International Journal of Applied Mathematics in Control Engineering 2 (2018) 119-126

Contents lists available at YXpublications

International Journal of Applied Mathematics in Control Engineering

Journal homepage: http://www.ijamce.com

An Automatic Refactoring Approach for Loop Parallelism

Dongwen Zhang, Mengmeng Wei, Yang Zhang*, Shixin Sun, Shicheng Dong

School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei, China

ARTICLE INFO

Article history:
Received 9 April 2018
Accepted 9 July 2018
Available online 25 December 2018

Keywords:
Loop parallelism
Safety analysis
Thread Pool mechanism
Abstract Syntax Tree

ABASRACT

This paper proposes an automatic software refactoring approach to achieve the parallelism for loops. Our approach uses pre-conditions and post-conditions to ensure the consistency of performance before and after refactoring. The Thread Pool mechanism provided by Java Executor framework is used to complete the parallelization of loops. We implement our approach as an automatic interactive refactoring tool (*R-loop*) based on Abstract Syntax Tree (AST) of the Eclipse JDT environment. Several benchmarks from the JGF Benchmark Suite are selected to evaluate *R-loop*. Experimental results show that the efficiency of programs execution has been improved. *R-loop* can successfully implement the automatic refactoring in a short time without introducing other parallelization errors.

1. Introduction

As the modern processor technology shifts from its main frequency to multi-core processing, multi-core processors are getting closer to us. It makes programmers begin to use parallel computing to improve program performance. As a basic structure of iterative processing of data by many algorithms, the loop contains abundant parallelism, but due to its complexity and variety, it is often the most time-consuming part of programs. Therefore, loop parallelism has become a hot research in high performance computing.

The parallel technology is gradually rising with the development of parallel computers. However, it is usually performed manually by programmers in candidate programs through parallel mining, parallel code generation and optimization. It is expensive, time-consuming, error-prone and simply not scalable. The manual parallelization is too complex and instabilities arising from newly introduced bugs.

This paper proposes the software refactoring approach and implements an automatic refactoring tool called *R-loop*. We present the safety analyses to determine whether loops are conformed to the conditions of parallelization and use *R-loop* to parallelize several loops in real programs. It completes the automatic loop parallelization efficiently and effectively.

The rest of the paper is organized as follows. The refactoring of loop is introduced in section 2 which includes the safety analysis, loop parallelization, and transformation. Experimental results are presented in section 3 and related literatures are examined in section 4. Finally, conclusions are drawn in section 5.

2. Refactoring for loop

Refactoring is the process of changing internal structure of software without changing the external behavior. It is a convenient way to clean up code that minimizes the chances of introducing bugs. Unlike a simple loop parallelization, refactoring is applicable to a wide of possible parallel structure.

2.1 Architecture

The architecture of refactoring for loop is shown in Fig.1. It is mainly composed of three parts: initialization, transformation, and consistency detection.

The front-end and back-end of the process are a serial program and a parallel program, respectively. We first check the pre-conditions that may be used to check whether the target program is suitable to be transformed or not. After validating successfully, we need to add some thread-related operations to convert a serial program to a parallel one. We use Java Executor framework to complete the parallelization of loops. Code transformation is the core of refactoring process, which usually includes locating the source code, traversing the AST, and generating JDT. The modified codes are not directly reflected in the original program, but saved as a *change* object, which is easily to be

* Corresponding author.
E-mail addresses: zhangyang@hebust.edu.cn (Y. Zhang)

checked for programmers. The post-conditions are usually conducted to ensure that the behavior of program is not changed.

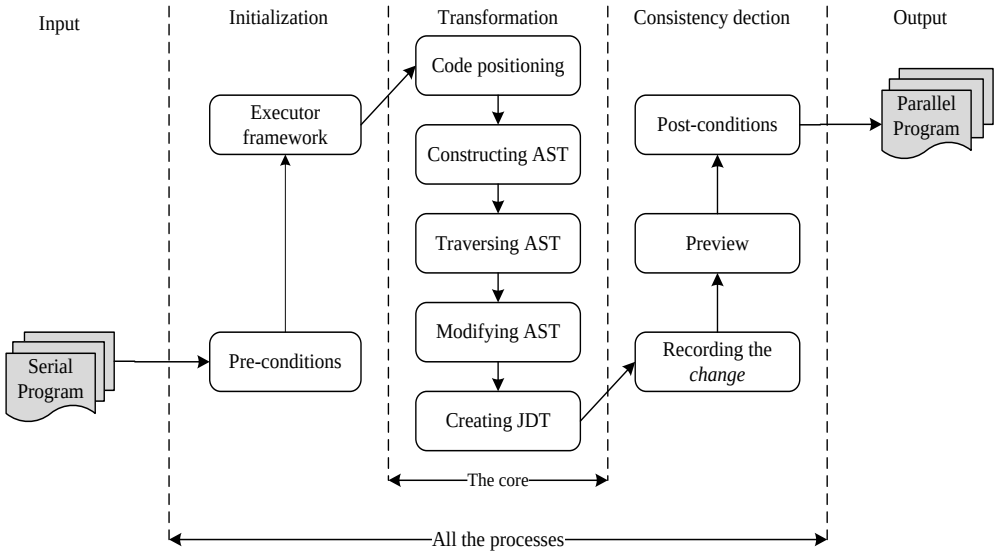


Fig. 1. The architecture of refactoring.

