Scalable Thread Visualization for Debugging Data Races in OpenMP Programs

Young-Joo Kim, Jae-Seon Lim, and Yong-Kee Jun*

Gyeongsang National University
Jinju, 660-701 South Korea
{yjkim,dember99,jun}@gnu.ac.kr

Abstract. It is important to debug unintended data races in OpenMP programs efficiently, because such programs are often complex and long-running. Previous tools for detecting the races does not provide any effective facility for understanding the complexity of threads involved in the reported races. This paper presents a thread visualization tool to present a partial order of threads executed in the traced programs with a scalable graph of abstract threads upon a three-dimensional cone. The scalable thread visualization is proved to be effective in debugging races using a set of synthetic programs.

Keywords: OpenMP programs, data race debugging, scalable thread visualization, three-dimensional visualization.

1 Introduction

OpenMP program model [14] is an industry standard of parallel programming model which supports Fortran and C language. However, it is still more difficult to debug OpenMP programs than sequential programs, because unexpected non-deterministic executions may be incurred from unintended data races [12] and such programs are often complex and long-running with a huge number of threads and accesses to shared variables. Thus these problems make users still difficult to debug races efficiently.

Thread Checker [4,5,16] of Intel Corporation is a unique tool to detect threading errors including data races in the relaxed sequential program which is a kind of programs parallelized only with OpenMP directives. During a sequentially monitored execution, Thread Checker projects the parallel memory traces of logical threads derived from the annotated sequential memory trace, and detects threading errors including races while every instruction in the program is executed. But this tool does not provide any effective facility for understanding the complexity of threads involved in the reported races.

This paper presents a thread visualization tool to represent the partial order of threads in the traced OpenMP programs with a scalable graph of abstract

* *Corresponding author*: In Gyeongsang National University, he is also involved in the Research Institute of Computer and Information Communication (RICIC).

C. Cérin and K.-C. Li (Eds.): GPC 2007, LNCS 4459, pp. 310–321, 2007.
© Springer-Verlag Berlin Heidelberg 2007

threads upon a three-dimensional cone. We consider OpenMP programs which may include critical sections and nested parallelism. The visualization on three-dimensional cone makes it overcome the limitation of visual space on one plane and use an execution graph [1,11] to represent effectively a partial order over threads. This tool solves the visual complexity using the abstract visualization which replaces a set of events with an abstract symbol and provides the thread information which is traced by RaceStand [9], an on-the-fly race detection tool. The abstraction concept reduces the space complexity of thread visualization and helps programmers to understand the complex structure of threads effectively. We experimented this visualization tool on a Windows-XP computer based on Pentium-4 using Visual C++ and OpenGL libraries.

Section 2 illustrates data races that occur in OpenMP programs, indicates the problems of the previous tool for debugging races. Section 3 presents the design concepts of our scalable thread-visualization tool. Section 4 shows the screen-shots of the implemented tool using a set of synthetic programs to demonstrate that scalable thread visualization is effective to debugging races efficiently. The last section includes conclusions and future work.

2 Background

This section illustrates data races which occur in OpenMP programs and introduces the problem of the previous tools, Thread Checker and RaceStand, that detect data races.

