Visualization Tool for Debugging Data Races in Structured Fork-join Parallel Programs

Myeong-Sin Kang¹, Ok-Kyoon Ha² and Yong-Kee Jun¹

¹Department of Informatics, Gyeongsang National University
²Engineering Research Institute, Gyeongsang National University
{mskang, jassmin, jun}@gnu.ac.kr

Abstract

Ensuring the reliability of structured fork-join parallel programs is difficult because the potential for subtle interactions between concurrent threads can cause concurrency bugs, such as data races, which are hard to detect, reproduce, and eliminate. The visualization for the executions of the programs may offer effective debugging environments with intuitively understanding. Unfortunately, visualization techniques for structured fork-join parallel programs still also difficult to represent and analyze the information of programs executions, because the information for analyzing thread executions and relevant events to data races are increased exponentially in proportion to maximum parallelism of the program. This paper presents a visualization tool that offers overall information for detecting data races by grouping and abstracting thread executions and accesses to shared variables. Moreover, the tool provides an effective approach to debug data races by indicating locations of the defects.

Keywords: Structured fork-join parallelism, data races, visualization, effective debugging

1. Introduction

Parallel or multithreaded programs are becoming a natural consequence to achieve improved performance of applications in multi-core systems. Structured fork-join parallel programs, such as OpenMP [1, 2], Cilk [3], Threading Building Block [4], are widely used due to fact that they make easier to produce a parallel program that is more robust and efficient [5]. However, it is still difficult to ensure the reliability of the programs due to concurrency bugs, such as data races, which cause the potential for subtle interactions between concurrent threads because the concurrency bugs are hard to reproduce with traditional sequential testing or debugging methods due to the non-deterministic interleaving of concurrent threads.

Data races [6-9] in parallel programs occur when two concurrent threads access a shared variable without explicit synchronization, and at least one of the accesses is a write. It is important to detect data races for debugging parallel programs, because they may lead to non-intended executions and results. The visualization techniques are introduced to help understanding and to effective debugging data races in the executions of parallel programs with a scalable graph. However, prior techniques still also difficult to represent and analyze the information of programs executions, because the information for analyzing thread executions and relevant events to data races are increased exponentially in proportion to the number of active threads and events in an execution of the program.

This paper presents an effective visualization tool that offers overall information for detecting data races by considering grouping and abstracting methods to thread executions and accesses to shared variables. We consider structured fork-join parallel
programs [10-14] which may include critical sections and nested parallelism. The visualization makes it overcome the limitation on a visual space and uses an execution graph to represent the partial order of threads and events. The tool solves the visual complexity problem of prior tools using the grouping the same execution of thread segments and abstracting parallel regions and events. The grouping and abstracting concepts reduce the space complexity of thread visualization and help to understand the large scale structure of parallel threads effectively. Moreover, the tool provides an effective approach to debug data races by indicating the locations of the defects.

```c
float price; /* a shared variable */
...
#pragma omp parallel for private (matrix)
for (i=0; i< NUM_RUNS; i++) {
  if (i%2 == 0 && i != 0) {
    matrix[i][i+1] = par_func (op1, i);
    matrix[i+1][i] = par_func (op2, i);
  }
}
...  
float par_func (int op, int i) {
#pragma omp parallel for
for (j=0; j<NUM_RUNS; j++) {
  price = max_pixel (t, op);
  ...  
  privates = price % privates;
  ...
}
...  
```

(a) A structured fork-join parallel program  (b) An execution graph of the program

**Figure 1. An Example of a Structured Fork-join Parallel Program and its POEG**

2. Background

This section illustrates data races in structured fork-join parallel programs and introduces the previous work that uses visualization techniques to detect data races.

2.1. Data Races in Structured Fork-join Parallelism

Structured fork-join parallelism [10-13] is a parallel threading model which provides a parallel execution of repeated data operations with parallel directives or APIs. This model requires at least one set of fork operations to create a set of concurrent threads, and a corresponding join operation to terminate the forked threads and to spawn a new single thread. A structured fork-join parallel program can contain nested parallelism or series-parallelism with other fork-join constructs, where nested parallelism is a nestable fork-join model of parallel execution in which a parallel thread can generate a team of other parallel threads [14, 15].