2.2 Safety analysis

One of the most important considerations in our refactoring is to preserve the original sequential semantics when executing methods in parallel. Therefore, safety analysis becomes an indispensable part of the whole refactoring, and it includes data dependence analysis and loop iteration analysis.

The aim of data dependence analysis is to determine if loop iteration may depend on data written in previous loop iteration. In such cases, automatic loop parallelization is usually prohibited. First, we show a couple of examples in Fig.2. On the left-hand of example 1, *S1: a[2] = b[1]+c;* is obtained in the first iteration (*i=1*), in which variable *a[2]* is written to memory. *S2: d[2] = a[2]+e;* is obtained in the second iteration (*i=2*), while variable *a[2]* is read but its read value is dependence on the write operation in the previous iteration, in other words *S2* is loop data dependency on *S1*. In this case, there are two or more operations simultaneously access *a[i]*, and at least one is the *write* operation. Similarly, in example 2 the control statement is the value written of *a[i]* in the last iteration. And we call that *S2* is loop control dependency on *S1*.

To avoid reading and writing to the same memory address, a new temporary variable *x* (in example 1) was allocated to hold the writes. By comparing the simple code, it can be seen that the original write twice, and read twice are reduced to the current write twice, and read once, and the data dependence are removed. In example 2, we define two temporary variables, temp and next. And the temp is used to save the previous value of *a[i]*.

Loop iteration analysis is to ensure that the loop parallelism should satisfy the following three conditions: the upper and lower bounds of loop variables should be a constant; conditional expression in loop should be conformed to the form "*loop_variable*<, <=, >,>= *loop_invariant_integral*"; and the loop increment is integer one. If these values are not constant, we will issue some warnings. And if the loop increment is not integer one, we will normalize it by multiplying the loop counter by the original.

In our refactoring, the properties of input validity illustrates all input from the users is legally, it is possible to apply the transformation to the given program with the given inputs. Class *RefactoringWizard* provides method *addUserInputPages* and *checkInitialCondition*. The former is used to add user input page, and the latter is used to check the initialization. If the security degree is *RefactoringStatus.FATAL*, the refactoring cannot continue unless it is passed.

example 1

```
1.for(inti=0;i<n;i++){          1.d[0] = a[0]+e;
2. a[i+1] = b[i]+c;    //S1     2.for(inti=0;i<n;i++){
3. d[i] = a[i]+e;      //S2     3. x = b[i-1]+c;
4.}                             4. a[i] = x;
                                5. d[i] = x+e;
                                6.}
                                7.a[n] = b[n-1]+c;
```

example 2

```
1. for(inti = 1;i <n;i++){      1. temp = a[0];
2.   a[i] = b[i]+c[i];   //S1   2. for(inti=1;i <n;i++){
3.   if(a[i-1]){          //S2   3.   next = b[i] + c[i];
4.      c[i] = a[i]+2;          4.   if(temp){
5.   }                          5.     c[i] = next + 2;
6. }                            6.   }
                                7.   a[i] = next;
                                8.   temp = next;
                                9. }
```

Fig. 2. Examples of loop.

2.3Loop parallelism

We introduce the multi-thread mechanism to complete the conversion form serial program to parallel program. Starting with JDK 1.5, the Executor framework provided by Java concurrency library begins to use thread pool to manage threads. The thread pool can receive object *Runnable* or *Callable* directly without inheriting class thread repeatedly. After the operation is completed, those threads in this pool will not be destroyed but transformed their current state into *sleep* state, thus decreasing the overhead of repeated creation of threads.

To ensure the sequence of data, a linked-hash map is chosen to save the input data, which is a First-In-First-Out (FIFO) queue. We create some threads for initialization which number is related to the amount of input data, and those threads are in *wait* state. It does not bind threads with processor cores, but creates an *ExecutorService* pool to manage threads.

To saving the invocation time, the original loop is encapsulated in a static internal class, which is inherited from class *thread.* The number of activated threads is determined by the parameter *n* in method *newFixedThreadPool(n).* Each thread in the thread pool fetches a data from the FIFO queue continuously, and finishes method invocation by the method *execute* of class *ExecutorService.*

It should be note that, there is a barrier operation implemented by method *waitForAll* of class *ExecutorService* at the end of each thread. The barrier operation checks whether current operation is finished. If not, it will wait until all the threads are processed.

2.4Transformation

Abstract Syntax Tree (AST) is an abstract grammar structure that represents the intermediate representation of the refactoring process. It can parse Java code into a tree. Programmers complete the modification of original code by traversing it, changing its property, inserting and deleting nodes and others operations.

Eclipse AST provides class *ASTPaster* for parsing source code, which means that the transformation from Java to AST will be completed by *ASTPaster*. If the input parameter is a whole java file, method *setSource* completes this parsing process. If the input parameter is other types (e.g., *compilation_unit*, *class_body_declarations*, *expression*, *andstatements*), method *setKind* will be used. We use the Method *selectionChanged* to acquire the target refactoring object and saved them in an*ICompilationUnit* which represents a source file that can be compiled.