2.1 OpenMP Program

OpenMP [14] is an industry standard model of shared memory with a set of directives and libraries that extend standard C/C++ and Fortran 77/90. OpenMP can easily convert sequential programs into parallel programs using OpenMP directives, and can provide scalable parallel programs using the orphan directive to make coarse-grain parallelism. The OpenMP directives include parallelism directives and synchronization directives. The parallelism directives include "#pragma omp parallel for" for parallel loops and "#pragma omp section" for parallel sections. We consider the parallel loop as an example of parallelism. If there is no other loop contained in a loop body, the loop is called an *innermost* loop. Otherwise, it is called an *outer* loop. In a nested loop, an individual loop can be enclosed by many outer loops. The *nesting level* of an individual loop is equal to one plus the number of the enclosing outer loops. The nesting depth of a loop is the maximum nesting level of loops in the loop. The synchronization directives include "#pragma omp atomic," "#pragma omp barrier," and "#pragma omp critical" that control an execution order among threads. OpenMP also provides library functions and environment variables that can control run-time execution of programs. For example, two logical threads are created by "#pragma omp parellel" through line 11 and line 13 of Figure 1. Due to "#pragma omp for private(i, y, z)" of line 12, the created thread takes the specified job in the loop

```
10:  ···
11:  #pragma omp parallel
12:  #pragma omp for private (i,y,z)
13:  for (i=1 ; i < 3 ; i++) {
14:    if (i==1) { y = x + 2;
15:  #pragma omp critical(L1)
16:    z = x + 2; x = y + z;
17:    } else {
18:  #pragma omp critical(L1)
19:    x = 100; y = x + 1;
20:    }}
21:  printf("x value = %d ", x);
22:  ···
```

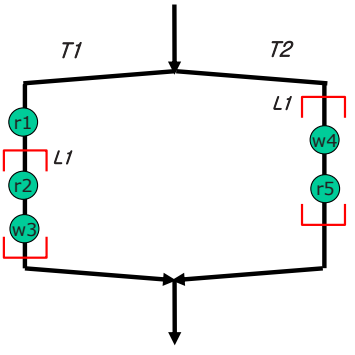

Fig. 1. An OpenMP Parallel Program and its POEG

body from line 14 to the brace of line 20, in which, the index variable i is a private variable used in each thread, and the integer variable x is a shared variable shared by the two threads.

Data races may occur in the program of Figure 1 during its program executions. First, we assume that the variable x has zero as an initial value. The statements of line 14, 15, and 16 are executed by the first thread of the two created threads and the statement of line 18 and 19 are executed by the second thread. Unintended races do not exist toward the variable x between line 16 and line 19, because these two blocks are protected as critical sections by "#pragma omp critical($L1$)." However, regarding the read access in the statement of line 14 and the write access to the shared variable x in the statement of line 19, the random speed of two threads may make the value of variable x in the statement of line 21 become 100 or 104 nondeterministically. It is because these two accesses are involved in a race which include at least one write access without proper inter-thread coordination for the accesses to the shared variable x.

The right of Figure 1 shows an execution instance of the program in Figure 1 by means of a directed acyclic graph called Partial Order Execution Graph (POEG) [1]. A vertex of POEG means a fork or join operation for parallel threads, and an arc started from a vertex represents a thread started from the vertex. The access r and w drawn with small disks upon the arcs represent a read and a write access which access a same shared variable. A number attached to each access indicates an observed order, and an arc segment delimited by the symbols $\{\sqcap, \sqcup\}$ means a critical section protected by the lock variable $L1$. With POEG, we can easily understand the partial order or happened-before relationship [10] of accesses occurred in an execution instance of programs. POEG of Figure 1 makes it easy to understand that $r1$ and $w4$ are involved in a race, because it shows that $r1$ in thread $T1$ and $w4$ in thread $T2$ are concurrent with each other, and $r1$ is not protected by any lock variable.

nsting levels	J2, J3, J2, J1
thread information	0: I, [11],–,0,–,–, , ,
	11:F,[1.1,1.2],1,1,-1,(0,2,1)
	11:F,[1.2,1.1],2,1,-1,(0,2,1)
	22:F,[1.2.1,1.1.2],1,2,-1,(0,2,1)
	24:J,[1.2.2,1.1.2],–,2,–,–, , ,
	28:C,[1.2.2,1.1.2],–,–,L1,–, , ,
	39:J,[1.2.2.2,1.1.2.2],–,2,–,–, , ,
	41:U,[1.2.2.2,1.1.2.2],–,–,L1,–, ,
	45:J,[1.2.2.2,1.2],–,1,–,–, , ,

