International Journal of Internet, Broadcasting and Communication Vol.12 No.3 87-94 (2020)
http://dx.doi.org/10.7236/IJIBC.2020.12.3.87

IJIBC 20-3-11

A Study on the Automatic Parallelization Method and Tool Development

Woochang Shin

Professor, Dept. of Computer Science, Seokyeong University, Korea
wcshin@skuniv.ac.kr

Abstract

Recently, computer hardware is evolving toward increasing the number of computing cores, not increasing the clock speed. In order to use the performance of parallelized hardware to the maximum, the running program must also be parallelized. However, software developers are accustomed to sequential programs, and in most cases, write programs that operate sequentially. They also have a lot of difficulty designing and developing software in parallel. We propose a method to automatically convert a sequential C/C++ program into a parallelized program, and develop a parallelization tool that supports it. It supports open multi-processing (OpenMP) and parallel patterns library (PPL) as a parallel framework. Perfect automatic parallelization is difficult due to dynamic features such as pointer operation and polymorphism in C/C++ language. This study focuses on verifying the conditions of parallelization rather than focusing on fully automatic parallelization, and providing advice to developers in detail if parallelization is not possible.

Keywords: *Automatic Parallelization, Code Transformation, PPL, OpenMP, Parallelization Tool*

1. Introduction

Until the early 2000s, computer hardware manufacturers used a method to increase the clock frequency to improve the performance of the computer, but reached the limit due to the increase in electricity usage and heat generation problems. Since then, hardware manufacturers have abandoned the competition to improve clock speed and switched to increasing the number of compute cores [1]. For example, Intel Core i9-9980XE includes 18 cores, and Xeon W-3175X includes 28 cores. GPUs also include more cores for faster image processing. For the Tesla v100, which Nvidia announced in June 2017, the number of cores reached 5120 [2].

In order to efficiently use the performance of parallelized computer hardware, software must be written for parallelism. However, software developers are familiar with sequential programs, and in most cases, write programs that operate sequentially. After all, when running on a CPU with 8 cores, it can only use about 1/8 of the performance provided by hardware. To achieve the best performance on such parallelized hardware, developers must design and implement parallel software. However, there are several difficulties in developing a parallelized program [3]. First, developers must have the ability to design parallel programs properly. They should be able to design and implement a structure that distributes tasks to multithreads and aggregates results.

Manuscript Received: May. 28, 2020 / Revised: June. 2, 2020 / Accepted: June. 9, 2020
Corresponding Author: wcshin@skuniv.ac.kr
Tel: +82-2-940-7755
Professor, Dept. of Computer Science, Seokyeong University, Korea

Second, it is necessary to understand various problems (competition status, synchronization, deadlock, etc.) that may occur in an environment operating in parallel, and to prepare a solution. Third, parallel programs can be difficult to maintain. Due to these problems, parallelization of the program remains a difficult topic for developers.

In the present paper, we introduce a method and a tool for automatically translating sequential programs into parallel programs. Using the method and tool, developers can easily parallelize existing programs and further learn how to parallelize them. The remainder of the paper is organized as follows.

In Section 2, related works are reviewed. In Section 3, we explain how to automatically change sequential source code to parallelized source code. Section 4 examines the development of a parallelization prototype tool that supports the proposed parallelization method, and shows the parallelized code through an example. In Section 5, we analyze the execution results of sequential and parallel programs to show the effectiveness of the parallelization tool. Finally, Section 6 summarizes and concludes this paper.

2. Related Work

Many studies have been conducted to parallelize existing sequential programs. Representative studies include Par4All, Cetus, iPat/OMP, AutoPar, and Stanford university intermediate format (SUIF).

Par4All aims to achieve the migration of software to multi-core and other parallel processors, as well as to accelerating processors such as GPU [4]. It is based on multiple components, including the source-to-source compiler framework PIPS [5]. Par4All takes C programs as input and generates OpenMP, CUDA and OpenCL programs. It also can transform FORTRAN programs to OpenMP. Cetus as a source-to-source C compiler written in Java and maintained at Purdue University. Cetus enables automatic parallelization by using data dependence analysis with the Banerjee-Wolfe inequalities, array and scalar privatization, reduction-variable recognition, and induction-variable substitution [6]. iPat/OMP is an interactive parallelization assistance tool for C language. This tool was implemented using EmacsLisp, Java, and shell scripts. It provides functions related to parallelism of programs, such as selecting a target portion of the program, invoking an assistance command, and modifying the program, in the source code editor (Emacs) environment [7].

