TMLAI TRANSACTIONS ON MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE
SOCIETY FOR SCIENCE AND EDUCATION UNITED KINGDOM
VOLUME 2, ISSUE 3
ISSN: 2054 - 7390

Parallelization of Termination Checkers for Algebraic Software

Rui Ding, Haruhiko Sato, and Masahito Kurihara
Graduate School of Information Science and Technology, Hokkaido University, JAPAN;
{ray,haru}@complex.ist.hokudai.ac.jp, kurihara@ist.hokudai.ac.jp

ABSTRACT

Algebraic software is modeled as a set of equations representing its specification, and when each equation is directed either from left to right or from right to left, the resultant set of directed equations (or rewrite rules) is called a term rewriting system, which can be interpreted as a functional program executed by the pattern matching and term rewriting. In the field of formal verification of information systems, most of the properties of such a system are formalized as inductive theorems, which are equations over terms which hold on recursively-defined data structure such as natural numbers, lists and trees. Well-known as a method for inductive theorem proving is the Rewriting Induction (RI) proposed by Reddy. Recently, this method was extended by Sato and Kurihara to the Multi-context Rewriting Induction with termination checker (MRIt), which is a variant of RI to try to find a suitable context for induction automatically. However, MRIt should perform a large amount of termination checking of term rewriting systems, causing a significant efficiency bottleneck. In this paper, we propose a method of parallelizing the termination checkers used in MRIt to improve its efficiency by focusing on the well-known typical termination checking method based on the lexicographic path orders. For implementation, we used the functional concurrent programming language Erlang. We discuss the efficiency of our implementation based on the experiments with the standard set of termination problems.

Keywords: Algebraic Software, Term Rewriting System, Termination, Parallelization.

1 INTRODUCTION

Algebraic software is modeled as a set of equations representing its specification, and when each equation is directed either from left to right or from right to left, the resultant set of directed equations (or rewrite rules) is called a term rewriting system [1]. A term rewriting system is a set of rewrite rules used for rewriting a term to another, and can be interpreted as a functional program executed by the pattern matching and term rewriting. Given a term rewriting system, we always concern about its termination. A software tool which checks its termination is called a termination checker. The termination property is a very important

DOI: 10.14738/tmlai.24.368
Publication Date: 28th August 2014
URL: http://dx.doi.org/10.14738/tmlai.24.368

property especially in automated inductive theorem proving, which tries to prove inductive theorems, which are equations over terms which hold on recursively-defined data structures, such as natural numbers, lists and trees. Once we have found that the system is terminating in the inductive theorem proving, we can use the transitive closure of the associated rewriting relation as a well-founded order over terms for the basis of induction.

To automate the inductive theorem proving, a lot of methods have been proposed. Here we only mention one of the most refined ones, i.e., the Rewriting Induction (RI) proposed by Reddy [2]. The RI is a principle which has successfully generalized and refined several procedures proposed so far for proving inductive theorems based on term rewriting systems. The RI relies on the termination of term rewriting systems created from the axiomatic equations. However, there are some strategic issues coming from the non-determinism in constructing proofs. The most critical issue is that, in the proof procedure of the RI, we have to choose appropriate proof steps, considering which reduction order should be employed to prove the termination and which rules should be employed for rewriting. In general, it is difficult to choose appropriate strategies leading to the success, because we do not know the final result beforehand. In the RI, the issue of choosing the reduction order is fixed before starting the procedure, by specifying a particular reduction order used to decide the direction of equations to transform them into rewrite rules and ensure that the resultant term rewriting systems have the termination property. However, it is most difficult to properly decide such a particular reduction order beforehand, making the RI really hard to automate.