After parsing, a corresponding abstract syntax tree is generated. Eclipse AST provides recursive traversal methods for class *Visitor*, *ASTNode* acts as an abstract element, *ASTVisitor* acts as an abstract visitor. According to the order of access, it can be divided into *PreVisit(Node node)*, *Visit(Node node)*, *PostVisit(Node node)*, respectively. Generally, programmers need to create a sub-class of *ASTVisitor*. The traversal of variables, method domain and methods are completed by modifying parameters of the sub-class to *TypeDeclaration*, *FieldDeclaration* and *MthodDeclaration*. Note that, *TypeDeclaration* and *MthodDeclaration* can obtain the related properties and output directly, but there are other variables that being declared under *FieldDeclaration*, therefore, we need to traverse its sub-node (*VariableDeclarationFragment*).

After traversing, we need to get the list of all statements, and generate the JDT code by inserting, deleting and repairing it. The AST of statement *ExecutorService pool = Executors.newFixedThreadPool(n)* is shown in Fig.3. We use it to illustrate the correspondence between variables in code and nodes in AST. This statement is a *VariableDeclarationStatement*, which is the body of method *buildTestData* and whose type is *ExecutorService*. The name of the fragment is *pool*, and its initialization is an *expression*(*Executors*) named *newFixedThreadPool*. The *numberLiteral(n)* represents the number of running threads. Note that the *binding* should remain unchanged throughout the whole process.

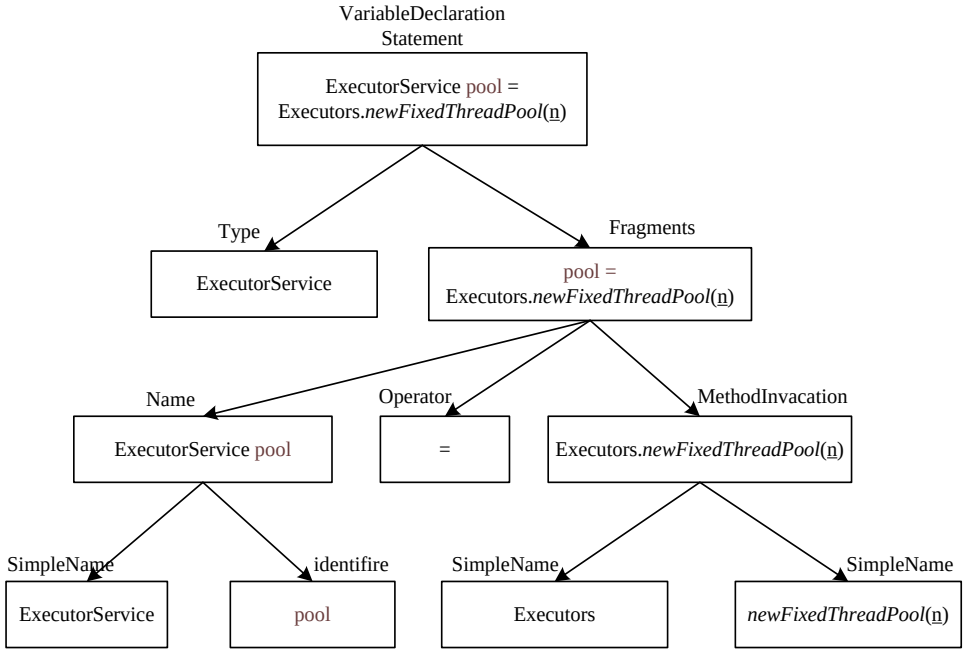


Fig. 3. An example of AST.

We have implemented an interactive refactoring tool, *R-loop*, which automates complete the safety analysis and the rewriting of code. It is integrated with Eclipse's refactoring engine, so it offers some convenient features of a refactoring engine: previewing the changes, preserving the formatting, undoing the changes. Our tool performs the safety analysis and warns the programmer if some pre-conditions are not met, then *R-loop* rewrites the code. The final user interface is shown in Fig.4.

The left side of the interface is the original program, and the right side is the program which has been refacted. The transformation part of the automated refactoring is marked with a gray contrast, so that the users can clearly understand which part has been changed. In the preview interface, it can be initially determined whether the automated refactoring program adds the thread-related operations or

not. For example, whether the *Execute()* method, wait for All() method are correctly added, and whether a fixed-size thread pool is created.