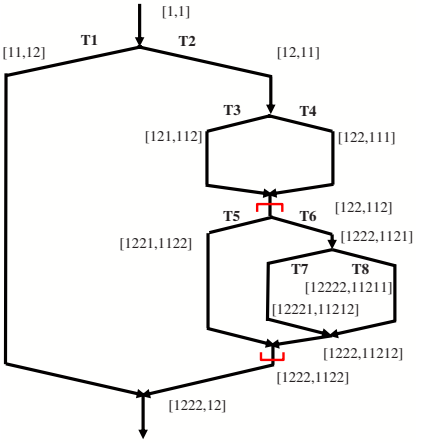

Fig. 2. An Example of RaceStand Traces and Labeling Information in POEG

2.2 Race Detection Tools

The projection technique of Thread Checker [4,5,16] for OpenMP programs collects execution information obtained during the compilation of program and checks data dependency detected during the sequential run-time of program. This technique is applied only to the relaxed sequential OpenMP programs [16] which provides only OpenMP directives for parallelism. Thread Checker detects races as follows. First, when the programs written in OpenMP directives are compiled by Intel C/C++ Compiler [3], a part of this tool integrated in the compiler modifies the programs to trace the information related to OpenMP directives and shared variables into an exclusive database. Second, when the complied program is executed sequentially, the tool uses the traced information in the database to check data dependency of accesses to shared variables whenever an OpenMP directive is located. Last, the tool reports the accesses as races if it satisfies an anti, flow, or output data dependency except an input data dependency.

Unfortunately, Thread Checker has some problems. First, although $r1$ and $w4$ are involved in a race in the POEG of Figure 1, this tool can not report the race because it ignores access $r1$ involved in the race. Second, this tool does not provide any effective information about the dynamic view of the detected races. This kind of reporting is difficult for users to understand the detected races and debug effectively OpenMP programs, because it does not provide any facility for understanding the complexity of threads involved in the reported races.

RaceStand [9] can verify the existence of races in OpenMP programs using a set of scalable thread-labeling techniques [2,13] and protocol techniques [2,11] for detecting races. The labeling techniques generate information called label for logical concurrency among the created threads during a program execution. A label is a unique identifier of thread, and is used to detect races because any

two labels can be compared to identify the logical concurrency between any two threads. The protocol techniques detect races by comparing the label of the current access with that of the previous accesses that are saved in a shared-data structure called *access history* whenever an access occurs in a thread. An access history consists of a set of mutually-concurrent accesses occurred in a program execution. These protocols guarantee to detect at least one race [12] if any in their corresponding model of programs. Unfortunately, RaceStand does not provide any effective information about the dynamic view of the reported races.

3 Scalable Thread Visualization

For a visual environment which can help users to debug races effectively using the additional information traced by RaceStand, this section presents two function modules for thread visualization and two abstraction concepts for scalable visualization.

3.1 Thread Visualization

Our tool visualizes a partial order of threads executed in the traced programs through a scalable graph of abstract threads upon a three-dimensional cone to help programmers to debug races intuitively. This tool requires the levels of nested parallelism and the thread information generated by RaceStand. The nesting levels can be traced whenever a join operation occurs in an execution. The thread information includes the thread labels generated whenever a parallel or synchronization directive is executed. The table of Figure 2 shows the information traced in an execution of OpenMP program captured with POEG in Figure 2. In the figure, the nesting depth is three since the nesting levels of $T1$ and $T2$ are one, the nesting levels of $T3$, $T4$, $T5$, and $T6$ are two, and the nesting levels of $T7$ and $T8$ are three. Each thread label in the right POEG of Figure 2 is a English-Hebrew (EH) label [13].

