Parallelization of Termination Checker of Term Rewriting Systems

Rui Ding, Haruhiko Sato, and Masahito Kurihara,

***Abstract*—Inductive theorem proving plays an important role in the field of formal verification of systems. The rewriting induction(RI) is a method for inductive theorem proving proposed by Reddy. And the multi-context rewriting induction(MRI) proposed by Sato based on the idea of Kurihara has improved the power of RI significantly. However a large amount of termination check of term rewriting systems should be performed in the MRIt, which is becoming the efficiency bottleneck of MRI. In this paper, we propose a method of parallelizing the termination checker used in MRI based on the lexicographic path order method to improve its efficiency. And then we will discuss its efficiency based on the experiments with the system implemented in a parallel functional programming language named Erlang.**

***Index Terms*—Parallelization, Term rewriting system, Termination checker, Multi-context rewriting induction, Lexicographic path order, Erlang**

I. Introduction

A term rewriting system, which is used in the field of automated theorem proving, is a set of rewrite rules that rewrite a term to another. A software tool which checks its termination is called a termination checker. In the field of formal verification of information systems, inductive theorem proving plays an important role. A lot of research has been done to automate the proof discovery, such us the $ewritinginductionwithterminationchecker$ developed by Aoto [8] , which enables researchers to improve the efficiency of the system by customizing the termination checkers. Recently, $multi-context$ rewriting (MRIt for short) developed by Sato [3] has improved the theorem proving techniques. However, a large amount of rapid check of termination is necessary in the MRIt. This causes the standard termination checker based on the dependency pair method proposed by Arts and Giesl [9], to take the most of time of calculation. This is becoming the obstacle of further improvement of its efficiency.

In order to automate and accelerate the MRIt, we propose parallelizing the lexicographic path order method, which is a traditional termination check method, by using multi-core CPU. In this paper we discuss the problem from two viewpoints. One is the exploration of lexicographic path orders, and the other is the large amount of term rewriting systems to be checked. For the implementation, a functional programming language named Erlang has been adopted. The paper is organized as follows. First we review the basic definitions of term rewriting systems and the MRIt. Then we describe our parallelization method and discuss its performance with the experiment. And finally we will come to the conclusion and future work.

Manuscript received December 30, 2011. This work was supported in part by the JSPS KAKENHI No. 22500022.

R. Ding, H. Sato, and M. Kurihara are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan 060-0814. E-mail: ray, haru, kurihara@complex.ist.hokudai.ac.jp

II. Preliminaries

A. Term Rewriting System

We recall the basic notion of term rewriting system briefly. A $signature$ Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative integer, which is the $arity$ of f. For any $f \in \Sigma$, we call f a $constant$ symbol if the arity of f is 0. Let X be a set of $variables$ that is disjoint from Σ. The set $T(\Sigma, X)$ of all $\Sigma - terms$ over X is inductively defined as follows.

- $X \subseteq T(\Sigma, X)$ (i.e. every variable is a term)
- for $t_1, t_2, ..., t_n \in T(\Sigma, X)$ and $f \in \Sigma$, then $f(t_1, t_2, ...t_n) \in T(\Sigma, X)$ where n is the arity of f (i.e. application of function symbols to terms yields terms).

For example, for the signature $\Sigma = \{f, g\}, f(x, g(x, y))$ is a Σ-term that contains the variable x and y. We write $s \equiv t$ if term s is identical with t. A term s is a $subterm$ of t if $s \equiv t$ or $t = f(t_1, t_2, ..., t_n)$ and s is a subterm of some t_i. We denote the set of all variables contained in s by $V(s)$. A $substitution$ is a function $\sigma : V \to T(\Sigma, X)$ such that $\sigma(x) \neq x$ for only finitely many xs. Every substitution σ can be extended to a mapping $\sigma : T(\Sigma, X) \to T(\Sigma, X)$ by defining $\sigma(f(s_1, s_2, ..., s_n)) = f(\sigma(s_1), \sigma(s_2), ..., \sigma(s_n))$. For example, let a substitution $\sigma = \{x \mapsto f(x), y \mapsto g(y)\}$ and a term $s = f(f(x), g(y))$, then $\sigma(s) = f(f(f(x)), g(g(y)))$. We also denote $\sigma(s)$ by $s\sigma$, and we say that t is an $instance$ of s if there exist a σ such that $t = s\sigma$.

