Y, *Z) \\
& \Rightarrow \{=(*X, [\,]), =(*Y, *Z)\}; \\
& \Rightarrow \{=(*X, [*a|*X1]), =(*Z, [*a|*Z1])\}, \mathit{app}(*X1, *Y, *Z1).
\end{array}
$$

Figure 4. Rules for solving Pic-a-Pix puzzles.

$$
\begin{array}{ll}
r_{19}\colon & \mathit{pat{:}c}(*R, [0, 0|*S]) \Rightarrow \mathit{pat{:}c}(*R, [0|*S]). \\
r_{20}\colon & \mathit{pat{:}c}([\,], [0, 1|*S]) \Rightarrow \{false\}. \\
r_{21}\colon & \mathit{pat{:}c}([\,], [0]) \Rightarrow. \\
r_{22}\colon & \mathit{pat{:}c}([*n|*R], [0, 1, 0|*S]), \{>(*n, 1)\} \Rightarrow \{false\}. \\
r_{23}\colon & \mathit{pat{:}c}([*n|*R], [0, 1, 1|*S]), \{>(*n, 1)\} \\
& \Rightarrow \{\mathit{subtr}(*n, 1, *m)\}, \mathit{pat{:}c}([*m|*R], [0, 1|*S]). \\
r_{24}\colon & \mathit{pat{:}c}([1|*R], [0, 1, *y|*S]) \Rightarrow \{=(*y, 0)\}, \mathit{pat{:}c}(*R, [0|*S]).
\end{array}
$$

Figure 5. ET rules for removing useless parts of *pat:c*-atoms.

list. Both r_1 and r_2 have no execution part—they make transformation merely by replacement of body atoms. The rules r_3–r_6 are specialized rules for *zeros*-atoms. Since the evaluation of the atom *false* fails, r_3 always makes clause removal when it is applied. Each of r_5 and r_6 contains both an execution part and a replacement part. By using a *pvar*-atom to constrain its applicability, r_6 is only applicable to a *zeros*-atom whose argument is a variable. The rule r_{18} is a general rule; it is applicable to any clause whose body contains any arbitrary *app*-atom. Since r_{18} has two bodies in its right side, its application typically splits a clause into two clauses.

D. Constructing a Rule Generator

Using rule generation based on common specializers described in Section V-B, a rule generator employed by a worker is constructed. To illustrate, suppose that a singleton atom set $\{a\}$, where

$$a = \textit{pat:c}([1,3],[0,*x1,*x2,*x3,*x4,*x5,*x6,0]),$$

is given to a worker. The worker applies the rules in Fig. 4 to make a transformation sequence producing the set G consisting of all ground instances of a that belong to $\mathcal{M}(\{\hat{C}'_0, C'_1, C'_2\} \cup D_{\text{PIC}} \cup D_{\text{BLT}})$. The resulting set G consists of the following three ground atoms:

- $\textit{pat:c}([1,3],[0,1,0,1,1,1,0,0])$
- $\textit{pat:c}([1,3],[0,1,0,0,1,1,1,0])$
- $\textit{pat:c}([1,3],[0,0,1,0,1,1,1,0])$

From the common part of these atoms, the sequence of equality atoms $[=(*x4,1),=(*x5,1)]$ is obtained. Accordingly, the rule

$$\begin{aligned}&\textit{pat:c}([1,3],[0,*x1,*x2,*x3,*x4,*x5,*x6,0])\\ &\Rightarrow \{=(*x4,1),=(*x5,1)\},\\ &\quad \textit{pat:c}([1,3],[0,*x1,*x2,*x3,*x4,*x5,*x6,0])\end{aligned}$$

is generated.

E. Parallel Computation

When the master receives an input problem $q \in Q_{6\times6}$, it creates an initial clause set $Cs_0 = \{\phi(q) \leftarrow q\}$. When the computation starts, the master transforms $\phi(q) \leftarrow q$ using the unfolding rule corresponding to $\hat{C}'_0$. This transformation yields a definite clause with only *pat*-atoms and *pat:c*-atoms in its body. Using the rules in Fig. 4, which are generated from D_{PIC}, *pat*-atoms are successively transformed. By application of the rules r_1 and r_2, *pat*-atoms are all replaced with some other atoms, while *pat:c*-atoms do not disappear. The single-body rules r_1–r_{17} are given priority over the multi-body rule r_{18}. When no single-body rule is applicable, the master uses r_{18}, which increases the number of clauses.

Supposing that the master only applies the rules in its initial rule set without requesting any worker to generate any additional specialized rule, the answer to the puzzle in Fig. 1 is obtained after 10,789 rule application steps, 1,520 of which are clause-splitting steps. In comparison, when workers are used to generate specialized rules on demand based on the proposed architecture, the total number of rule application steps in the master process reduces from 10,789 to 512 in our experiment, with the number of clause-splitting steps reducing from 1,520 to zero.

Employment of specialized rules obtained by rule generation on demand based on run-time content of computation states usually decreases problem-solving time greatly. Compared to application of an existing rule, generation of a new rule itself may take much time. By distributing the tasks of run-time rule generation to workers, the master does not bear the cost of rule generation and, therefore, the overall computation time in the master process substantially reduces.

VII. Conclusions

We establish a theory of direct connection between specifications and correct parallel programs, which shows a sharp contrast to the usual parallel logic programming approaches ([5], [6], [7]), where a human programmer constructs a parallel program based on a specification without a guarantee of correctness. A parallel program in our framework consists of ET rules and rule generators. If all ET rules and all rule generators in a parallel program are correct, then the program is correct. Since there are a large variety of ET rules and rule generators, our framework provides a large space of correct parallel programs, which widens the possibility of finding an efficient program with a relatively small cost. ET rules and rule generators are constructed not only by human programmers, but also by automatic rule generators. A program construction scheme based on rule-set generators and rule-generator generators is described. As long as correct rule-set generators and rule-generator generators are used, a resulting program always yields correct computation and, consequently, verification of the obtained program, which is usually a very expensive task, is not necessary.

References

[1] K. Akama and E. Nantajeewarawat, Formalization of the Equivalent Transformation Computation Model, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 10: 245–259, 2006.

[2] K. Akama, E. Nantajeewarawat, and H. Koike, Program Synthesis Based on the Equivalent Transformation Computation Model, Proc. the 12th International Workshop on Logic Based Program Synthesis and Transformation, Madrid, Spain, pp. 285–304, 2002.

[3] K. Akama, E. Nantajeewarawat, and H. Koike, Program Generation in the Equivalent Transformation Computation Model Using the Squeeze Method, *Perspectives of System Informatics*, Lecture Notes in Computer Science, Vol. 4378, pp. 41–54, Springer-Verlag, Berlin Heidelberg, 2007.

[4] K. Akama, E. Nantajeewarawat, and H. Koike, Constructing Parallel Programs Based on Rule Generators, Technical Report, Hokkaido University, Sapporo, Japan, 2011.

[5] J. Chassin de Kergommeaux and P. Codognet, Parallel Logic Programming Systems, *ACM Computing Surveys*, 26: 295–336, 1994.

[6] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo, Parallel Execution of Prolog Programs: a Survey, *ACM Transactions on Programming Languages and Systems*, 23: 472–602, 2001.

[7] V. Santos Costa, Parallelism and Implementation Technology for Logic Programming Languages, in *Encyclopedia of Computer Science and Technology*, Vol. 42, pp. 197–237, Marcel Dekker Inc., New York, 2000.