Our tool consists of two function modules: The *Cone Visualizer* and The *Thread Visualizer*. The Cone Visualizer parses the trace of nesting levels and then draws a three-dimensional cone by calculating the nesting depth and the number of multi-way loops which are defined as executed serially in a thread at each nesting level. The number of multi-way loops executed in a thread at a nesting level i is the number of 'J_i's generated by the thread, where J means a join operation and an integer i means a nesting level less than i. The maximum value of i is the nesting depth. The table of Figure 2 shows a trace of four nesting levels, by which the nesting depth is three because the maximum level is three. In the initial thread or $T6$, the number of multi-way loops is one, and the thread $T2$ executed two multi-way loops.

The Thread Visualizer parses the thread information and then draws the threads on the three-dimensional cone. The thread information consists of seven elements: source line number, event type, EH-label, loop index, nesting level, lock

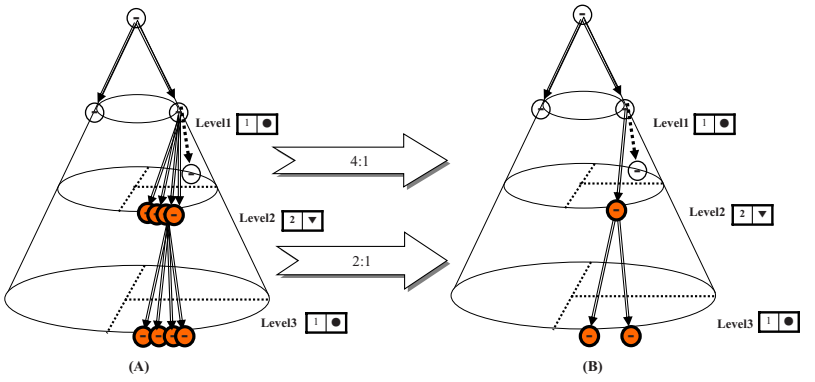

Fig. 3. The Abstract Visualization

variables, and for-statement information. The source line number identifies the source code location at which the threads occurred. The event type expresses a type of operations occurred in the execution: I-type for the initial thread, F-type for a fork operation, J-type for a join operation, C-type for a lock operation, and U-type for an unlock operation. An EH label is a thread label created by English-Hebrew Labeling scheme [13]. The table of Figure 2 shows an example trace of thread information.

3.2 Scalable Visualization

This section presents the concepts of *space abstraction* and *thread abstraction* for scalable three-dimensional visualization using the traced information. To illustrate an abstract visualization, we use the visualization information shown in POEG and the table of Figure 2.

The space of thread visualization is represented with a three-dimensional cone which is divided vertically as many layers as the nesting depth. Each nesting level is associated with a combo box which represents the number of loops executed by the thread in the upper nesting level. Figure 3(A) shows an example of the space abstraction. The first or third nesting level has only one loop and the second nesting level has two loops. The combo box for the second level allows to select one of the two loops as shown in Figure 3(A). the user can set the nesting depth at will. For example, if the user set the value of the nesting depth to five in the case of nesting levels ($J4$, $J3$, $J3$, $J2$, $J1$), the cone becomes divided into five layers. In this case, each combo box for the nesting level but the third has one loop. The combo box for the third nesting level has two loops, because $J3$ appears twice. The combo box for the fifth nesting level can not be created, because the information corresponding to the nesting level does not exist.

The threads at the same nesting level are visualized as circles on the same circumference of the corresponding cone layer with the optional vertical and horizontal abstraction. The vertical abstraction represents a thread which created