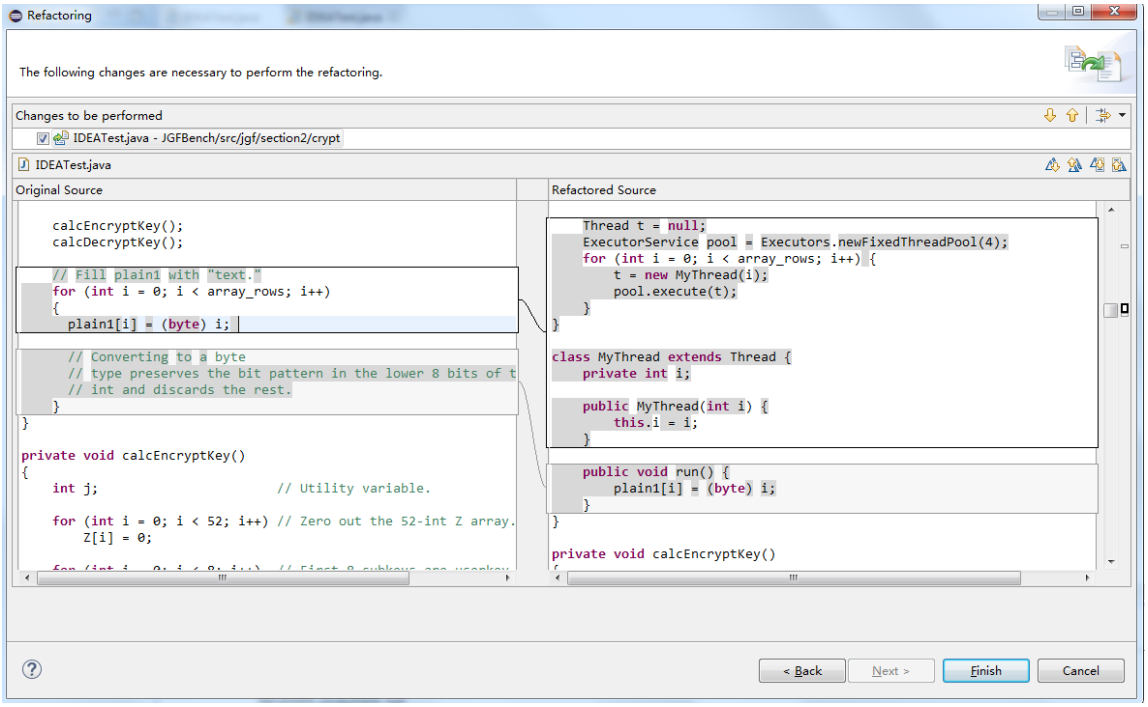


Fig. 4. The final UI of refactoring.

3. Experimental methodology

3.1 Experiment environment

We used Intel Xeon i5-6500 CPU with 8GB of RAM running Windows 7 server (4 processors, 4 cores on each processors and each core ran at 3.2 GHz; 4 hardware threads). At software level, we used Eclipse4.5.1 and JDK1.8.0_31.

3.2Benchmarks

We evaluate our experiment using four benchmarks (Crypt, LUfact, Series and SparseMatmulti) from Java Grande Forum (JGF) Benchmark Suite (Smith and Bull, 2001). This suite contains a set of benchmarks that can be used to test the performance of Java for scientific computations. Crypt benchmark performs International Data Encryption Algorithm on an array of N bytes. LUfact benchmark performs an algorithm of Lu matrix decomposition. Series benchmark performs an algorithm for solving the Fourier coefficient. And The SparseMatMulti benchmark is mainly used to perform multiplication of sparse matrices.

These benchmarks and their input data size are shown in Tab. 1.

Tab. 1. Benchmarks and the input data.

Benchmark	Size A	Size B	Size C
Crypt	3000000	20000000	50000000
LUfact	500	1000	2000
Series	1000	100000	1000000
SparseMatmulti	250000	500000	25000000

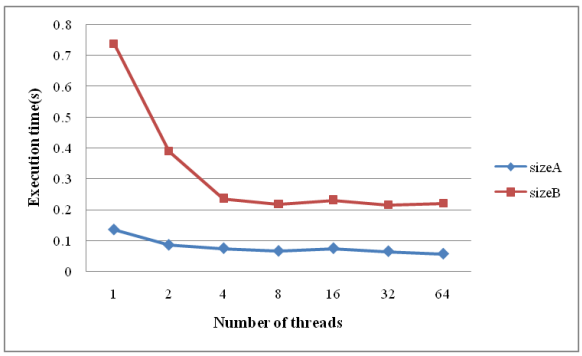
3.3 Experiment results and analysis

Due to the uncertain time of parallel execution, we executed each benchmark ten times, and calculated the average value. The performance of Crypt, LUfact, Series and SparseMatmulti are illustrated in Fig.5 (a), Fig.5 (b) , Fig.5 (c) and Fig.5 (d). The graphs plot execution time, so lower is better. Considering the high execution time of these benchmarks in Size C, we present the results in two figures respectively.

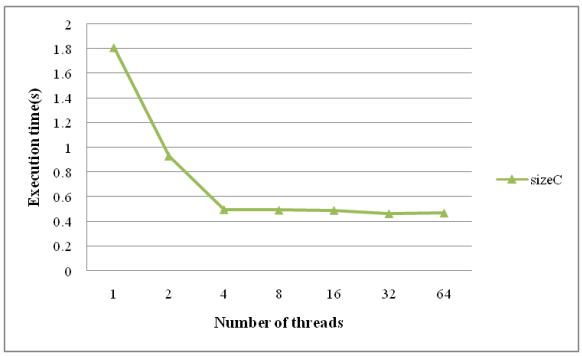
The execution of Crypt benchmark is shown in Fig.5 (a). We can see that the execution time is dramatically decreased when the benchmark is executed from serial to parallel, the execution efficiency is also significantly improved. With the increase of number of software threads, the execution time is decreased and gradually stabilized. Longitudinal comparison shows that when the amount of input data in the program is different, the parallelization effect generated is also different. It is illustrated that the size of data is one of the important factors for influencing the effect of parallelization. Similarly, this situation happens in Fig.5 (c) and Fig.5 (d).