To solve this problem, Aoto [3] proposed a variant of the RI, named the Rewriting Induction with termination checker (RIt). This method has enabled researchers to improve the efficiency of the inductive theorem proving systems by customizing the external termination checkers, instead of using the reduction order given beforehand in the built-in termination checkers. However, although the RIt has solved some strategic issues mentioned above, there comes another issue instead: in which direction equations should be directed. From the viewpoint of the strategy, the use of the external termination checkers gives us more flexibility in the direction strategy, because the dynamic decision on the direction contributes to the increase in the possibility of the success of the theorem proving. Thus in order to prove inductive theorems automatically, we can now exploit this flexibility by trying various direction strategies in parallel. However, it is clear that if we physically created and ran a number of parallel processes in a naive parallelization scheme, it would cause serious inefficiency.

Recently, Sato and Kurihara [4] proposed a new variant of the rewriting induction procedures called the Multi-context Rewriting Induction with termination checkers (MRIt) based on the idea of multi-completion of Kurihara and Kondo [5] . The procedure simulates the execution of parallel RIt processes in a single process. There are inductive theorems which are easily proved by the MRIt but were not proved by the standard RI or RIt unless the strategies and contexts were chosen correctly or else auxiliary lemma were discovered and supplied. The

Copyright © Society for Science and Education United Kingdom

MRIt improved the efficiency of inductive theorem proving significantly. However, a large amount of rapid check of termination is necessary in the MRIt. This causes the standard termination checker to take a lot of time for calculation, especially when the checker is based on the dependency pair method, one of the most powerful methods recognized in the associated community, proposed by Arts and Giesl [6]. This is becoming the obstacle for further improvement of its efficiency. In order to automate and accelerate the MRIt, we propose in this paper the use of the multi-core CPU to parallelize the lexicographic path order method, a well-known termination checking method implemented in a lot of termination checkers. We discuss the problem from two viewpoints. One is the exploration of the lexicographic path orders, and the other is a large amount of term rewriting systems to be checked. For the implementation, a functional concurrent programming language named Erlang has been adopted.

The paper is organized as follows. In Section 2, we will briefly review the basic definitions on term rewriting systems. Then we will present our parallelization method in Section 3, and discuss its performance in Section 4. Finally we will come to the conclusion and discuss our future work in Section 5.

2 PRELIMINARIES

Let us briefly review the basic definitions and notations for term rewriting systems (TRSs). In TRSs, a term will be built from function symbols and variables in the usual way. For example, if f is a binary function symbol and x and y are variables, then $f(x, y)$ is a term. To make clear which function symbols are available in a certain context, you need to specify a signature as defined below.

Definition 1: A *signature* Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative integer *n*, the arity of f. The elements of Σ with arity *n*=0 are called *constant symbols*.

For example, if we want to talk about a group, a well-known algebraic structure equipped with an identity element e, a unary inversion operation i and a binary multiplication operation f, we use the signature $\Sigma = \{e, i, f\}$, where e has arity 0, i is unary, and f is binary. If we want to talk about the set of non-negative integers, we may use the signature consisting of the smallest non-negative integer 0, the successor function *s* (meaning *s*(*x*)=*x*+*1*), and some arithmetic functions such as $+$ and $\times$.

With the definition of signature, we can define terms as follows.

Definition 2: Let Σ be a signature and X be a set of variables such that $\Sigma \cap X = \{\}$. The set $T(\Sigma, X)$ of all Σ*-terms over* X (or simply *terms* if Σ and X are clear from the context) is inductively defined as

- $X \subseteq T(\Sigma, X)$ (i.e., every variable is a term)

- If $t_1, t_2 \ldots, t_n \in T(\Sigma, X)$ and $f \in \Sigma$, then $f(t_1, t_2, \ldots, t_n) \in T(\Sigma, X)$, where n is the arity of f (i.e., application of a function symbol to argument terms yields a term).

For example, for the signature $\Sigma = \{f, g\}$ with two binary function symbols, $f(x, g(x, y))$ is a term containing the variables x and y. For a constant symbol e, we write the corresponding term simply as e instead of $e()$. Some binary function symbols (such as $+$ and $\times$) are written in infix form, with parentheses if necessary, like $(x + y) + z$ instead of $+(+(x, y), z)$.