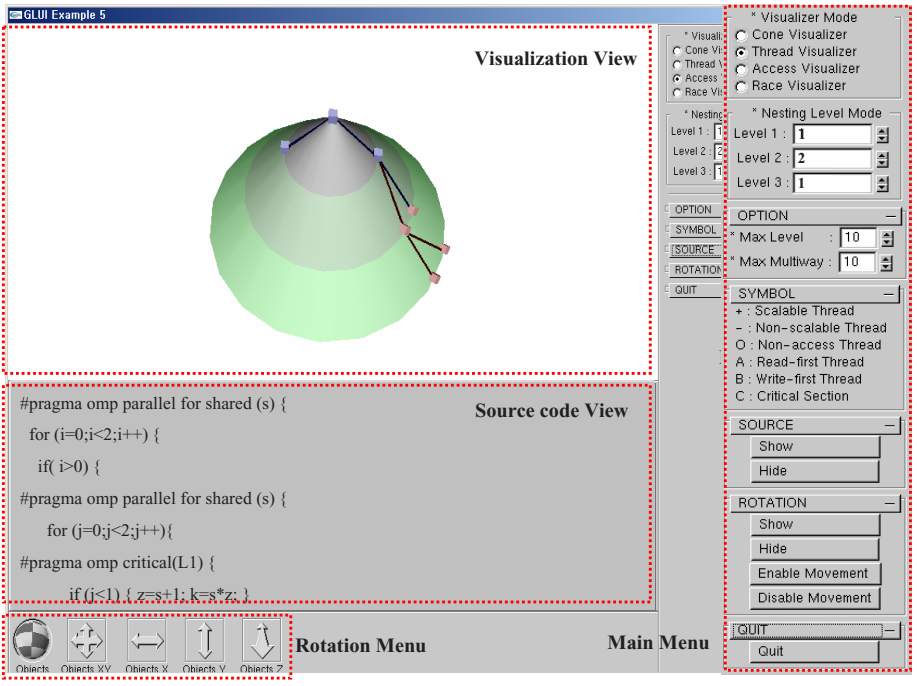

Fig. 4. The Overall Interface for Scalable Thread Visualization

child thread in the lower nesting levels with a special circle symbol. A parent thread can be represented with a symbol "+" or "-" inside a circle. The symbol "+" means that the parent thread has child threads which are not shown and the symbol "-" means that the parent thread has child threads which are drawn on the cone. A circle symbol which is colored and rounded by a thick line is an abstract thread which includes a critical section. Figure 3(A) shows an expanded example of the vertical abstraction. Although threads can be visualized with vertical abstraction, the space complexity for visualization may be still big. The horizontal abstraction reduces the number of threads visualized on the same circumference, by representing a set of threads with one abstract thread. Figure 3(B) shows an example of horizontal abstraction. The second nesting level in the figure shows horizontal abstraction by the rate of four and the third nesting level by the rate of two.

The thread abstraction allows us to understand intuitively whether a pair of threads is concurrent or ordered with each other, because we can see easily an explicit path between any two threads on the cone. For example, in the Figure 3(B), the left thread in the first nesting level is concurrent with the right thread in the third nesting level, because the explicit path from the upside to the downside does not exist on the visualized cone. Users can check easily whether a pair of threads at the different nesting levels are concurrent or ordered with each other through the thread abstraction.

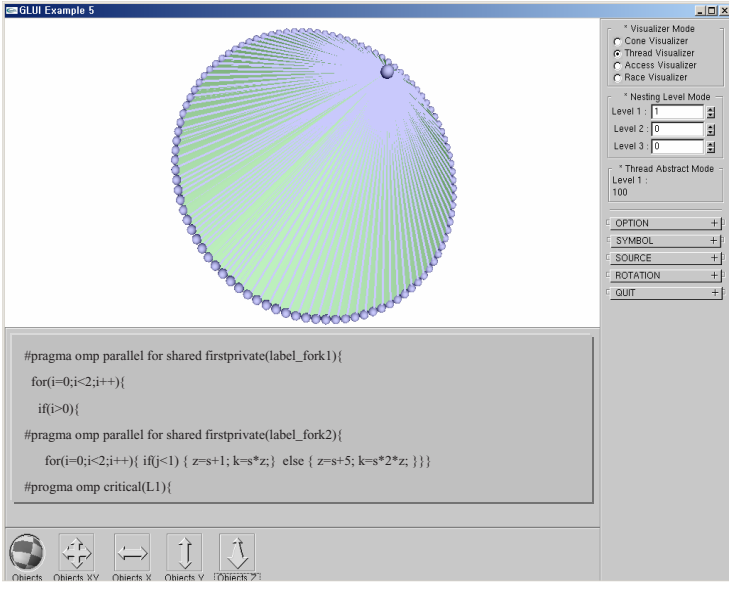

Fig. 5. No Critical Sections and No Nested Parallelisms

4 Experimentation

We implemented scalable thread visualization and experimented its functionality using a set of synthetic programs. This section presents the interface of implemented tool and the principles in which the tool draws the symbols using an execution trace of the synthetic programs.