In general, the larger the input data of the program, the more obvious performance comparison before and after refactoring, and the better efficiency is gain. A reason that cannot be ignored is that when the amount of data is sufficient, the time consumed by the creation, management, release the threads is relatively reduced in the total execution time, which makes the degree of parallelization obviously.

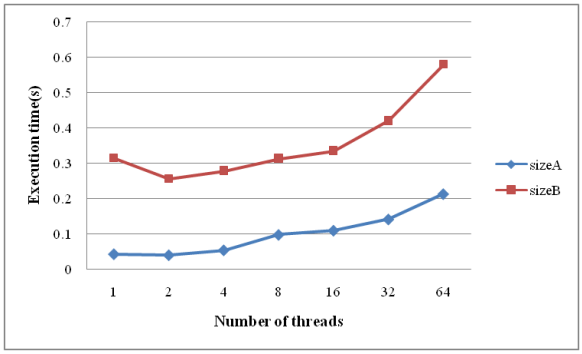
The execution of LUfact benchmark is shown in Fig.5 (b), when the number of software threads is equal to 8, the number of software threads is more than the number of hardware threads, execution time begins to increase. This result is inconsistent with the conclusion that "the more threads, the less execution time". In this situation, if we continue to increase the number of software, the performance will continue to become poor. If the number of threads is more than 64, the execution time even exceeds the execution time of sequence. It occurs because the creation, management, and release of a large number of threads bring time consuming when the input data is not insufficient, thus reduces the efficiency of the whole program.



