Prototyping for a Parallel Programming Tool

Kyoko Iwasawa
Department of Computer Science
Takushoku University
Hachioji Tokyo, Japan
e-mail: kiwasawa@cs.takushoku-u.ac.jp

***Abstract*—We propose a tool to enable even beginners in parallel processing to develop a parallelization program using Open Multi-Processing (OpenMP) directives. Our proposed tool is characterized by its analysis of source programs for C and OpenMP directives written by users and its display of parallel structure diagrams. Further, the discovery of source program bugs is facilitated by the static analysis of interactive data access regions and decisions on the feasibility of parallelization using these parallel structure diagrams. While our proposed tool currently handles only basic OpenMP directives, our aim is to improve the analysis of parallel structure diagrams by including more complex simultaneous processing and more precise data access.**

Keywords-parallel programming; OpenMP directive; data flow analysis.

I. INTRODUCTION

While the recent years have seen a proliferation in systems capable of parallel execution, including multicore and General Purpose computing on Graphics Processing Units (GPGPU), in general, the development of programs for parallel execution is difficult. While this is also the case with algorithm development, writing parallel processing code in an editing environment for the coding of sequential processing easily produces errors. Further, it is difficult to identify the errors, because in parallel programs the execution results are not reproducible.

Therefore, we propose a programming environment particularly for beginners in parallel processing, using a parallel structure that is easily understood visually and also statically analyses the feasibility of parallel execution from the execution statement data access regions during program editing. We are developing the prototype of this tool.

Open Multi-Processing (OpenMP) is an application programming interface that supports multi-platform shared memory multiprocessing programming by the OpenMP Architecture Review Boards [1]. The details of OpenMP spec are written in [2] and [3]. There are several tools for OpenMP programming. [4] and [5] are integrated tools for OpenMP programming, which include compiler and parallel execution environments. They have various functions and can be somewhat difficult for beginners of parallel programming. We simplify the analyzing method in [6] and [7] because our proposed tool does not generate parallel object code, but suggests user appropriate directives for parallelization.

The rest of paper is structured as follows. Section II presents the overview of the proposed tool; Section III describes the parallel structure diagrammatic display; Section IV explains the access region analytical method; Section V explains the parallelization feasibility decision method. Finally, we conclude and present the future issues in Section VI.

II. PROPOSED TOOL OVERVIEW

Our proposed tool is an environment for the C programming language used in creating and editing programs that give parallelization directions using OpenMP directives. It has the following three main functions.

(1) OpenMP directive analysis
(2) Parallel structure graph display
(3) Interactive, static data flow analysis, and parallelization feasibility decisions.

In addition, it display the structure written in OpenMP in an easy understood manner for users not accustomed to parallel processing, as well as for beginners to perform debugging by displaying static analytical results interactively.

Figure 1 shows the overall proposed tool structure.

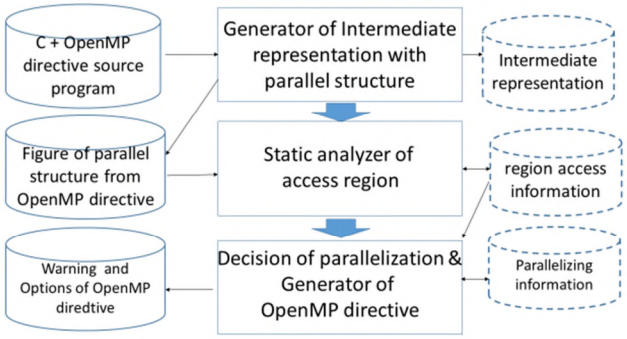

Figure 1. Overall tool structure

A source program with an OpenMP directive parallel execution direction is entered into a C program to analyze the C programming language execution directions and OpenMP directives. Subsequently, they are joined in an intermediate representation. This is formed and displayed as the parallel structure diagram in Figure 3. In this diagram, the user selects the quadrangle in the execution direction and the elliptical shape in the parallel execution direction to decide the data access region and parallel execution feasibility.