4.1 Visualization Engines

Figure 4 shows the interface of our thread visualization tool which is composed two views and two menus: *Visualization View*, *Source code View*, *Main Menu*, and *Rotation Menu*. In the Main Menu, *Visualizer Mode* has four modes in which two modes are currently implemented: *Cone Visualizer* and *Thread Visualizer*. *Nesting Level Mode* provides the possible values of each nesting level and then users can select a numeral in each nesting level. The *OPTION* menu make it possible to set the maximum value of nesting levels and multi-way loops. The *SYMBOL* menu shows the legend of symbols to be used for scalable visualization. The *SOURCE* and *ROTATION* menus allow users to control the activation of Source code View and Rotation Menu. The *QUIT* menu quits the interface. The Rotation Menu located at the lower left part of the interface allows users to rotate on the three-dimensional space or move up, down, left, and right using one button labelled *Objects* or the other four buttons labelled *Objects XY*, *Objects X*, *Objects Y*, and *Object Z*. When the visualized cone is rotated, its position and size are fixed. The Visualization View shown at the top of the figure visualizes

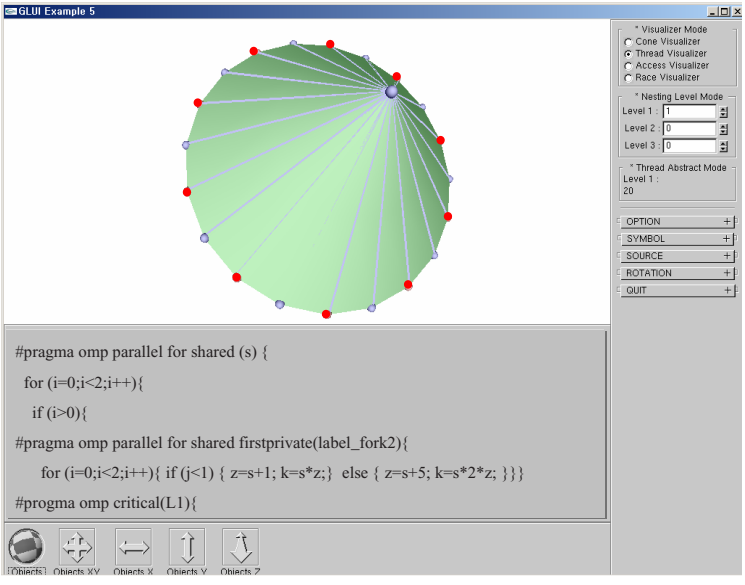

Fig. 6. Critical Sections and No Nested Parallelisms

the cone and abstract threads. The Source code View shows the corresponding program codes.

For visualization, a cone is divided horizontally by the nesting depth acquired from trace as shown in the figure. A thread is drawn on the cone based on the calculated height, angle, and symbol's position and can be abstracted for a thread set, critical sections, and nested parallel loop which are created during a program execution. The user understands races intuitively by visualizing a partial order of threads involved in races selectively. For example, in Figure 4, left symbol at the first nesting level is concurrent with the right symbol at the second nesting level, because these is no path between the left symbol and the right symbol.

4.2 Visualization Cases

The visualization tool has been implemented using Visual C++ and OpenGL library under Windows XP on Pentium 4 computer. We verified the cone and thread visualization with four kinds of synthetic programs with respect to the existence of critical sections and nested parallelisms: (1) *no nested parallelisms and no critical sections*, (2) *nested parallelisms and no critical sections*, (3) *no nested parallelisms and some critical sections*, (4) *nested parallelisms and critical sections*. Any critical section uses one lock variable. The nesting depth is three, and each nesting level has 20, 100, 300 threads.