The main difference between constant symbols and variables is that the latter may be replaced by terms specified with substitutions. A *substitution* is a function $\sigma : V \to T(\Sigma, X)$ that maps every variable to a term. The set of variables $\{x_1, \ldots, x_n\}$ with $\sigma(x_i) = t_i \neq x_i$, $1 \leq i \leq n$, is called the domain of σ. In this case, we may write $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Every substitution σ can be extended to a mapping $\sigma : T(\Sigma, X) \to T(\Sigma, X)$ from terms to terms by introducing a new regulation $\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n))$. In words, the application of a substitution to a term simultaneously replaces all occurrences of variables by their respective images.

Definition 3: A *rewrite rule* is an ordered pair of terms (l, r) such that l is not a variable and $Var(l) \supseteq Var(r)$. We may write $l \to r$ instead of (l, r). A *term rewriting system* (TRS) is a set of rewrite rules. Note that the rewrite rule can be considered as an equation $l = r$ directed from left to right.

Let $\Box$ be a new symbol which does not yet occur in $\Sigma \cup X$. A *context* is a term $C \in T(\Sigma, X \cup \{\Box\})$ with a single occurrence of $\Box$. For a term s and a context C, $C[s]$ denotes the term obtained by replacing $\Box$ in C by s. For any terms $s, t \in T(\Sigma, X)$ and a TRS R, if there exists a rewrite rule $l \to r \in R$, a context C, and a substitution σ such that $s \equiv C[\sigma(l)]$ and $t \equiv C[\sigma(r)]$, we say that s can be *rewritten to* t by a rewrite rule of R and write $s \to_R t$. We call $\to_R$ a reduction relation. A TRS R *terminates* if it allows no infinite rewrite sequences $s_0 \to_R s_1 \to_R \cdots$. In this case, one often says that R is *terminating* or R has the *termination* property. We can prove the termination of term rewriting systems by using the following definition and theorem on reduction orders.

Definition 4: A strict partial order $>$ on $T(\Sigma, X)$ is called a *reduction order,* if it satisfies the following properties.

- closed under context: $s > t$ implies $C[s] > C[t]$, for all contexts C.
- closed under substitution: $s > t$ implies $\sigma(s) > \sigma(t)$ for all substitutions σ.
- well-founded: there are no infinite decreasing sequences $s_0 > s_1 > \cdots$.

Theorem 1: A term rewriting system R terminates if, and only if, there exists a reduction order $>$ that satisfies $l > r$ for all rewrite rules $l \to r$ of R.

Copyright © Society for Science and Education United Kingdom

3 PARALLELIZATION

3.1 Programming Language Erlang

To implement the termination checker efficiently in a multi-core CPU, we have adopted a programming language named Erlang [7]. Erlang is a general-purpose concurrent 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.

Pattern Matching

A term is either a variable or a function symbol followed by zero or more argument terms. To store terms in memory, we often design a recursively-defined tree-like structure, distinguishing between variables and function symbols by using different data types or naming conventions. To conduct term rewriting, we compare the structure of two terms (the pattern and the data) and decide the substitution σ, if it exists, to rewrite the data by a rewrite rule with that pattern in its left-hand side. Erlang has a convenient mechanism, *pattern matching*, which can be used for this purpose. An expression of the form $L = R$ in Erlang means the instruction for matching the value of R with the pattern of L. If they match well, the variables in L will get the corresponding value in R; otherwise there will be an error. For example, if you run in the shell of Erlang the commands $X = 1 + 2$, $Y = X + 3$, $Y = 6$, $X = Y$, you will easily get $X = 3$ and $Y = 6$ from the first three commands. However, the last one will throw a bad-match error, because X is not equal to Y. Pattern matching in Erlang is simple, but when the left-hand side of the equation has a complex structure, it becomes very convenient to valuate all the variables in it. There are two data structures in Erlang we would like to mention. A *tuple* is a structure with a fixed number of data specified in the form $\{x_1,\ldots,x_n\}$ with fixed *n*, while a *list* is a structure with a variable number of data specified in the form $[x_1,\ldots,x_n]$. If you want to get values from a complex tuple like $\{a,[b,c,\{d,e,[f,g]\}]\}$, you only need to do the pattern matching $\{A,[B,C,\{D,E,[F,G]\}]\} = \{a,[b,c,\{d,e,[f,g]\}]\}$ to get the variables in uppercase letters valuated with the data in lowercase letters. Such characteristics make it convenient to deal with TRSs.