The structured fork-join parallel programs make it easier to identify accesses to a shared variable on parallel threads and to produce a parallel program that is more robust and efficient [5]. Thus, the several parallel programs employ the parallel threading model, such as OpenMP [1, 2], Cilk [3], Intel Threading Building Block [4], and so on.
Figure 1(a) shows an example of a structured fork-join parallel program using OpenMP parallel directives. The program spawns a set of parallel thread segments as NUM_RUNS at the entry of each for loop in lines 4 and 12, and join operations terminate the forked threads and spawn a single thread at the end of the loops. The program contains both the nested parallelism and series-parallelism due to two function calls in lines 6 and 7. The execution of the program in Figure 1(a) can be represented as directed acyclic graph, called POEG (Partial Order Execution Graph). A POEG for the program shown in Figure 1(a) is illustrated in Figure 1(b).

We assume that NUM_RUNS is set 3 and that a proper locking for inter-thread synchronization is used during an execution of the program. In the graph, a vertex is either a fork or a join operation for a set of parallel thread segments, and an arc starting from a vertex represents a logical thread segment started from a thread operation. Two events, read and write, for a shared variable price in Figure 1(a) are denoted as \( R_i \) and \( W_i \) in Figure 1(b). The numbers attached to each thread segment and event name indicate an observed order, and an arc segment delimited by a pair of symbols \( \sqcap \) and \( \sqcup \) which means a region of the thread segments protected by a common lock.

There are three pairs of data races, \{ \( R_1-W_6, W_2-R_5, W_2-W_6 \) \}, during the execution of program shown in Figure 1(b). Two events on thread segment \( T_9 \) consist of data races potentially with the two events on thread segment \( T_2 \), because \( T_2 \) concurrent with \( T_9 \), and two events on \( T_9 \) are not protected by any lock.

### 2.2. Visualization for Detecting Data Races

Various visualizations are introduced to debug data races in parallel or multithreaded programs (e.g., Directed Acyclic Graph (DAG) [10], Process-Time Diagram [16], Unified Modeling Language (UML) [17]). Especially, there are three representative visualization techniques [18]-[20] for considering structured fork-join parallel programs. Figure 2 shows captured execution of the fork-join parallel programs by each of techniques.

![Figure 2. Visualization Techniques for Structured Fork-Join Parallel Programs](image)

The visualization technique of [18] employs a 3D model of a cube to represent the structure of parallel threads and information of data races during an execution of the parallel program, such as OpenMP. Figure 2(a) shows the shape of the visualization technique. The technique represents proper size of threads which have one of nesting
depth to an abstracted thread, called *superthread*, and the superthreads are replaced as the vertical edges of the cube visualization. Accesses to a shared variable and nested parallel loops are indicated on the superthreads by district symbols. This visualization technique reduces the space complexity to represent the information of thread executions and relevant events. However, it does not provide any information of nested parallel loops (i.e., the nesting depth is larger than two), and possible to maximally represent only four superthreads. Thus, the technique cannot apply to large scale applications which use a large amount of threads and have the complex execution model.

Figure 2(b) shows a 3D model of the cone visualization [19] which represents a partial order of threads executions with a scalable graph of abstract threads in the traced fork-join parallel programs. The visualization technique represents each nesting level to corresponding horizontal areas on a 3D cone, and indicates the execution flows of threads from top of the cone to bottom. Accesses to a shared variable are replaced on a line of demarcation for nesting level. By the technique, we can eidetically understand where the threads are concurrent with others considering nesting levels. However, the cone visualization is difficult to indicate other parallel regions which have the same nesting levels (e.g., either $T_3$ or $T_7$ shown in Figure 1(b) appears on the cone). Moreover, it is also difficult to clearly understand where accesses to a shared variable were occurred, because it possible to indicate only one access for each thread.