AutoPar can automatically insert OpenMP pragmas into input serial C/C++ codes. It uses a source-to-source compiler infrastructure, ROSE [8], to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization [9]. SUIF is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. The SUIF parallelizer translates sequential FORTRAN/C programs into parallel code for shared address space machines. Loops containing reductions can be parallelized with simple synchronization [10]. In addition, research on parallel system development in a distributed environment was conducted [11].

Most of these studies, with the exception of AutoPar, are focused on automatic parallelization of C and FORTRAN programs. In the case of AutoPar that supports C++, parallel automation is limited due to the polymorphism features of the C++ language. The parallel automation method proposed in this paper supports the C/C++ language. In addition, this study does not focus on fully automatic parallelization, but focuses on verifying parallelization conditions and giving detailed advice to developers if parallelization is not possible.

3. Automatic Parallelization Method

Parallelism can be divided into task parallelism and loop parallelism depending on the level. This study focuses on loop parallelism. It is loop parallelism that can be easily parallelized without changing the structure of the existing sequential source code. In loop parallelism, the independence and synchronization of variables

across multiple iterations of the loop must be guaranteed. It is difficult to check the independence of variables used in each loop iteration only by static analysis of the source code. Moreover, it is impossible to completely analyze the side-effect of loop execution code due to pointer operation or polymorphism.

This study does not analyze the side-effects caused by pointer operations and function calls in the loop, and complements it by providing the developer with a detailed parallelization guide generated through code analysis. In this study, OpenMP and PPL were used as parallel frameworks, both of which have the advantage of being easy to parallelize while minimizing changes to existing code. They also help developers reduce the burden of parallelism learning and allow parallel code to be written concisely. Developers can choose which framework to apply when running the parallelization tool.

3.1 Parallel Framework

3.1.1 OpenMP

OpenMP is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN [12]. Since the first release of OpenMP for FORTRAN in October 1997, version 5.0 was most recently released in November 2018.

One of the great things about OpenMP is that it's easy. It is easy to convert an existing program into parallel code, and it can reduce the burden of parallelization among team members. For this reason, many automatic parallelization studies use OpenMP. For example, as shown in Figure 1, if you add the "#pragma omp parallel for" directive before the for-loop, the sequential image distortion correction code is executed as a parallel program.

```
void MappingMgr::map(char *srcBuffer, char *targetBuffer, float *mappingTable) {
#pragma omp parallel for
    for (int col = 0; col < _imageWidth; ++col) {
        uint32_t *tptr = ((uint32_t *)targetBuffer) + col;
        for (int row = 0; row < _imageHeight; ++row, tptr += _imageWidth)
            *tptr = getBlendedPixel(srcBuffer, mappingTable, col, row);
    }
}
```

Figure 1. Image distortion correction code with OpenMP

3.1.2 Parallel Pattern Language

The PPL is a Microsoft C++ library that provides features for multicore programming. PPL is provided in the form of a C++ template library, and is similar to standard template library (STL). It provides generic algorithms and containers in a style similar to STL, and is well compatible with existing STL algorithms and containers [13]. The sequential loop can be made into a parallel loop by replacing the "for-loop" with a "parallel_for" as shown in Figure 2.

```
void MappingMgr::map(char *srcBuffer, char *targetBuffer, float *mappingTable) {
    Concurrency::parallel_for(0, _imageWidth, [&](int col) {
        uint32_t *tptr = ((uint32_t *)targetBuffer) + col;
        for (int row = 0; row < _imageHeight; ++row, tptr += _imageWidth)
            *tptr = getBlendedPixel(srcBuffer, mappingTable, col, row);
    });
}
```

Figure 2. Image distortion correction code with PPL

3.2 Parallelization of Loop Statements

The easiest way to change an existing program written sequentially into a parallel program is for-loop statement parallelization. In order to parallelize for-loop statements, the form of the statement must be canonical. For example, in the sentence "for(*init-expr; test-expr; incr-expr*) { *exec-stmts* }", the condition for canonical form must be satisfied for each part [12]. Also, in order for each loop iteration to be executed in parallel, there should be no race condition or data synchronization problem in *exec-stmts*. To identify this problem, analyze the variables used in *exec-stmts*. To parallelize sequential for-loop statements, the following conditions must be satisfied.

for(*init-expr* ; *test-expr* ; *incr-expr*) { *exec-stmts* }

[Condition 1] The for-loop statement must conform to the canonical form.