Efficient Parallelization

An amazing thing to the users of Erlang is the fact that the program will run n times faster in a n core CPU without any modification. But to achieve this, you must make sure that the program is constructed with processes and there are no interferences and sequential bottlenecks among them. To avoid sequential bottlenecks in the implementation, you can use

the feature named the *process link* in Erlang. After creating a process P_b, you 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 processes which have received the termination signal also terminate unless they are system processes. A system process can be set at the beginning of the process. This link mechanism is a great help in relieving sequential bottlenecks in the implementation.

Extendability

To communicate with other applications or programs, Erlang can create a process called a *port*. Ports provide your programs with various features to cooperate with external programs. The external programs are run outside the Erlang runtime system. The virtual machine running the Erlang processes copies data through the port to and from the port's driver controlling the external programs. Messages can be sent to a driver through a port by using the same operator, !, used to send messages to regular Erlang processes. Messages sent by drivers to Erlang are also received using the same operator, receive. With this mechanism, your Erlang programs can be easily extended with external programs in a transparent way.

3.2 Lexicographic Path Order

To verify the termination, we use the lexicographic path order (LPO), which is a basic reduction order used in the literature.

Definition 5: Let Σ be a finite signature and $>$ be a strict partial order (called a *precedence*) on Σ. The *lexicographic path order* $>_{lpo}$ on $T(\Sigma, X)$ induced by $>$ is defined as follows:

$s >_{lpo} t$, if and only if

(**LPO1**) s is not a variable and t is a variable that occurs in s, or

(**LPO2**) $s = f(s_1,\ldots,s_m)$, $t = g(t_1,\ldots,t_n)$, and

(**LPO2a**) there exists $i, 1 \le i \le m$, with $s_i \ge_{lpo} t$, or

(**LPO2b**) $f > g$ and $s >_{lpo} t_j$ for all $j, 1 \le j \le n$, or

(**LPO2c**) $f = g$, $s >_{lpo} t_j$ for all $j, 1 \le j \le n$, and

there exists $i, 1 \le i \le m$, such that $s_1 = t_1,\ldots,s_{i-1} = t_{i-1}$ and $s_i >_{lpo} t_i$.

The definition of the LPO is recursive, since in (LPO2a), (LPO2b) and (LPO2c) it refers to the relation $>_{lpo}$ to be defined. Nevertheless, $>_{lpo}$ is well 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 $>_{lpo}$ is a reduction order, so the termination of TRS R with the signature Σ is

Copyright © Society for Science and Education United Kingdom

proved if we can find out a precedence $>$ over Σ such that the LPO $>_{lpo}$ induced by $>$ satisfies $l >_{lpo} r$ for all rewrite rules $l \to r$ of *R*.

3.3 Data Structure

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

- A variable v is represented by a tuple $\{v\}$ with a single element.
- A constant c is represented by a tuple $\{c,[]\}$ of a symbol and the empty list.
- A term with the function symbol Fun and its arguments $Arg1, Arg2, \cdots$ is represented by a tuple $\{Fun,[Arg1, Arg2, \cdots]\}$ of a symbol and a non-empty list.
- A rewrite rule $Left \to Right$ is represented by a tuple $\{Left, Right\}$ of two elements where the second one is not a list.

This definition is based on the recursive definition of terms. Note that those four kinds of objects can be clearly distinguished by their syntactical patterns. With those representations, we can store any objects for TRSs in the Erlang environment. For example, we can store the TRS given in Figure 1 in a TRS file as in Figure 2.