A $rewrite\ rule\ l \to r$ is an ordered pair of terms such that l is not a variable and every variables contained in s are also in l. A $term\ rewriting\ system$ (TRS for short) is a set of rewrite rules. A $context$, denoted by C, is a term $t \in T(\Sigma, V \cup \{\Box\})$ with exactly one occurrence of $\Box$. $C[s]$ denotes the term obtained by replacing $\Box$ in C by s. For any term s, $t \in T(\Sigma, X)$ and a TRS R , if there exists a rule $l \to r \in R$ and substitution σ that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$, we denote it by $s \to_R t$ and we call $\to_R$ $reduction\ relation$. A term s is $reducible$ if $s \to_R t$ for some t; otherwise, s is a $normal\ form$. For any term s_0 and TRS R, if infinite rewrite sequence $s_0 \to_R s_1 \to_R \ldots$ does not exist, then $R\ terminates$. A relation R on $T(\Sigma, V)$ is $closedundersubstitution$ if $s\ R\ t$ impliles $s\sigma\ R\ t\sigma$ for any substitution σ. A relation R on $T(\Sigma, V)$ is closed under context if $s\ R\ t$ implies $C[s]\ R\ C[t]$ for any context C. A $reductionorder$ $\succ$ is a well-founded strict partial order on $T(\Sigma, V)$ that is closed under substitution and context. For a term $s \equiv f(t_1, t_2, \ldots t_n)$ where n is the arity of f, we call the symbol f the $root\ symbol$ of s and denote it by $root(s)$. The set of root symbols of all the rules in a TRS R is called the $defined\ symbols$ of R.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

$$\text{DELETE} \quad \langle \mathcal{E} \uplus \{s = s\}, \mathcal{H} \rangle \vdash \langle \mathcal{E}, \mathcal{H} \rangle$$

$$\text{SIMPLIFY} \quad \langle \mathcal{E} \uplus \{s = t\}, \mathcal{H} \vdash \langle \mathcal{E} \cup \{s' = t\}, \mathcal{H} \rangle \quad \text{if } s \to_{\mathcal{R} \cup \mathcal{H}} s'$$

$$\text{EXPAND} \quad \langle \mathcal{E} \uplus \{s = t\}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \text{Expd}_u(s,t), \mathcal{H} \cup \{s \to t\} \rangle \quad \text{if } u \to \mathcal{B}(s) \text{ and } s \succ t$$

Fig. 1. Inference Rules of RI

B. Rewriting Induction

The rewriting induction (RI) proposed by Reddy [6], is a principle for proving inductive theorems in equational logic. We will review the basic notion of it briefly.

Given a set R of rewrite rules representing equational axioms, RI is an inference system defined in Fig.1, working on a pair of set of equations $\mathcal{E}$ and a set of rewrite rules $\mathcal{H}$. Intuitively, $\mathcal{E}$ represents conjectures to be proved and $\mathcal{H}$ represents inductive hypotheses applicable to $\mathcal{E}$.

In Fig.1, Expd denotes the function defined as

$$\text{Expd}_u(s,t) = \{C[r]\sigma = t\sigma | s \equiv C[u], l \to r \in \mathcal{R}, \sigma = mgu(u,l), l : basic\}$$

where, if necessary, the variables used in $l \to r$ should be renamed in a one-to-one manner so that $V(l,r) \cap V(s,t) = \emptyset$. Let $s = t$ be an equation such that it can be oriented from s to t to form a rewrite rule $s \to t$, Given such an equation $s = t$ and a basic subterm u of s, $\text{Expd}_u(s,t)$ overlaps u with the basic left-hand sides l of a rewrite rule $l \to r$ of $\mathcal{R}$. The resultant equations are collected in a set and returned by Expd. Those equations will be used as new conjectures in the EXPAND inference rule for a case analysis to cover the original conjecture $s = t$, if $\mathcal{R}$ is quasi-reducible. In the succeeding inference steps, the rewrite rule $s \to t$ can be used as an inductive hypothesis. And the DELETE rule removes the trivial equations while the SIMPLIFY rule reduces an equation using a rule of $\mathcal{R}$ and $\mathcal{H}$.