The 3 parts – *init-expr, test-expr, incr-expr* – in the for-loop statement must satisfy the canonical form condition. If the for statement is canonical, the number of iterations is determined at the first execution. Canonical form conditions are described in detail in [12].

[Condition 2] The *exec-stmts* of each iteration should not affect the execution of other iterations.

Since side-effects generated by execution of each iteration can affect the execution of other iterations, the independence between each iteration is confirmed by checking the scope and read/write operations of variables used in *exec-stmts*. The procedure to check is as follows.

(a) Let *EV* is the set of all external variables (global variables, external local variables) used in *exec-stmts*.
- $EV = \{ x \mid x \in GV \vee x \in ELV \}$
  - *GV* : a set of global variables used in *exec-stmts.*
  - *ELV* : a set of external local variables used in *exec-stmts.*

(b) Each variable of *EV* is included in *WS* if it is used as L-value in *exec-stmts*, and in *RS* if it is used as R-value. At this time, the variable whose value is changed by increment and decrement operators (++/--), or compound assignment operators (+=, -=, etc.) is not included in *RS*, but only in *WS*.
- $RS = \{ x \mid x \in EV \wedge \text{isReadAccess}(x) \}$
- $WS = \{ x \mid x \in EV \wedge \text{isWriteAccess}(x) \}$
  - isReadAccess(x) : true if x is used as R-value in *exec-stmts*.
  - isWriteAccess(x) : true if x is used as L-value in *exec-stmts*.

(c) If a variable x other than an array is included in *WS* and *RS* at the same time, it cannot be parallelized because interdependencies between iterations may occur. Therefore, the following conditions must be satisfied for loop parallelization.
- $\forall x \in WS[(\text{not isArray}(x)) \rightarrow x \notin RS]$
  - isArray(x) : true if x is array.

(d) If an array is used in *exec-stmts*, the race condition is analyzed based on the variable *i* used in loop iteration. If *i* is used as the index of array A, A[*i*] is regarded as an independent variable for each iteration. However, if A[*i*+1] or A[*i*-1] is used together with A[*i*], race conditions between iterations may occur. Therefore, in the case of an array, the following conditions must be satisfied. (In the case of an n-dimensional array, conditions can be extended and specified in a similar way.)
- $\forall x \in WS[(\text{isArray}(x) \wedge (x=<A, index_1>) \wedge (index_1 \in \text{expr}(i))) \rightarrow$
  $\text{not } (\exists y \in (WS \cup RS)[\text{isArray}(y) \wedge (y=<A, index_2>) \wedge (index_2 \in \text{expr}(i)) \wedge (index_2 \neq index_1)])]$

- *i* : variable used for loop iteration.
- expr(*i*) : set of all expressions containing *i*.

[Condition 3] Synchronization is required for external variables shared by each iteration of the loop.

If an external variable is used only as an L-value in *exec-stmts*, synchronization is necessary to prevent simultaneous access to multiple threads.

- $\forall x \in WS$[(not isArray(x)) → reqSync(x)]
  - reqSync(x) : Synchronization code generation is required when parallelizing.

For-loop statements that satisfy Condition 1 and Condition 2 can be parallelized. Condition 3 defines a variable that requires synchronization when parallelizing for-loop statements that satisfy the conditions 1 and 2. If each condition is not satisfied, the help for the problem is output, and the developer learns the concept of parallelism and helps the developer to perform the parallelism correctly. The above conditions do not completely check for problems that may occur when parallelizing. For example, if the pointer operation is used as in the example shown in Figure 1, or if a side-effect occurs due to a virtual function call, it is not possible to check whether a race condition will occur by static analysis alone. In this case, it is left to the developer to check the occurrence of a race condition related to the pointer operation and polymorphism.

4. Development of Parallelization Tools

In the present study, we developed a prototype tool that supports the parallelization method presented in this paper. The development environment of the tools is as follows.