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

Figure 1. An Example of TRS

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

Figure 2. A TRS File

Although the TRS file is difficult for us to read, Erlang can read it easily by using the pattern matching. For example, we only need to match an object with the pattern {_} ("_" means a "wild card" matching with any data) to decide whether it is a variable or not.

Besides the data structures for TRSs, we define the data structure for a precedence $>$ over the function symbols as the list of binary tuples of them. For example, the precedence defined by $f > g$, $g > h$ and $f > h$ is represented as the list $[\{f,g\},\{g,h\},\{f,h\}]$.

Since there is no precedence when the termination checker starts, we initiate it as $I = []$. When we need to add a new element $f > c$, we put $\{f,c\}$ into I, making sure that it preserves the properties required for precedences, without causing no conflict with the current precedence represented as I.

3.4 Parallelization

In this subsection we describe the parallelization architecture for termination verification. Actually, we propose two schemes of parallelization: microlevel and macrolevel.

Microlevel parallelization

First, the architecture for the termination verification of a single TRS is shown in Figure 3.

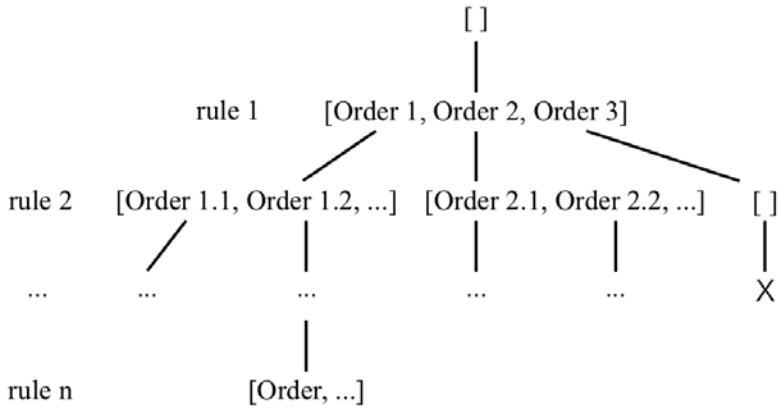

Figure 3. Microlevel parallelization in termination check with Lexicographic Path Orders

If the TRS is empty, the procedure terminates; otherwise it computes the list of the precedences $>^{(i)}$, $i = 1,2,\cdots$, that make the left-hand side of the first rule greater than its right-hand side in $>_{lpo}^{(i)}$. This can be easily computed from the definition of LPO [8]. The procedure then tries to extend each precedence $>^{(i)}$ obtained, so that it can further compute the list of the precedences $>^{(i,j)} \supseteq >^{(i)}$, $j = 1,2,\cdots$, that make the left-hand side of the second rule greater than its right-hand side in $>_{lpo}^{(i,j)}$. The procedure continues this operation until it finds a

Copyright © Society for Science and Education United Kingdom

precedence $>^{(i,j,\cdots)}$ that makes the left-hand side of the last rule greater than its right-hand side in $>_{lpo}^{(i,j,\cdots)}$ or else it finds that there is no such precedence in any branches. In the figure, a single [] means there is no such precedence. (This should be distinguished from [[]], which means there is an empty precedence in the list.) At each choice point, the procedure creates a parallel process for each precedence just obtained. This procedure can be summarized as follows.

1. Create a supervisor process to monitor the set of created processes by the link feature of Erlang, and create and start a process that executes the step 2 with the rule number $i=1$ and the precedence $p=[]$. The supervisor process waits until a created process returns "terminate" successfully or else it finds there are no created processes, and returns "terminate" in the former case and "failure" in the latter case.
2. Given i and p, if there is no ith rule, then return "terminate"; otherwise compute $P_{list}=\{p_1, p_2, \ldots\}$, which is the list of all the precedences that are extensions of p and make the left-hand side of the ith rule greater than its right-hand side in the induced LPO, and
   - terminate this process, if P_{list} is empty.
   - execute the step 3, if P_{list} is not empty.
3. For each $p_j \in P_{list}$, create and start a process that executes the step 2 with $i+1$ and p_j.