For example, Figure 5 visualizes an execution of synthetic program with no nested parallelism and no critical section, which creates one hundred threads.

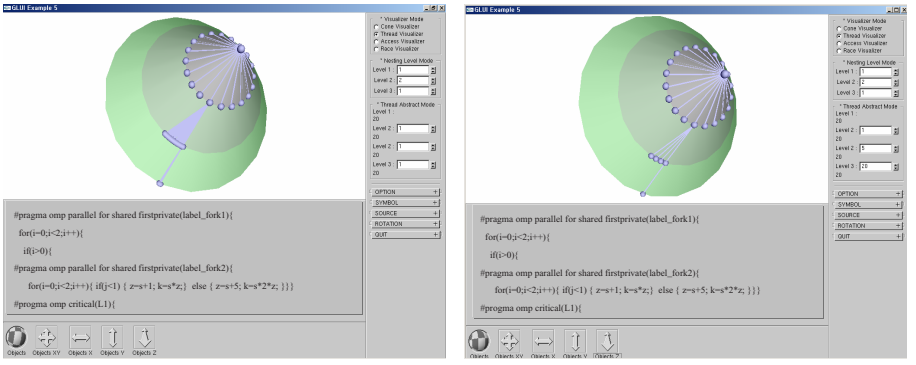

Fig. 7. Nested Parallelisms and No Critical Sections

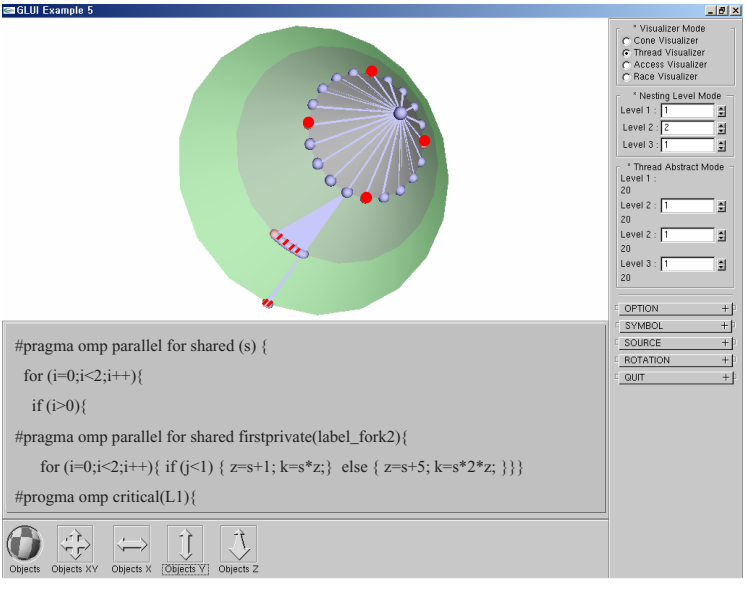

Fig. 8. Critical Sections and Nested Parallelisms

The cone in the figure is not divided, because the execution does not include nested parallelism. Figure 6 visualizes an execution of synthetic program with critical sections and no nested parallelisms, which has twenty threads and contains critical sections in every other thread. The figure shows every thread with critical section has a unique color according to its lock variable. Figure 7 visualizes an execution of synthetic program with nested parallelisms and no critical sections. Each nesting level has twenty threads; the nesting depth is three; a one-way loop within the second nesting level is two, the second one-way loop of

the second nesting level has the third nesting level. Figure 7(A) marks twenty threads within the first nesting level and one of them has twenty nested threads to exist in the second nesting level. These threads are marked in the limited area like the second nesting level of Figure 7(A), because the overlap among threads occurs in the second nesting level if all threads of the first nesting level have nested threads. If this overlap phenomenon is occur, we can not understand duly the visualized results so we provide a horizontal abstraction like Figure 7(B). Figure 7(B) abstracts the threads at the rate of a quarter about twenty threads of the second nesting level of Figure 7(A). As the result, only four threads are visualized in the second level. Figure 8 visualizes threads the synthetic program with nested parallelism and critical section. It is identical with the explanation of Figure 7(A) except the mark of critical section.