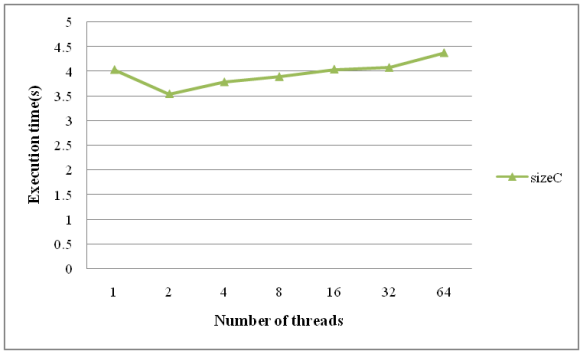
（a）Crypt



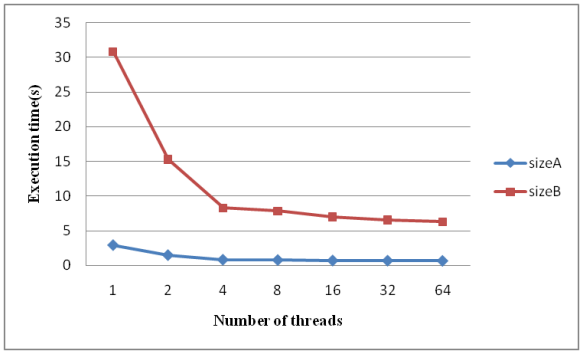
（a）Crypt-sizeC



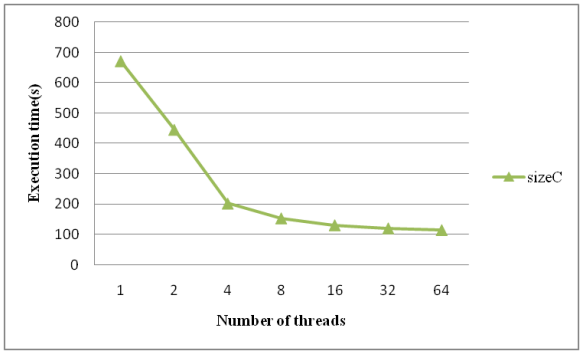
（b）LUfact



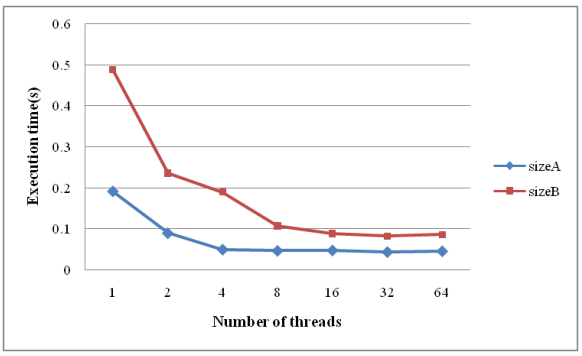
（b）LUfact-sizeC



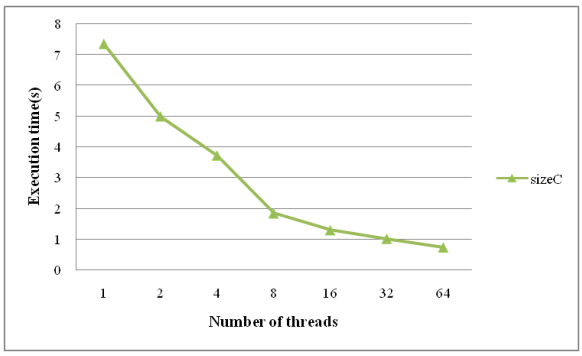
（c）Series



（c）Series-sizeC



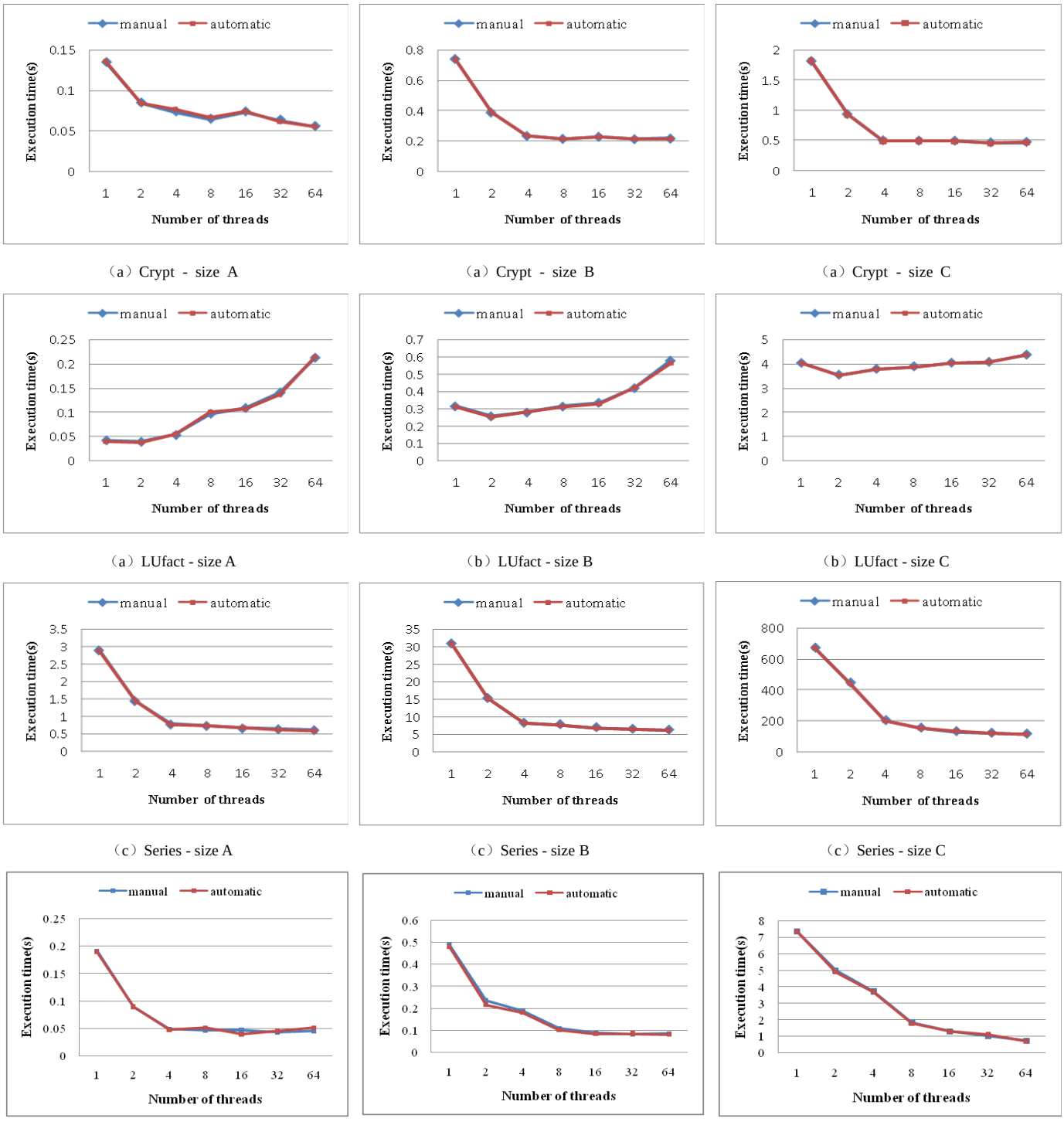
（d）SparseMatmulti



（d）SparseMatmulti-sizeC

Fig. 5. Execution time of benchmarks from varying threads

We evaluate refactoring time of each benchmark. When we run the four benchmarks on *R-loop*, the automatic refactoring time is 365 ms, 668ms, 547ms and 399ms, respectively. All time are less than 1 second, which illustrates *R-loop* is more efficient than manual parallelization. We also evaluate execution time of each benchmark. The difference of manual parallelization time and automatic parallelization time is between 1% and 3%, which is basically consistent with each other. Experimental results show *R-loop* can get good scalability. The results are shown in Fig.6.



（a）Crypt - size A　（a）Crypt - size B　（a）Crypt - size C

（a）LUfact - size A　（b）LUfact - size B　（b）LUfact - size C

（c）Series - size A　（c）Series - size B　（c）Series - size C

(d) SparseMatmulti - size A　(d) SparseMatmulti - size B　(d) SparseMatmulti - size C

Fig. 6. The execution time of benchmarks by manual and automatic refactoring

In addition to comparing the execution time of the program before and after refactoring, this paper also analyzes the speedup of the program. The speedup is used to measure the performance and effectiveness of parallel system or program parallelization.