We write $\langle \mathcal{E}, \mathcal{H} \rangle \vdash_{RI} \langle \mathcal{E}', \mathcal{H}' \rangle$ if the latter may be obtained from the former by one application of a rule of RI. Given a set of equations $\mathcal{E}_0$, a quasi-reducible terminating TRS $\mathcal{R}$ and a reduction order $\succ$ containing R, if we have a derivation sequence $\langle \mathcal{E}_0, \mathcal{H}_0 \rangle \vdash_{RI} \langle \mathcal{E}_1, \mathcal{H}_1 \rangle \cdots \vdash_{RI} \langle \mathcal{E}_n, \mathcal{H}_n \rangle$ where $\mathcal{H}_0 = \mathcal{E}_n = \emptyset$, then all equations in $\mathcal{E}_0$ are inductive theorems of $\mathcal{R}$. Quasi-reducibility is decidable and there is an algorithm for it, so the possible derivation occurs in the choice of the reduction order $\succ$. This means the choice of the reduction order determines the efficiency of inductive theorem proving with RI. However, the choice of reduction order is undecidable. It means that if the procedure faces the choice for a reduction order, it can not tell which one will lead to the success or failure in the whole procedure. Futhermore, if we simply traversal all the choices, some of the choice might lead to very long sequences of inference or even infinite loops, which causes inefficiency.

III. Multi-Context Rewriting Induction

A. Multi-Context Rewriting Induction

In this section, we describe the multi-context rewriting induction with termination checkers (MRIt) proposed by Sato, which improves the rewriting induction significantly. MRIt is based on the framework of MKB procedure, which simulates the inferences made in all the choices in one process.

$$\text{DELETE}: \quad N \cup \{\langle s : s, H_1, H_2, E \rangle\} \vdash N$$

$$\text{EXPAND}: \quad N \cup \{\langle s : t, H_1, H_2, E \uplus E' \rangle\} \vdash N \cup \{\langle s : t, H_1 \cup E', H_2, E \rangle\} \cup \{\langle s' : t', \emptyset, \emptyset, E' \rangle | s' = t' \in \text{Expd}_u(s,t)\} \text{ if } E' \neq \emptyset, u \in \mathcal{B}(s) \text{ and } \mathcal{H}[N,i] \cup \mathcal{R} \cup \{s \to t\} \text{ terminates for all } i \in E'$$

$$\text{SIMPLIFY} - \text{R}: \quad N \cup \{\langle s : t, H_1, H_2, E \rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, H_1, H_2, \emptyset \rangle \\ \langle s' : t, \emptyset, \emptyset, E \rangle \end{array} \right\} \text{ if } E \neq \emptyset \text{ and } s \to_{\mathcal{R}} s'$$

$$\text{SIMPLIFY} - \text{H}: \quad N \cup \{\langle s : t, H_1, H_2, E \rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, H_1, H_2, E \setminus H \rangle \\ \langle s' : t, \emptyset, \emptyset, E \cap H \rangle \end{array} \right\} \text{ if } E \cap H \neq \emptyset \text{ and } \langle l : r, H, \ldots, \ldots \rangle \in N \; s \to_{l \to r} s'$$

$$\text{FORK}: \quad N \vdash \psi_P(N) \text{ for some fork function } \psi \text{and a set } P \text{ of processes in } N$$

$$\text{GC}: \quad N \cup \{\langle s : t, \emptyset, \emptyset, \emptyset \rangle\} \vdash N$$