Figure 2(c) shows a 3D model of the disk visualization [20] which reduces the space complexity through the representation of parallel regions, such as the fork-join loop, in structured fork-join parallel programs. This technique provides easy to understand over all structure of program execution with a thread abstraction, because it represents threads on a same nesting level to a disk objective and indicates nested disks for nested parallel regions with distinguished symbols. However, the technique also increases the runtime overhead of visualization and the complexity of visual analysis for the large scale applications.

### 3. Design of Space Efficient Visualization

Several real applications which use structured fork-join parallelism usually have the complex aspects of thread executions. This section illustrates the design of space efficient visualization that uses grouping and abstracting method for most thread segments which execute the same instruction of target program.

#### 3.1. Visualization Symbols

This paper uses some symbols to provide understanding the aspects of thread executions and information of accesses to shared variables occurred during an execution of the structured fork-join parallel program. The symbols appear in Figure 3. The symbols classify into the thread symbols and the event symbols.
Thread Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚡️</td>
<td>A thread segment</td>
</tr>
<tr>
<td>🔰</td>
<td>A set of thread segments</td>
</tr>
<tr>
<td>🖋️</td>
<td>An extended parallel region</td>
</tr>
<tr>
<td>🗝️</td>
<td>An abstracted parallel region</td>
</tr>
</tbody>
</table>

Event Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>📜</td>
<td>A read access</td>
</tr>
<tr>
<td>🌱</td>
<td>A write access</td>
</tr>
<tr>
<td>⊔⊔</td>
<td>A critical section</td>
</tr>
<tr>
<td>⬠</td>
<td>A race indicator</td>
</tr>
</tbody>
</table>

**Figure 3. Visualization Symbols**

The thread symbols consider a thread segment, a set of thread segments, and the parallel region of thread segments. The detail view of each thread symbol is as follows:

- A *thread segment* is represented by an arrow shape started from an arc which means a starting pointer of a thread segment.
- A *set of thread segments* that have same aspects of thread executions is represented by an arrow shape started from a filled circle with a positive integer which indicates the number of integrated thread segments.
- An *extended parallel region* is represented as a bounded rectangle for a parallel region of thread segments, that usually considers a fork-join construct which consists of one or more thread segments or other parallel regions.
- An *abstracted parallel region* is represented as a filled rectangle for abstracting a parallel region. A positive integer is also employed to indicate the number of hidden thread segments or parallel regions.

The event symbols consider events occurred on an execution of thread segments, such as accesses to a shared variable (read/write), critical sections for mutual thread executions, and a racy pair accesses. The detail view of each event symbol is as follows:

- A *read access* of a shared variable is represented by a symbol 📜.
- A *write access* to a shared variable is represented by a symbol 🌱.
- A *critical section* for a mutual exclusion of thread segments is represented by a pair of symbols ⊔⊔ and ⊔⊔.
- A *race indicator* is represented by a dotted line to indicate a relation of a pair of accesses that involved in a data race.
3.2. Grouping Thread Segments

This paper employs grouping concept of thread segments that have same aspects of thread executions in a parallel region for effective visualization, because almost all these thread segments usually do not involve directly any data race (i.e., only two thread segments are related to a data race). The grouping thread segments replaces two or more thread segments into a set of thread segments symbol without considering the execution order of thread segments or occurring order of events.

![Diagram](image)

(a) Grouping  
(b) Abstracting  
(c) Indicating

Figure 4. The Space Efficient Visualization

For example, in Figure 1(b), two thread segment \(T_2\) and \(T_3\) can be integrated as a grouped thread segment for a parallel region started from a thread segment \(T_1\) by the grouping concept although \(T_2\) has different events with \(T_3\). However, another sibling thread in the same parallel region, from \(T_4\) to \(T_{12}\), is excluded for the grouping process, since the thread includes other nested parallel regions. Two set of thread segments \{\(T_5, T_6, T_7\)\} in a parallel region started from \(T_4\) and \{\(T_9, T_{10}, T_{11}\)\} in a parallel region started from \(T_8\) are integrated as a grouped thread segment, respectively. The representation using the grouping concept is appeared in Figure 4(a) (see \(G_1, G_2,\) and \(G_3\) in the figure). Each number indicates the number of grouped thread segments on three set of thread segment symbols.