One may notice a subtle synchronization problem in the procedure: if the supervisor process starts before any other children processes, it will return "failure" even before the termination verification is started. To avoid this, we lock the supervisor process until all the possible precedences are obtained and sent to children processes. Besides this, there is also another small synchronization problem, but it is solved by the link and message transfer features of Erlang.

Macrolevel parallelization

Now let us think about two or more TRSs to verify the termination of. Our architecture for such an application is based on the macrolevel viewpoint as shown in Figure 4, where the termination checker should take a stream of TRSs and their identifiers as input. We create an input port in Erlang which receives TRSs from other applications or programs. Each received TRS is assigned to a new process, and its termination will be checked using the procedure described above (designated as lpo in the figure). When the verification is over, the result is sent to the output port to send it to the external program. In Erlang, each process is executed on its independent memory, so there is no interference among the processes in the macrolevel parallelization.

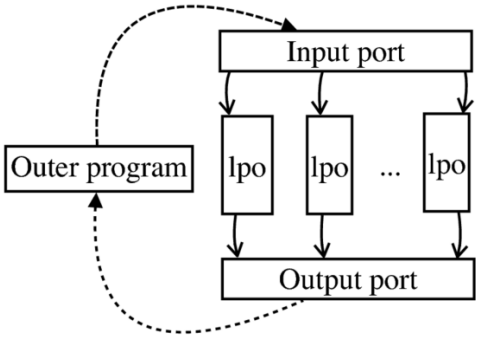

Figure 4. Macrolevel Parallelization in termination check for multiple TRSs

4 EXPERIMENT

In this section, we show and discuss the results of the experiments. In the experiments, we used the standard problems stored in the Termination Problem Data Base [9], which contains 2,125 TRSs to be checked for their termination. The implementation and experiments were performed on a workstation with two AMD Opteron 2.3GHz CPUs with 12 cores. This means we had 24 cores in the workstation.

First we conducted the experiment on all the 2,125 problems. We deliberately eliminated all the IO operation time in the measurement of the computation time so that we only measure the computation time of the termination check. In order to compare the proposed parallelization with the non-parallelization, we also wrote a sequential termination checker in Erlang. We show the result in Figure 5, in which the horizontal axis represents the number of the cores, while the vertical axis represents the computation time in microseconds.

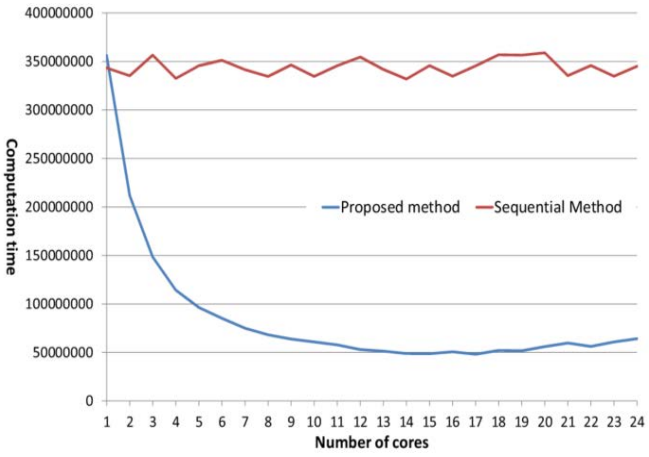

Figure 5. Result of experiment on all the TRSs

With a single core, the computation time of the proposed method was almost the same as the sequential method. With multi-cores, however, the computation time decreased significantly when we increased the number of cores. As for the sequential program, its computation time was basically unchanged, because it creates no parallel processes even when a lot of cores are available. The change in efficiency with the increase of cores can be observed

Copyright © Society for Science and Education United Kingdom

more clearly in Figure 6, where the vertical axis represents the reciprocal number of the computation time, indicating the speed of computation in terms of the amount of work done in a microsecond.