 ISBN: 978-1-61208-666-8

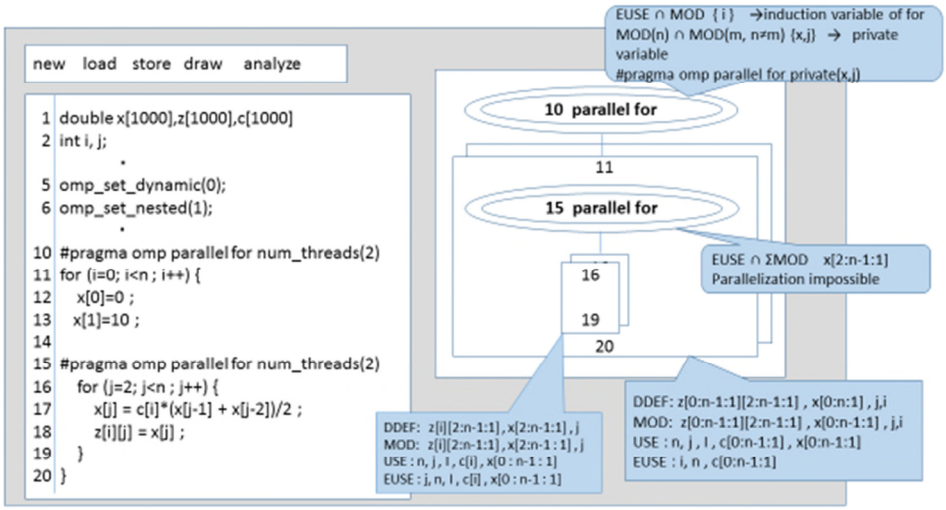

Figure 2. Display screen

Figure 2 shows the display screen and an analytical example. In the editing window on the left, the user performs the parallelization program coding using the C programming language script and OpenMP directives. The command "draw" is selected for the tool to display a parallel structure diagram on the left. The details of this parallel structure diagram are presented in Section III. The command 'analyze' is selected to enable the selection of the diagram quadrangle and elliptical shape (line number and OpenMP directive). Selecting one of these displays the parallel block access regions for that OpenMP directive and the parallelization decision.

III. GRAPH DRAWING OF OPENMP DIRECTIVE ANALYTICAL RESULTS

The tool analyzes syntax and context of OpenMP directives in the C source program, and these directives are reflected in the intermediate representation. This displays the diagram expressing the parallel structure in a graph from this intermediate representation. This is a graph structure with quadrangles expressing the parallel execution unit and ellipses expressing parallel execution direction as nodes.

Quadrangles do not display the execution directions merely by inserting the first and last direction numbers. Ellipses have line numbers and OpenMP directives as labels.

Although there are many OpenMP directives, the current parallel structure diagrams are expressing only for the following three basic types thought necessary for beginners as subjects of analysis.

(1)#pragma omp parallel

(2)#pragma omp parallel for

(3)#pragma omp parallel sections and #pragma omp section

The "parallel for" for the do-all-type parallel processing is expressed in double ellipses, and their loops are expressed by overlapping quadrangles. The "parallel sections" that express parallel-case type parallel processing are single ellipses. The nested parallel execution is expressed by drawing ellipses and quadrangles in other quadrangles.

The requirements of parallel structure graph to express directives are the following:

- To distinguish between the execution of same statements in parallel for the number of threads (#pragma omp parallel) and the execution of different statements in parallel (#pragma omp parallel for, #pragma omp parallel sections).
- To arrange statements to be executed simultaneously, side by side.
- To arrange sequential statements vertically and clarify the order of execution by using connected line.
- To show the synchronization point.
- To disclose parallel nesting structure.

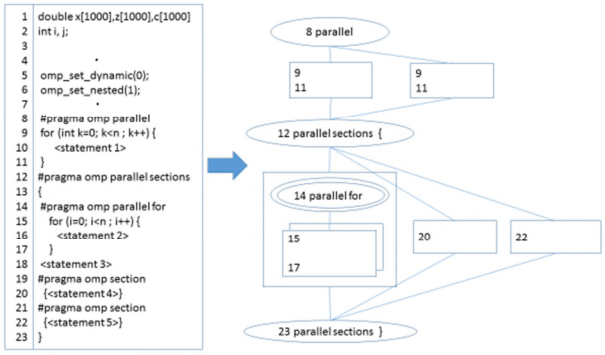

Figure 3. Source program and parallel structure diagrams

Figure 3 shows an example of source program and parallel structure graph. Since the eighth line is a "parallel" directive, it directs to execute the entire following for-loop parallel by the thread number. On the other hand, the 14th line is a "parallel for" directive that divides the loop

 ISBN: 978-1-61208-666-8

repetition and directs to execute by dividing them in a parallel manner. This is expressed with a double ellipse directive and overlapping quadrangles such that the difference can be intuitively understood. The 12th line is a "parallel section" directive, and the quadrangles are able to execute parallel and the synchronizing point for the "}" in the 23rd line to clarify its scope. While internal statements of each quadrangle and the overlapping quadrangle execute sequentially, when there is any parallel directive graph shows nested parallel execution.