Assumed that some parts of the program perform in the serial execution. If the ratio of the serial execution part to the entire program is *f*, the ratio of the parallel part is *1-f*. So the total execution time of using *n* threads is calculated as follows, and t_s represents the execution time of using a single thread.

$$ft_s + \frac{(1-f)t_s}{p} \tag{1}$$

Therefore, the speedup of the program is calculated as follows:

$$S = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}} = \frac{p}{1+(p-1)f} \tag{2}$$

The results of the speedup are given in Tab.2. It is illustrated that the automated refactoring for loop parallelization can bring an increase in execution speed. *R-loop* can successfully implement the automatic refactoring in a short time without introducing other parallelization errors.

Tab. 2. The speedup of each benchmarks.

Benchmark	Size A	Size B	Size C
Crypt	2.41	3.43	3.69
LUfact	1.07	1.33	1.14
Series	4.71	4.91	5.82
SparseMatmulti	4.43	5.94	7.23

4. Related works

Some related works are discussed in this section. ReLooper (Dig et al., 2009) is an automatic refactoring tool which developed by Professor Danny dig that can help the programmer parallelize regular loop nests in Java code. Parallelization is done using Parallel Array framework, and relies on static data dependency analysis to detect parallelism. To use ReLooper, the programmer selects a target array or vector and is then told if the loops that access the array can be parallelized safely. Our work is complementary to ReLooper. But our approach is neither data-driven nor array-driven, whereas we focus on loops. And we use Executor framework to implement the parallelization of loops.

An interactive compilation feedback system (Larsenet al., 2012) is developed which guided the programmer in iteratively modifying the source code. Their compiler feedback suggested that the restrict keyword be used to eliminate sets of dependence, and suggested steps to resolve key issues. Vandierendonck et al. (Vandierendonck et al., 2010) propose the Paralax infrastructure, which exploits programmer knowledge for optimization. Paralax is tool that suggests 0how the programmer may add annotations to the program. It parallelizes irregular, pointer-intensive code and relies on profiling information. Kale et al. (Kale et al., 2013) present a method to capture dependence in loops with VD3 (variable distance data dependences). They use a two variable LED and its parametric solutions to analyze and mathematically formulate precise dependence.

Rajam et al. (e.g., Rajam et al., 2015; Jimborean et al., 2014; Sukunmaran-Rajam et al., 2014) propose a framework called Apollo that implements a dynamic and speculative adaptation of the ploy tope model. Apollo is able to optimize and parallelize loop nests that cannot be handled at compile-time and that exhibit a linear behavior at runtime. The next year, they will go further by proposing a strategy to extend their framework to loop nests that exhibit a non-linear behavior.

Mazaheri et al. (Mazaheri et al., 2015) extend DiscoPoP profiler to create a specialized tool for identifying nested communication patterns inside shared-memory applications and propose a static analysis approach for annotating loop regions. Mata et al. (Mate et al, 2013) achieved the cyclic structure's parallelism by analyzing the regular code, separating data, selecting appropriate communication strategies and eliminated the data competition.

Li et al. (Li et al., 2015) used the tree and graph recursive algorithm to dealing with the irregular cyclic structure, and proposed some different parallelization templates which are relying on the dynamic parallel analysis technology. Dutta et al. (Dutta et al., 2016) proposed a new algorithm for constructing dependency graph of the parallel programs, which verifies the correctness of the parallel programs by checking the equivalence of the original sequence. Klimek et al. (Klimek et al., 2017) proposed a method of parallelization of the non-synchronous mechanism in the nested loop, which allows extracting any nested loop parallelism.

The research team led by Hammond and Brown (e.g., Brown et al., 2013; Hammond et al., 2012; Danelutto et al., 2014; Loidl et al., 2011) set up a formal refactoring rule to enhance programmability of parallel programming, and implemented a parallel refactoring tool ParaPhrase. They also proposed a language-independent parallel refactoring framework; the Erlang program has been refectory. In addition, they studied how to convert a serial Haskell program to a parallel Haskell program.

Ruixia (Ruixia, 2018) proposed a novel adaptive parameter identification method to identify the parameter vectors. Some newly multil-ateration systems for monitoring and scene surveillance were proposed by Tongjuan (Tongjuan et al., 2018) and Rui (Rui et al. 2018).

5. Conclusions

In this paper, we proposed an automatic software refactoring approach for parallelization of loops. Our approach presented some specific details of its implementation, such as safety analysis, loop parallelization and transformation of parallel. We also developed an automatic refactoring tool named *R-loop*, and evaluated it by several benchmarks in JGF Benchmark Suite, such as Crypt, LUfact, and Series. Experimental results had shown that *R-loop* can complete the parallelization in a short time, and the efficiency of programs execution has been improved.

Acknowledgements

This work is partially supported by National Natural Science Foundation of China under grant No. 61440012, Natural Science Foundation of Hebei Province under Grant No. F2016208007 and Fundamental Research Foundation of Hebei Province under Grant No.18960106D. The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers, which have improved the presentation.

References

Smith, L. A., & Bull, J. M. (2001). A parallel java grande benchmarksuite. Supercomputing, ACM/IEEE 2001 Conference (pp.8-8). IEEE.