$$\text{SUBSUME}: \quad N \cup \left\{ \begin{array}{l} \langle s : t, H_0, H_1, E \rangle \\ \langle s' : t', H_0', H_1', E' \rangle \end{array} \right\} \vdash N \cup \{\langle s : t, H_0 \cup H_0', H_1 \cup H_1', E'' \rangle\} \text{ if } s : t \text{ and } s' : t' \text{ are variants and } E'' = (E \setminus (H_0' \cup H_1')) \cup (E' \setminus (H_0 \cup H_1))$$

$$\text{SUBSUME} - \text{P}: \quad N \vdash sub(N,L)p \in L, \exists p' \in I(N) \setminus L : (\mathcal{E}[N,p], \mathcal{H}[N,p]) = (\mathcal{E}[N,p'], \mathcal{H}[N,p'])$$

Fig. 2. Inference Rules of MRIt

MRIt is represented by an inference system working on a set of nodes. A node is a 4-tuple $\langle s : t, H_1, H_2, E \rangle$ consists of an ordered pair of term $s : t$, three sets of indexes of processes H_1,H_2,E, where each of them is a sequence of natural numbers. In MRIt, the procedure starts with one root process and in the course of the execution, adds new process created by forking existing processes if necessary, when we have nondeterministic choices in applying inference rules. In the node, E represents all processes containing $s = t$ as a conjecture to be proved, and H_1 (resp. H_2) represents all processes containing $s \to t$ (resp. $t \to s$) as an inductive hypothesis. Integrating all the inferences the divergence into node operation, MRIt defines inference rules as shown in Fig.2. In the inference rules, $I(N)$ denotes the set of all processes that appear in a label of a node in N and $sub(N,L) = \{\langle s : t, H_1 \setminus L, H_2 \setminus L, E \setminus L \rangle \mid \langle s : t, H_1, H_2, E \rangle \in N\}$.

Let us set our focus on the EXPAND and SIMPLIFY-R rules. The EXPAND rule operates on a node $n = \langle s : t, H_1, H_2, E \uplus E' \rangle \in N$, and applies the EXPAND rule of RI in all processes of E' that can orient the equation $s = t$ from left to right. The set E' is moved from the third label to the first in n since in each process in E' the conjecture $s = t$ is removed and the new hypothesis $s \to t$ is added after the expansion. In addition, for each new conjecture $s' = t'$ in $\text{EXPD}_u(s,t)$, a new node $\langle s' : t', \emptyset, \emptyset, E' \rangle \in N\}$ is created in order to store the conjecture in the processes of E'. And the DELETE simulates its counterpart of RI, while GC, SUBSUME, and SUBSUME-P are optional rules for efficiency. Especially the third optional rule stops redundant processes, which have the same state as other existing processes. The SIMPLIFY-H rule is almost the same as SIMPLIFY-R. The

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
COMPUTATION TIME OF MRIT

Problem	Total	Term	Simplify
109	0.347	0.281	0.043
301	0.305	0.244	0.044
115	0.042	0.026	0.010
1018	0.028	0.014	0.009
216	0.014	0.003	0.010
total	0.876	0.627	0.169

difference is that SIMPLIFY-R applies a rule of R, which is common to all processes, while SIMPLIFY-H applies an inductive hypothesis of $\mathcal{H}$, which may exist only in some distinguished processes. Finally the FORK is an rule that enables us to produce new copies of existing processes to make nondeterministic choices in parallel.

Let N and N' be two sets of nodes. We write $N \vdash N'$ if the latter is obtained from the former by one application of an inference rule of MRIt. Given a set $\mathcal{E}_0$ of equations and a quasi-reducible terminating TRS $\mathcal{R}$, MRIt starts from the initial set of nodes $N_0 = \{\langle s : t, \emptyset, \emptyset, \{\varepsilon\}\rangle \,|s = t \in \mathcal{E}_0\}$, since MRIt starts with the root simulated process denoted by the empty sequence ϵ. MRIt generates a sequence $N_0 \vdash N_1 \vdash \cdots$.