IV. STATIC ANALYSIS OF DATA ACCESS REGION

When the user selects the quadrangle in a generated parallel structure graph from an OpenMP directive analysis, the tool finds and displays the data access region by its execution. Additionally, when user selects the ellipse that expresses parallel directive, the tools decides the parallel execution feasibility. An example is shown in Figure 2.

Access region analysis to decide parallelization feasibility analyses what regions are accessed in what order according to a control flow.

A. Access Types

Four data access types are available:

- Possible use (USE)
  Data that might be used within a certain scope (flow graph pass)
- Possible exposed use (EUSE: Exposed USE)
  Data that might be used within a certain scope before definition (flow graph pass)
- Possible definitions (MOD: MODified)
  Data that might be updated within a certain scope before definition (flow graph pass)
- Definitely defined (DDEF: Definitely Defined)
  Data that is definitely updated within a certain scope (flow graph pass)

The 'flow graph pass' above widens the scope of analysis to the parallelization block through the process of one statement → basic block → loop i-th iteration → all repetitions loop → outer loop.

While the 'possible use' and 'possible definitions' are control flow insensitive, 'possible exposed use' and 'definitely defined' are control flow sensitive. These regions are related as follows:

Possible use ⊆ Possible exposed use

Possible definitions ⊆ Definitely defined

As understood from the analytical methods in Section V, the 'possible use' and 'possible definitions' are required to guarantee safety.

B. Method 1: [fusing]

In the if-then-else structure, when node 1 is 'then' and node 2 is 'else', the tool integrates the access regions as in Figure 4 (+ is union and * is intersection).

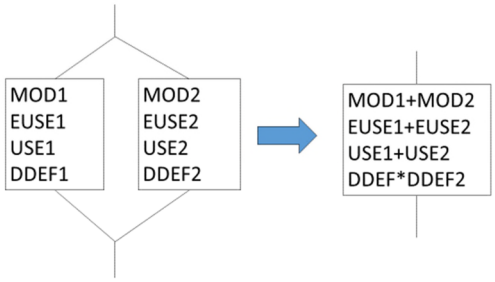

Figure 4. Access region integration (conditional branches)

C. Method 2: [join]

After fusing the if-then-else structure, the nodes sometimes line up in a row. Node 1 is the priority node and node 2 is the next node. The tool integrates the access region as in Figure 5 (- is the difference set excluding the intersection set from the first operand).

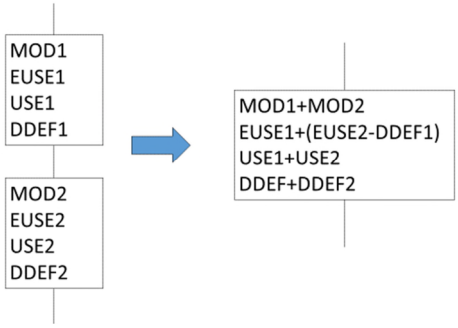

Figure 5 . Access region integration (connection)

D. Method 3: [expansion of loops]

Concerning loops, the access region of the i-th iteration is analysed by *Method 1* and *Method 2* and the access region of the entire loop is analysed, as shown in Figure 6. The information of data access in a loop is expanded.

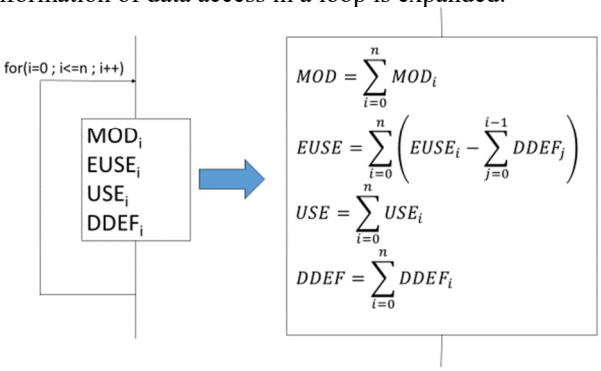

Figure 6. Access region expansion

The balloons indicating the quadrangles in Figure 2 contains an example of the analysis results.

V. PARALLEL EXECUTION FEASIBILITY DECISION

By using the parallel structure graph, the user knows feasibility of parallel execution. The tool decides whether each iteration of a loop can be executed independently for the do-all type, and for the parallel-case type the tool decides whether the parallel blocks surrounded by the section directives can be executed independently. When dependency that impedes parallel execution occurs, the tool displays it as the reason of impossible of parallelization. In some cases, the usage of the reduction instructions and the privatization of variables are confirmed.

Decisions of parallel execution are conducted using access regions as follows.