5 Conclusion

Data race in OpenMP programs must be detected for debugging, because it may cause unexpected results incurred from unintended non-deterministic executions. OpenMP programs are often complex and long-running, because parallel programs may consist of a large number of threads and accesses to shared variables. Thread Checker of Intel Corporation is a unique tool to detect threading errors including data races in the relaxed sequential program which is defined as parallelized only with OpenMP directives. The tool however does not provide any effective facility for understanding the complexity of threads involved in the reported races.

This paper presents a thread visualization tool to represent the partial order of threads in the traced OpenMP programs with a scalable graph of abstract threads upon a three-dimensional cone. This tool solves the visual complexity using the abstract visualization which replaces a set of events with an abstract symbol and provides the thread information which is traced by RaceStand, an on-the-fly race detection tool. We have been trying to apply this tool using a set of published benchmark programs in addition to our synthetic programs specially developed for experimenting this tool.

References

1. Dinning, A., and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *2nd Symp. on Principles and Practice of Parallel Programming*, pp. 1-10, ACM, March 1990.
2. Dinning, A., and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *2nd Workshop on Parallel and Distributed Debugging*, pp. 85-96, ACM, May 1991.
3. Intel Corp., *Getting Started with the Intel C++ Compiler 9.0 for Windows.*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2004.
4. Intel Corp., *Getting Started with the Intel Thread Checker*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2004.

5. Intel Corp., *Intel Thread Checker for Windows 3.0 Release Notes*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2005.
6. Intel Corp., *VTune(TM) Performance Analyzer 8.0 Release Notes*, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA, 2006.
7. Jun, Y. and K. Koh, "On-the-fly Detection of Access Anomalies in Nested Parallel Loops," *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pp.107-117, ACM, May 1993.
8. Kim, Y., M. Park, S. Park, and Y. Jun, "A Practical Tool for Detecting Races in OpenMP Programs," Proc. of 8th Int'l Conf. on Parallel Computing Technologies (PaCT), Krasnoyarsk, Russia, Lecture Notes in Computer Science, 3606: 321-330, Springer-Verlag, Sept. 2005.
9. Kim, Y., and Y. Jun, "An Optimal Tool for Verifying Races in OpenMP Programs," *06 Int'l Conference on Hybrid Information Technology*, SERC, Cheju Island, Korea, Nov., 2006
10. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of ACM*, 21(7): 558-565, ACM, July 1978.
11. Mellor-Crummey, J. M., "On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Supercomputing*, pp. 24-33, ACM/IEEE, Nov. 1991.
12. Netzer, R. H. B., and B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations," *Letters on Programming Lang. and Systems*, 1(1): 74-88, ACM, March 1992.
13. Nudler, I., and L. Rudolph, "Tools for the Efficient Development of Efficient Parallel Programs," *In 1st Israeli Conference on Computer System Engineering*, 1986.
14. OpenMP Architecture Review Board, *OpenMP Application Programs Interface*, Version 2.5, May 2005.
15. Park, S., M. Park, and Y. Jun, "A Comparision of Scalable Labeling Schemes for Detecting Races in OpenMP Programs," *Int'l Workshop on OpenMP Applications and Tools* (Wompat), pp. 66-80, West lafayette, Indiana, July 2001.
16. Petersen, P., and S. Shah, "OpenMP Support in the Intel Thread Checker," *Proc. of the Int'l Workshop on OpenMP Application and Tools* (WOMPAT), Berlin Heidelberg, Lecture Notes in Computer Science, 2716: 1-12, Springer-Verlag, 2003.