B. Efficiency Issue

In this section, we will discuss the efficiency of MRIt through some experiments. In the procedure of MRIt, the most frequently used rules are EXPAND, SIMPLIFY and FORK, which determine the efficiency of the MRIt. The FORK simply creates new indexes of processes so its computation time will not be too long to affect the whole procedure. The situation is similar in SIMPLIFY (both R and H), where only reduction relations should be verified during or before the application of them. However, we need to check whether $\mathcal{H}[N, i] \cup \mathcal{R} \cup \{s \to t\}$ terminates for all $i \in E'$ and make a nondeterministic choice according to the result. Since the number of indexes of processes in E' rapidly increases in procedure, the number of TRSs necessary to be checked increases at the same time.

To see how much computation time accounts for the termination verification cost, we can see the experiments of MRIt with the Dream Corpus examples, which are standard examples for inductive theorem proving. In those examples, there were 69 unconditional equational problems suitable for the input to the MRIt. The result is shown in Table 1, in which the column Term and Simplify represent the computation time of termination verification and the application of SIMPLIFY respectively. In the table we only show the problems that required computation time more than 0.01 seconds, and the last row of it indicates the total computation time of all 35 solved problems. Although the computation time isn not very long according to the table, we can see the termination verification takes the most computation time of MRIt. Since the termination checker in MRIt is independent of the whole procedure, if its efficiency was improved, we can say that the efficiency of MRIt will be more improved. The termination checker receives TRSs from SIMPLIFY rule and verify them sequentially, so the coming TRSs must wait until the last verification is completed, which could cause inefficiency. To relieve this bottleneck and improve the efficiency of MRIt, We consider parallelizing this termination check component in both macro and micro viewpoints in terms of multi-core CPU, which can carry out the tasks separately in each core and cut down the waiting time.

IV. PARALLELIZATION

A. Programming Language Erlang

To implement the termination checker efficiently in a multi-core CPU, we have adopted a programming language named Erlang. Erlang is a general-purpose concurrent, garbage-collected programming language run on an efficient runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing. For concurrency it follows the Actor model. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system. We have selected this language because of the following three characteristics.

1) Pattern Matching: A term of TRS always contains function symbols and arguments associated with them. If we want to store them in memory or files, we must treat them separately because they are not necessarily the same data type and combine them together when necessary. In Erlang, there is a distinguished definition of the symbol $=$ called pattern matching. When we have $Lv = Rv$, it means matching the value of Rv with the pattern of Lv. If they match well, Lv will get the value of Rv, otherwise there will be an error. For example, if we run the following command in the shell of Erlang

$$\begin{aligned} X &= 1 + 2 \\ Y &= X + 3 \\ Y &= 7 \\ X &= Y \end{aligned}$$

we will easily get $X = 3$ and $Y = 6$ from the first three command. However, the last of them will throw an badmatch error because X is not equal to Y. Pattern matching in Erlang sounds trivial, but when the left side of the equation has a complex structures, it becomes very convenient to valuate all the variables in it. There are two data structure in Erlang we'd like to mention. A tuple is a structure like $\{x_1, x_2, \ldots, x_n\}$ with a fixed number of data, while a list is a structure like $[x_1, x_2, \ldots]$ with a variable number of data and they can be created recursively. If you want to get values from a complex tuple like $\{a, [b, c, \{d, e, [f, g]\}]\}$, you only need to do a pattern matching

$$\{A, [B, C, \{D, E, [F, G]\}]\} = \{a, [b, c, \{d, e, [f, g]\}]\}$$

and the uppercase letters will be valuated with the lowercase letters (or an error for bad matching). Such characteristics makes it convenient to deal with TRS, which will be described later.

2) Parallelization Orient: There is an amazing thing to the users of Erlang that the program will run n time faster in a n core CPU without any modification. But to achieve this, one must make sure that the program is constructed with processes and there are no interference and sequential bottleneck between them. To avoid the sequential bottleneck

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

in the implementation, we use the feature named the process link in Erlang. After creating a process P_b, we can link it with an existing process P_a for message transfer. A process will send a signal to the linked processes once its task has been completed (or exit with error), and the process which has received the termination signal also terminates if it is not a system process. A system process can be set at the beginning of the process. This link mechanism is a great help in relieving the sequential bottleneck in the implement.