- Software
  - Microsoft Windows 10
  - Visual Studio 2017 C++ (x64)
  - LibClang (based on LLVM ver.10.0.0)
- Hardware
  - CPU: Intel Skylake i7 (3.6GHz, 4 Cores)

LibClang is a C/C++ parsing library based on the LLVM compiler [14]. This parallelization tool analyzes the source code after converting C++ file to C++ AST using libClang.

```
double multiply_matrix(double** m1, double** m2, double** result, int size) {
    double sum = 0.0;
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; j++) {
            double temp = 0;
            for (int k = 0; k < size; k++)
                temp += m1[i][k] * m2[k][j];
            result[i][j] = temp;
            sum += temp;
        }
    }
    return sum;
}
```

Figure 3. Example C++ code

To parallelize the sequential matrix multiplication function code written in C++, shown in Figure 3, run this parallelization tool as shown in Figure 4.

```
> cpp_parallel  -[openmp | ppl]  filename  linenum
```

Figure 4. Command to execute the proposed automatic parallelization tool

According to the selected value (openmp or ppl), parallelized code for the corresponding parallel framework is generated. When parallelization is executed with OpenMP, the execution screen is shown in Figure 5. As the parallelization work progresses, a description of parallelism is output at each step. If a problem occurs during parallelization, advice to solve the problem is also displayed. For example, when a race condition occurs, the name of the variable that caused the problem is displayed. It is also advised to change the variable to a local variable in the loop statement so that it can be used independently for each iteration.

```
> cpp_parallel  -openmp  matrix.cpp  54
[C++ Parallelizer]
Checking parallelization conditions.
The for-loop statement (matrix.cpp, line 54) is a canonical form.
There are no variables in the loop execution statement that can cause race conditions.
The variable "sum" needs synchronization. Synchronization statements will be inserted.
Generating parallelized code.
Code parallelization succeeded. (Generated file: matrix_parallel.cpp)
```

Figure 5. Example C++ code

The example code parallelized to OpenMP and PPL is shown in Figure 6 and Figure 7, respectively. For the variable *sum* that needs to be synchronized, the "#pragma omp atomic" directive is inserted in OpenMP. In the case of PPL, for performance improvement, the tool declares a new combinable variable *sum2*, uses it to obtain the value of each thread, and then adds all of these values after the parallelization is complete, and moves the value to the *sum* variable.

```
double multiply_matrix(double** m1, double** m2,
double** result, int size) {
   double sum = 0.0;
#pragma omp parallel for
   for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; j++) {
             double temp = 0;
             for (int k = 0; k < size; k++)
                  temp += m1[i][k] * m2[k][j];
             result[i][j] = temp;
#pragma omp atomic
             sum += temp;
        }
   }
   return sum;
}
```

Figure 6. Parallelized code based on OpenMP

```
double multiply_matrix(double **m1, double **m2,
double **result, int size) {
   double sum = 0.0;
   combinable<double> sum2;
   sum2.local() = sum;
   Concurrency::parallel_for(0, size, [&](int i) {
        for (int j = 0; j < size; j++) {
             double temp = 0;
             for (int k = 0; k < size; k++)
                  temp += m1[i][k] * m2[k][j];
             result[i][j] = temp;
             sum2.local() += temp;
        }
   });
   sum = sum2.combine(plus<double>());
   return sum;
}
```

Figure 7. Parallelized code based on PPL

5. Analysis of Execution Results

The results of executing the sequential code shown in Figure 3 and the parallelized code shown in Figure 6 and Figure 7 while changing the size of the matrix are shown in Table 1. Analyzing the results in Table 1, it can be seen that the performance of parallelized code is worse than that of sequential code in the case of small computational tasks. In particular, the performance of PPL was relatively poor. For matrix calculations over 1000x1000, parallelized code showed a performance improvement of 198% to 389% over sequential code. It can be seen that the performance increases by a multiple of the number of CPU cores as the calculation workload increases.

Table 1. Comparison of execution time

Size of matrix	Sequential code	OpenMP code		PPL code	
	Exec. Time (a)	Exec. Time (b)	Performance ratio (a/b)	Exec. Time (c)	Performance ratio (a/c)
100x100	1.01	1.41	71.67%	3.97	25.47%
1000x1000	2,003	1,008	198.70%	1,005	199.21%
1500x1500	9,660	3,689	261.84%	3,669	263.30%
2000x2000	36,549	9,482	385.45%	9,375	389.84%

Figure 8 shows the execution result as a chart. Similar to the research by Shin, PPL shows slightly better performance than OpenMP [15].