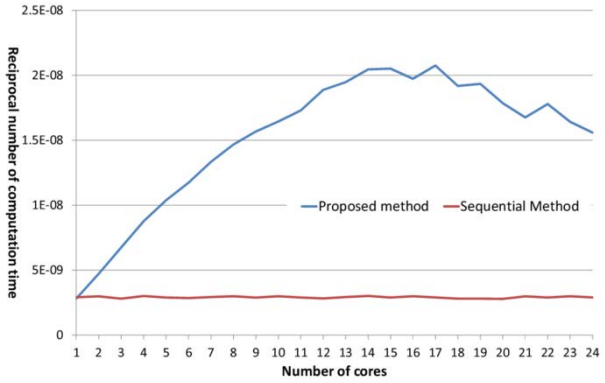

Figure 6. Efficiency result of experiment on all the TRSs

Obviously, the speed of computation increased with the increase in cores, when the number of cores is less than 16. However, when the number of cores exceeded 16, the increase in the efficiency of the proposed method came to the limit and even went down.

In order to figure out the reduction in efficiency, we divided all the TRSs into three groups by the number of rewrite rules they contain. The 46 TRSs which contain 50 or more rules were classified as a group named *Large,* and 300 TRSs with 15 or more but less than 50 rules were named *Medium.* The remaining 1,779 TRSs with less than 15 rules were put into the last group named *Small*.

New experiments were conducted for each of the three groups. First we repeated the same experiment described above, but this time, the program was run for each group as input. In addition, we performed another experiment where only microlevel parallelization was activated for each group. The results are shown in Figure 7, Figure 8 and Figure 9, where "Proposed method" shows the results when both microlevel and macrolevel parallelizations were activated, while "Microlevel Parallel" only activated the microlevel parallelization.