3) Extendability: When we need data or functions from other applications or programs, Erlang can create a port for data transfer instead of running the external program, which makes it extendable with various features.

B. Lexicographic Path Order

To verify the termination, we use the lexicographic path order method, which is a standard method of termination check. First we show the theorem of termination verification and the definition of lexicographic path order.

Theorem 1: A term rewriting system R terminates iff there exists a reduction order $>$ that satisfies $l > r$ for all $l \to r \in R$.

Definition 1: Let Σ be a finite signature and $>$ be a strict order on Σ. The lexicographic path order $>_{lpo}$ on $T(\Sigma, V)$ induced by $>$ is defined as follows: $s >_{lpo} t$ iff
(**LPO1**) $t \in V(s)$ and $s \neq t$, or
(**LPO2**) $s = f(s_1, ..., s_m), t = g(t_1, ..., t_n)$, and
(**LPO2a**) there exist i, $1 \leqslant i \leqslant m$, with $s_i \geqslant_{lpo} t$, or
(**LPO2b**) $f > g$ and $s >_{lpo} t_j$ for all j, $1 \leqslant j \leqslant n$ or
(**LPO2c**) $f = g, s >_{lpo} t_j$ for all j,$1 \leqslant i \leqslant n$, and
there exists i, $1 \leqslant i \leqslant m$ such that
$s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i >_{lpo} t_i$.

The definition of the lexicographic path order is recursive since in (LPO2a), (LPO2b) and (LPO2c) it refers to the relation $>_{lpo}$ to be defined. Nevertheless, $>_{lpo}$ is defined since the definition of $s >_{lpo} t$ only refers to the relation $>_{lpo}$ applied to pairs of terms that are smaller than the pairs s, t. It is proved that the $>_{lpo}$ is stricter than the reduction order, so the termination of TRS R with the signature Σ is proved if we can find out a partial order over Σ.

C. Data Structure

Now we are ready to present the data structure for parallelizing the termination checker. Since only terms and rewrite rules should be constructed by the data structure, we can define representations using the Erlang data type called the tuple as follows:

- r is represented by a tuple $\{Left,\ Right\}$ if $r = Left \to Right$ is a rewrite rule of TRS.
- t is represented by a tuple $\{Fun, [Arg1, Arg2, \ldots]\}$ if t is a term with the function symbol Fun and $Arg1, Arg2, \ldots$ as its arguments.
- t is represented by a tuple $\{t, []\}$ if t is a constant.
- v is represented by a tuple $\{v\}$ if v is a variable.

This definition is based on the recursive definition of term, and we can store any TRS with Erlang easily. For example, we can store the TRS in Fig.3 in a TRS file like Fig.4 above.

Although the TRS file is difficult for us to read, Erlang can read it easily by pattern matching. For example, we only need to match a component with {_} ("_" means any type) to find whether it's variable.

$$\begin{cases} not(not(x)) \to x \\ not(or(x,y)) \to and(not(x), not(y)) \\ not(and(x,y)) \to or(not(x), not(y)) \\ and(x, or(y,z)) \to or(and(x,y), and(x,z)) \\ and(or(y,z), x) \to or(and(x,y), and(x,z)) \\ or(or(x,y), z) \to or(x, or(y,z)) \end{cases}$$