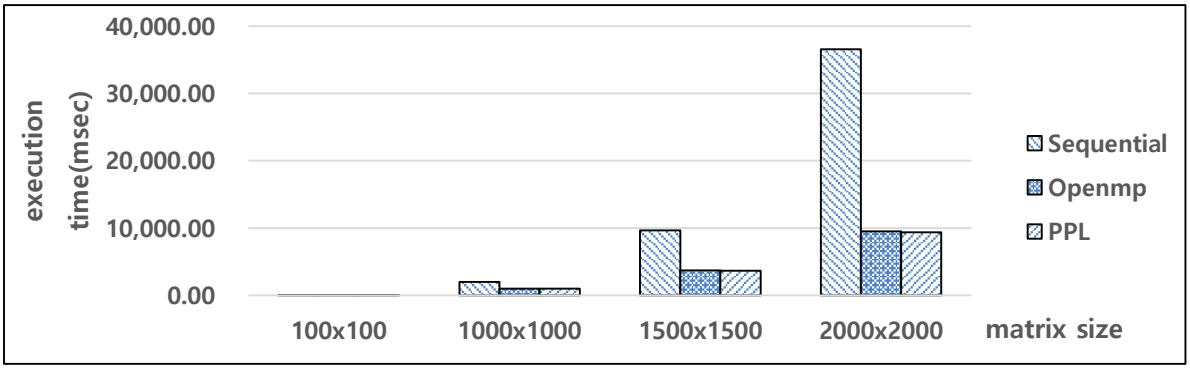

Figure 8. Performance comparison chart between sequential and parallelized code

6. Conclusion

In this paper, we present a method for automatically converting sequential C/C++ programs into parallel programs, and describe parallelization tools to support them. It supports PPL framework that can be optimized for Windows environment with OpenMP that can be used universally.

Full parallel automation is difficult due to dynamic features such as pointer operations and polymorphism in the C/C++ language. Rather than focusing on fully automatic parallelization, this study focused on verifying the conditions of parallelization and providing advice to developers in detail if parallelization is not possible.

When the computational workload is large enough, it can be seen that the automatic parallelized program by the tool developed in this study is faster than the sequential program by a multiple of the number of CPU

cores. Using these methods and tools, developers can easily parallelize sequential programs and become more familiar with program parallelism.

Acknowledgement

This Research was supported by Seokyeong University in 2020.

References

[1] Y.H. Jung, OpenMP Parallel Programming, Freelec Publishing, Jan. 2011.
[2] NVIDIA, Tesla V100 Specification, *https://www.nvidia.com/en-us/data-center/v100*
[3] B. Chapman, G. Jost, and R. Pas, "Using OpenMP Portable Shared Memory Parallel Programming," USA, MIT Press, 2007. DOI: http://doi.org/10.5860/choice.46-0930
[4] SILKAN, The Par4All, *https://github.com/Par4All/par4a*
[5] The PIPS Team, Overview of PIPS, *https://pips4u.org*
[6] D. Chirag, H.S. Bae, S.J. Min, S.Y. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," IEEE Computer, vol. 42, no. 12, pp 36-42, Dec. 2009. DOI: http://doi.org/10.1109/MC.2009.385
[7] M. Ishihara, H. Honda, and M. Sato, "Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: iPat/OMP," IEICE - Transactions on Information and Systems, Feb. 2006. DOI: https://doi.org/10.1093/ietisy/e89-d.2.399
[8] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," Proceedings of Conference on Parallel Compilers, 2000.
[9] C.H. Liao, D.J. Quinlan, J.J. Willcock, and T. Panas, "Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions," Journal of Parallel Programming, Oct. 2010. DOI: http://doi.org/10.1007/s10766-010-0139-0
[10] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.W. Liao, C.W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," ACM SIGPLAN Notices 29(12):31-37, April 1996. DOI: http://doi.org/10.1145/193209.193217
[11] Y.H. Ko, G.S. Heo, and S.H. Lee, "A Study on Distributed System Construction and Numerical Calculation Using Raspberry Pi," International Journal of Advanced Smart Convergence, vol. 8, no. 4, pp 194-199, 2019.
[12] OpenMP Architecture Review Board, OpenMP Application Programming Interface version 5.0, OpenMP ARB, Nov. 2018.
[13] K.J. Kim, VC++ parallel programming using PPL, Hanbit Media, Mar. 2014.
[14] S. Schaub and B.A. Malloy, "Comprehensive analysis of C++ applications using the libClang API," 23rd International Conference on Software Engineering and Data Engineering, pp. 131-136, 2014.
[15] W.C. Shin, "Performance Comparison of Parallel Programming Frameworks in Digital Image Transformation," International Journal of Internet, Broadcasting and Communication, vol. 11, no. 3, pp 1-7, Aug. 2019.