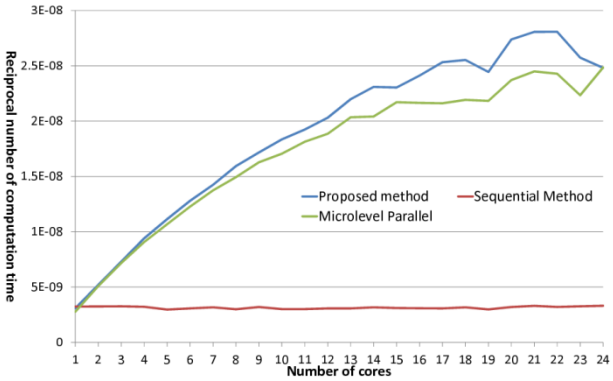

Figure 7. Efficiency result of experiment on Large TRSs

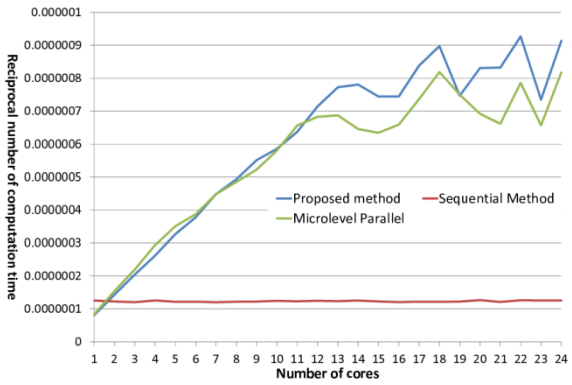

Figure 8. Efficiency result of experiment on Medium TRSs

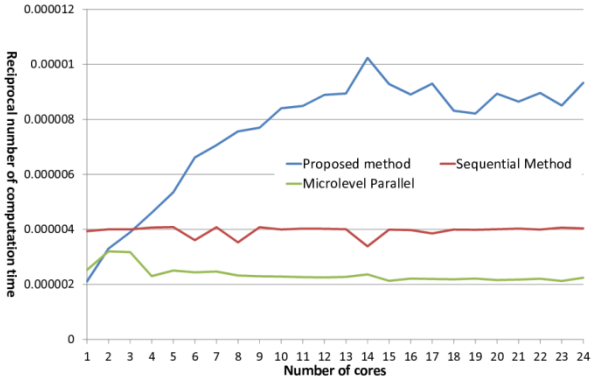

Figure 9. Efficiency result of experiment on Small TRSs

As in the original experiment, the efficiency of the sequential program was almost unchanged with the number of cores, and the efficiency of the proposed method increased with the increase of the number of cores, showing obvious better efficiency than the sequential program. However, a difference from the original experiment can be seen in the figure for the Large group, where the efficiency of the proposed method increased until the number of cores came to 23 rather than the original 16. This is in contrast with the figures for the Medium and Small groups, where the increase in efficiency came to limit when the number of cores became nearly 14.

For the cases of the Large and Medium TRSs, the microlevel parallelization attained almost the same performance as the proposed method, which performs both micro- and macro-level parallelizations. This implies that the microlevel parallelization was effective but the macrolevel one was not effective for those cases. For the case of the Small TRSs, on the other hand, we can see the macrolevel parallelization was very effective and the efficiency of the microlevel parallelization was even worse than the sequential method. The results show that the macrolevel parallelization did not work well for a small number of TRSs, and the microlevel parallelization decreased its efficiency for TRSs with a small number of rewrite rules. From the results, we can say that if the number of TRSs and the number of rewrite rules in those TRSs is large enough, the proposed method is useful and efficient. Fortunately, in practice of

Copyright © Society for Science and Education United Kingdom

termination checking performed in the inductive theorem proving, we often encounter a large number of complex TRSs, which make the proposed method satisfactory to us.

5 CONCLUSION

In this paper, we have proposed a parallel method of termination checking for term rewriting systems using the lexicographic path order method. The efficiency of the proposed method was shown to be satisfactory for the applications with a large number of complex TRSs generated for termination checking. However, the power of the lexicographic path order is not strong enough to solve a lot of termination problems we encounter. It is a challenging task as a future work to try to parallelize a more powerful termination checking method such as the dependency pair method [6, 10], and finally improve and automate the inductive theorem proving.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 25330074.

REFERENCES

[1]. Baader, F and Nipkow, T. Term Rewriting and All That, Cambrdge University Press, 1998.

[2]. Reddy, U. Term Rewriting Induction, Proc. of 10th International Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 449, pp. 162-177, 1990.

[3]. Aoto, T. Rewriting Induction Using Termination Checker, Proceedings of JSSST 24th Annual Conference, 3C-3, 2007.

[4]. Sato, H and Kurihara, M. Muti-Context Rewrting Induction with Termination checkers, IEICE Transactions on Information and Systems, vol. E93-D, no. 5, pp. 942-952, 2010.

[5]. Kurihara, M and Kondo, H. Completion for Multiple Reduction Orderings, Journal of Automated Reasoning, vol. 23, no. 1, pp. 25-42, 1999.

[6]. Arts, T and Giesl, J. Termination of Term Rewriting Using Dependency Pairs, Theoretical Computer Science, vol. 236, no. 1-2, pp. 133-178, 2000.

[7]. Armstrong, J. Programming Erlang: Software for a Concurrent World, Second Edition, Pragmatic Bookshelf, 2013.

[8]. Kurihara, M and Kondo, H. Efficient BDD Encodings for partial order constraints with application to expert systems in software verification, Proceedings of 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Lecture Notes in Artificial Intelligence, vol. 3029, pp.827-837, 2004.

[9]. Termination problems data base. [Online] http://termination-portal.org/wiki/TPDB.

[10]. Hirokawa, N and Middeldorp, A. Tyrolean Termination Tool: Techniques and Features, Information and Computation, vol. 205, no. 4, pp. 474-511, 2007.