Fig. 3. An Example of TRS

$$\begin{cases} \{\{not, [\{not, [\{x\}]\}]\}, \\ \quad \{x\}\} \\ \{\{not, [\{or, [\{x\}, \{y\}]\}]\}, \\ \quad \{and, [\{not, [\{x\}]\}, \{not, [\{y\}]\}]\}\} \\ \{\{not, [\{and, [\{x\}, \{y\}]\}]\}, \\ \quad \{or, [\{not, [\{x\}]\}, \{not, [\{y\}]\}]\}\} \\ \{\{not, [\{and, [\{x\}, \{y\}]\}]\}, \\ \quad \{or, [\{not, [\{x\}]\}, \{not, [\{y\}]\}]\}\} \\ \{\{and, [\{x\}, \{or, [\{y, \{z\}]\}]\}, \\ \quad \{or, [\{and, [\{x\}, \{y\}]\}, \{and, [\{x\}, \{z\}]\}]\}\} \\ \{\{and, [\{or, [\{y\}, \{z\}]\}, \{x\}]\}, \\ \quad \{or, [\{and, [\{x\}, \{y\}]\}, \{and, [\{x\}, \{z\}]\}]\}\} \\ \{\{or, [\{or, [\{x\}, \{y\}]\}, \{z\}]\}, \\ \quad \{or, [\{x\}, \{or, [\{y\}, \{z\}]\}]\}\} \end{cases}$$

Fig. 4. A TRS File

Besides the definition of TRS, we define a partial order between function symbols by the list of binary tuple of them. Since there is no partial order when the termination check procedure starts, we initiate it as $I = [\ ]$. When there is a new order element $f > g$ to be added, we put $\{f, g\}$ into I if it causes no conflict the the partial orders in I.

D. Parallelization Algorithm

In this section we describe the parallelization algorithm of termination verification. First the termination verification of a single rule is shown in Fig.5. If the TRS is empty, the algorithm terminates; otherwise we compute the list of partial orders that make the left-hand side of the first rule greater than its right-hand side in $>_{lpo}$, which can be easily obtained by the definition of lexicographic path order and the data structure of Erlang. For each partial order obtained, we compute the list of partial orders that make the left-hand side of the second rule greater than its right-hand side in $>_{lpo}$. We continue this operation until we find one partial order that makes the left-hand side of the last rule greater than its right-hand side in $>_{lpo}$ or we find that there is no such partial order in any branches. In the process, a single [ ] means there is no proper partial order, while [[ ]] means there is a proper partial order with no constraint. At each choice point in the procedure, we create a process for each partial order just obtained. This leads to our algorithm as follow:

1) If the TRS is empty, return terminate.
2) Create a supervisor process to monitor the set of created processes by the link feature of Erlang. If there is no created process of termination verification, return non-terminate.
2) Initiate a partial order as $p = [\ ]$, do step 3 with the first rule of TRS and p.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

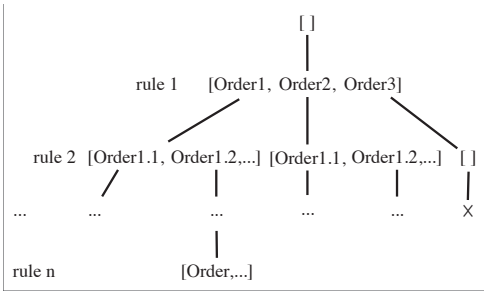

Fig. 5. Procedure of Lexicographic Path Order Method

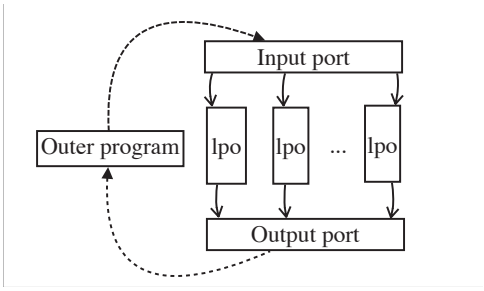

Fig. 6. Macrolevel Parallelization

3) Based on p, compute $P_{list} = \{p_1, p_2, \dots\}$, which is the list of all the possible partial orders of given rewrite rule. Then we see if L is empty:

- do step 4, if P_{list} is not empty.
- terminate this process, if L is empty.

4) For each $p_i \in P_{list}$, create a process for it and do step 3 in the process.

One may notice the synchronization problem in the algorithm : if the supervisor process starts before any other children processes, it will return non-terminate even before the termination verification. So we lock the supervisor process until all the possible partial orders are obtained and sent to children processes. Besides this, there is also another small synchronization problem, but it can be solved by the link and message transfer features of Erlang.