Dig, D., Tarce, M., Radoi, C., Minea, M., & Johnson, R. (2009). Relooper:refactoring for loop parallelism in Java. Companion to the Acm Sigplan Conference on Object-oriented Programming (pp.793-794).

Dig, D. (2010)A Practical Tutorial on Refactoring for Parallelism. Proceedings of IEEE International Conference on Software Maintenance. Piscataway, NJ: IEEE.

Larsen, P., Ladelsky, R., Lidman, J., Mckee, S. A., Karlsson, S., & Zaks, A. (2012). Parallelizing more loops with compiler guided refactoring. 410-419.

Vandierendonck, H., Rul, S., & Bosschere, K. D. (2010). The Paralax infrastructure: automatic parallelization with a helping hand. International Conference on Parallel Architectures and Compilation Techniques(pp.389-400). IEEE.

Kale, A., Patil, A., & Biswas, S. (2013). Parallelization of loops with variable distance data dependences. Computer Science.

Rajam, A. S., Campostrini, L. E., Caamano, J. M. M., & Clauss, P. (2015). Speculative Runtime Parallelization of Loop Nests: Towards Greater Scope and Efficiency. Parallel and Distributed Processing Symposium Workshop (Vol.4, pp.245-254). IEEE.

Jimborean, A., Clauss, P., Dollinger, J. F., Loechner, V., & Caamaño, J. M. M. (2014). Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. International Journal of Parallel Programming, 42(4), 529-545.

Sukumaran-Rajam, A., Caamaño, J. M. M., Wolff, W., Jimborean, A., & Clauss, P. (2014). Speculative Program Parallelization with Scalable and Decentralized Runtime Verification. Runtime Verification. Springer International Publishing.

Mazaheri, A., Jannesari, A., Mirzaei, A., & Wolf, F. (2015). Characterizing Loop-Level Communication Patterns in Shared Memory. International Conference on Parallel Processing (pp.759-768). IEEE.

Mata, L. L. P. D., Pereira, F. M. Q., & Ferreira, R. (2013). Automatic parallelization of canonical loops. Science of Computer Programming,78(8), 1193-1206.

Li, D., Wu, H., & Becchi, M. (2015). Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations. International Conference on Parallel Processing (Vol.17, pp.979-988). IEEE.

Dutta, S., Sarkar, D., Rawat, A., & Singh, K. (2016). Validation of Loop Parallelization and Loop Vectorization Transformations. International Conference on Evaluation of Novel Software Approaches To Software Engineering (pp.195-202).

Klimek, T., Palkowski, M., & Bielecki, W. (2017). Synchronization-Free Automatic Parallelization for Arbitrarily Nested Affine Loops. International Symposium on Computer Architecture and High PERFORMANCE Computing Workshops (pp.43-48). IEEE.

Brown, C., Hammond, K., Danelutto, M., Kilpatrick, P., Schöner, H., & Breddin, T. (2013). Paraphrasing: Generating Parallel Programs Using Refactoring. Formal Methods for Components and Objects. Springer Berlin Heidelberg.

Brown, C., Hammond, K., Danelutto, M., & Kilpatrick, P. (2012). A language-independent parallel refactoring framework. The Workshop on Refactoring TOOLS (pp.54-58). ACM.

Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., & Elliott, A. (2014). Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming, 42(4), 564-582.

Brown, C., Loidl, H. W., & Hammond, K. (2011). ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. Trends in Functional Programming. Springer Berlin Heidelberg.

Ruixia Meng. (2018). Adaptive Parameter Estimation for Multivariable Nonlinear CARMA Systems. International Journal of Applied Mathematics in Control Engineering, 1, 96-102.

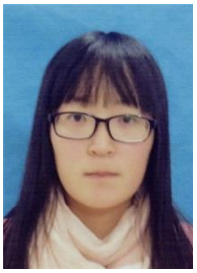
Tongjuan Zhao, Jiuhe Wang. (2018). Adaptive Parameter Estimation for Multivariable Nonlinear CARMA Systems. International Journal of Applied Mathematics in Control Engineering, 1, 55-61.

Rui Peng, Lei Chrng, Yating Dai, Xitong Zhao, Huaiyu Wu, Yang Chen (2018). Adaptive Parameter Estimation for Multivariable Nonlinear CARMA Systems. International Journal of Applied Mathematics in Control Engineering, 1, 85-91.

Dongwen Zhang received her PHD degree in School of computer at Beijing Institute of Technology. She is an professor in school of Information Science and Engineering at Hebei University of Science and Technology. Her research interests focus on parallel programming model and software refactoring for parallelism.

Mengmeng Wei is currently a candidate for a Master's degree at Hebei University of Science and Technology. Her research interests focus on parallel programming and software refactoring for parallelism.

Yang Zhang received his PHD degree in School of Computer at Beijing Institute of Technology. He is an associate professor in school of Information Science and Engineering at Hebei University of Science and Technology. His research interests focus on parallel programming model and software refactoring for parallelism.



Shixin SUN is currently a candidate for a Master's degree at Hebei University of Science and Technology. Her research interests focus on parallel programming and software refactoring for parallelism.

Shicheng DONG is currently a candidate for a Master's degree at Hebei University of Science and Technology. His research interests focus on parallel programming and software refactoring for parallelism.