A. do-all Type

The entire loop-accessed region is checked for reliance upon the following three types of loop-carried data dependence.

$$\forall_i (0 \le i \le n, n: loop\ iteration\ number)\ EUSE_i \cap \sum_{j=0}^{i-1} MOD_j \neq \emptyset \quad (1)$$

→ loop carried flow dependence

$$\forall_i (0 \le i \le n, n: loop\ iteration\ number)\ MOD_i \cap \sum_{j=0}^{i-1} (USE_j - EUSE_j) \neq \emptyset \quad (2)$$

→ loop carried anti dependence

$$\forall_i (0 \le i \le n, n: loop\ iteration\ number) MOD_i \cap \sum_{j=0}^{i-1} MOD_j \neq \emptyset \quad (3)$$

→ loop carried output dependence

When condition (1) is satisfied, confirming data that is detected causes loop carried data dependence. It becomes parallelization impeding factor. When conditions (2) and (3) are satisfied, confirming data that is detected causes loop carried anti and output data dependence. In this case, parallelization might be possible by privatisation of these confirming data. The tool recommends users to add private clause to parallel for directive.

The balloon indicating the ellipse with the parallelization direction in Figure 2 contains an example of the analysis results.

B. parallel-case Type

In a parallel-case-type parallel processing, whether all quadrangles that are connecting the parallel section ellipses can be independently executed is decided as follows. They are similar to do-all case. If defined area of a given section is not overlapping with defined and used regions of any other sections, these sections can be executed independently. The overlapping of regions causes memory hazard.

$$\forall_i (0 \le i \le n, n: section\ number)\ EUSE_i \cap \sum_{j=0}^{n} MOD_{j \neq i} \neq \emptyset \quad (4)$$

→ flow dependence

$$\forall_i (0 \le i \le n, n: section\ number)\ MOD_i \cap \sum_{j=0}^{n} (USE_{j \neq i} - EUSE_{j \neq i}) \neq \emptyset \quad (5)$$

→ anti dependence

$$\forall_i (0 \le i \le n, n: section\ number)\ MOD_i \cap \sum_{j=0}^{n} MOD_{j \neq i} \neq \emptyset \quad (6)$$

→ output dependence

The case of condition (4) is satisfied and there is flow dependence, which inhibit parallel execution without any synchronization. When condition (5) or condition (6) is satisfied, the tool recommend user to privatize confirming data that is detected.

VI. CONCLUSION

In a structure as presented herein, providing a parallel program development environment allows the meaning of the written OpenMP directives to be easily understood and mistakes in directives to be easily recognized by beginners not accustomed to parallel processing. Further, this enables the detection and correction of errors peculiar to parallel processing at an early development stage for an accurate static analysis. Inserting OpenMP directives into C programs, such as parallel structures graph enables the easy understanding of parallel structure mistakes and missing synchronous processes because when necessary OpenMP directive is missed out the graph does not have parallel structure. Then the tool makes comments reason why the tool cannot make parallel structure.

Currently, the prototype of the proposed tool is under developing. The GUI specifications have developed as they are considered. We are going to connect the result of static analysis to parallel structure graph. In the future, we would like to increase the types of OpenMP directives for analysis, display complex synchronous processes in an easily understood manner, and provide appropriate advice from the analytical results. Once the parallel structure specifications are established, we would like for users to draw parallel structure graph, input execution statements in them, and for the tool to generate C and OpenMP source programs.

REFERENCES

[1] http://www.openmp.org/, “HOME OpenMP”, 2018.03.19

[2] B. Chapman, G. Jost, and R. Van Def Pas, “Using OpenMP: Portable Shared Memory Parallel Programming”, MIT Press, 2008

[3] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, “Parallel Programming in OpenMP”, Morgan Kaufmann, 2000

[4] O. Harnandez, C. Liao, and B.Chapman, “Dragon: A Static and Dynamic Tool for OpenMP”, Internationa Workshop on OpenMP Applocation and Tools, pp.54-66, 2004

 ISBN: 978-1-61208-666-8

[5] https://pm.bsc.es/ompss, “Programming Models@BSC”, 2018.03.20

[6] http://coins-compiler.osdn.jp/international/index.html, “COINS project”, 2018.03.19

[7] K. Iwasawa, Automatic Parallelizing “Method of Loops by Conditional Region Analysis”, Proceedings of the 16th IASTED International Conference Applied Informatics, pp.310-313, 1998

[8] T., Watanabe, T. Fujise, K. Mori, K. Iwasawa, and I. Nakata, “Design assists for embedded systems in the COINS Compiler Infrastructure”, Proceedings of the 10th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.60-09, 2007