In the macrolevel viewpoint as shown in Fig.6, the termination checker should take a stream of TRSs and their identifiers as input. In Erlang, we set an input port, which receives data from other applications or programs. Then each received TRS is assigned to a new process, and its termination will be checked using the algorithm described above. When the verification is over, the result is sent to the output port for sending it to the external program. In Erlang, each process is computed on its independent memory, so there is no any interference between the processes in the macrolevel parallelization.

V. Experiment

In this section, we discuss the result of the experiment. To test the efficiency of the proposed method, we have used 100 TRSs in http://www.termination-portal.org/wiki/TPDB which is the database of TRSs. The implementation and experiments are have been conducted on a workstation with two AMD Opteron 2.3GHz CPUs which have 12 cores each. This means we have 24 cores in the workstation. Because the

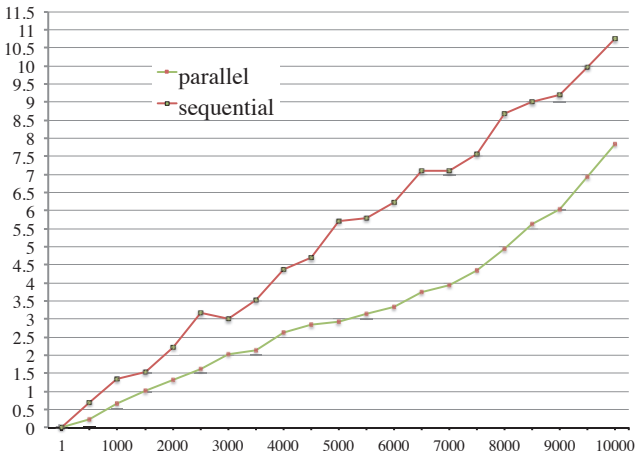

Fig. 7. Experiment Result

number of TRSs which could be proved terminating with the lexicographic path order was moderate, we have simply conducted the termination verification of 100 examples as many as 100 times instead of verifying distinct 10000 examples. Nevertheless, those set of examples is sufficient to measure the efficiency of the proposed method. We have also written a sequential termination checker to compare with our method.

From the result shown in Fig.7, in which the green line represents the computation time of the proposed method while the red line represents that of the sequential program, we can see the proposed method is much efficient than the sequential program especially when the number of TRSs is large. However, when the number of TRSs goes near 10000, the efficiency of the proposed method begins to be slowly getting down. We think this is due to the constraint of the number of cores and processes allowed in the CPU.

VI. Conclusion

In this paper, we have presented a parallelized implementation of termination checker of term rewriting systems, and discussed its efficiency based on the experiments. As a future work, we are planning to implement the dependency pair method, which is complicated but can be more powerful than the lexicographic path orders. Finally we will use our termination checker to improve the efficiency of MRIt.

References

[1] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge, England: Cambridge University Press, 1999.
[2] J. Armstrong, *Programming Erlang:Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
[3] H. Sato and M. Kurihara, "Muti-Context Rewrting Induction with Termination checkers," *IEICE Transactions on Information and Systems*, vol. E93-D, no. 5, pp. 942-952, May. 2010.
[4] M. Kurihara and H. Kondo, "Completion for Multiple Reduction Orderings," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 25-42, 1999.
[5] N. Hirokawa and A. Middeldorp, "Tyrolean Termination Tool: Techniques and Features," *Information and Computation*, vol. 205, no. 4, pp. 474-511, 2007.
[6] U. Reddy, "Term Rewriting Induction," *10th International Conference on Automated Deduction, vol.814 of Lecture Notes in Computer Science*, pp. 162-177.
[7] N. Hirokawa and A. Middeldorp, "Automating the Dependency Pair Method," *Information and Computation*, vol. 199, no. 1-2, pp. 172-199, 2005.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

[8] T. Aoto, "Rewriting Induction Using Termination Checker," *JSSST 24th Annual Conference*, 3C-3, 2007.
[9] T. Arts and J. Giesl, "Termination of Term Rewriting Using Dependency Pairs," *Theoretical Computer Science*, vol. 236, no. 1-2, 2007.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)