Because the structured fork-join parallel programs usually employ the parallel program model of SPMD (Single Program Multiple Data), this grouping concept can be simply applied to the visualization of the programs and significantly reduces the space complexity of visualization for debugging data races.

3.3. Abstracting Parallel Regions and Indicating Data Races

Since the nested thread segments and parallel regions in an extended parallel region that they do not related to any racy pair can be regarded as the unnecessary elements to present the visualization for debugging data races, the abstracting parallel region concept is employed to replace an extended parallel region into an abstracted parallel region for our visualization method. Moreover, the concept helps understanding overall structure of the program executions and provides the scalable visualization.
Figure 4(b) shows a scalable graph which represents the parallel regions by applying the abstracting concept to each nested level. In the figure, two parallel regions $P_2$ which forked from $T_4$ and consists of three thread segments $\{T_5, T_6, T_7\}$ and $P_3$ which forked from $T_8$ and consists of $\{T_9, T_{10}, T_{11}\}$ are can be represented as each abstracted parallel region for the nesting level one. For the nesting level zero, a fork-join construct which forked by $T_1$ (i.e., $E_1$ is used to indicate an extended parallel region for this fork-join construct in Figure 4(b)) can be also represented an abstracted parallel region by our abstracting concept (see $P_1$ in Figure 4(b)). An abstracted parallel region may contains access symbols, $\textcircled{R}$, $\mathbb{W}$, $\textcircled{R}\mathbb{W}$, or $\mathbb{W}\mathbb{W}$, for erroneous accesses which consist of a data race, if a thread segment which includes one or more the accesses is hidden by the abstracting parallel region. The abstracting concept provides the space efficient visualization without any limitation for the nesting depth of parallel programs.

Both two concepts, the grouping thread segments and the abstracting parallel regions, provide the effectiveness and the scalability of visualization for debugging data races. We use an additional concept to precisely indicate two positions of a racy pair to check where a data race has been run into an execution of the program. Because data races are occurred by a tuple $t(e_i, e_j)$ which two threads access a shared variable, at least one of them is a write, the indicating figures out a pair of erroneous accesses between two separated thread segments with a race indicator symbol. Figure 4(c) depicts indicating a data race.

4. Evaluation

This section illustrates implemented visualization tool for data race detection and evaluates the effectiveness of the tool with a benchmark application which uses 512 concurrent threads in an execution.

4.1 Implementation

We implemented our visualization tool based on three simple concepts which appeared in Section 3 using Java and SWT (standard widget toolkit). The SWT provides reach graphic libraries for Java and is used in order to flexibly cope with platform environment and fast graphic processing. The implementation was carried on a system with an Intel core i5 3.4 GHz CPU and 4GB Memory under Windows 7 OS. Our visualization tool uses the results of data race detection and information of threads execution, which are produced by a data race detection tool [5], and captures the original source codes.

Figure 5 shows the overall interface of implemented visualization tool. The tool consists of three views and a control menu, and their details as follows:

- **Visualization view** is an actual visualization panel to display a POEG for an execution of a structured fork-join parallel program with thread symbols and access symbols. In this view, three concepts for our visualization are used in order to reduce drawing space for limited area of the view panel. For the abstracting concept, we employed two additional symbols, $\oplus$ and $\ominus$, to change mutuals a parallel region to each of abstracted one and extended one.
Figure 5. The Overall Interface of our Visualization Tool

- **Source code view** provides the original source codes to display where a data race ran with the execution of the program by indicating each of two lines that related to the racy pair of accesses. Debugging data races can be more practical by this view.

- **Race list view** displays a list of racy pairs to select a pair of erroneous accesses. To indicate a data race, the symbols on the visualization view are rearranged and the code lines in the source code view are highlighted by a selected racy pair of the race list view.

- **Control menu** provides additional functions, such as sorting the race list, unabstracting all parallel regions which include accesses to shared variables, and deleting already indicated racy information in the visualization view and the source code view.

4.2. Experimentation

We evaluated empirically the effectiveness of our visualization tool with a well-known benchmark, MolecularDynamic [21, 22] which targets HPC (High Performance Computing) with OpenMP parallel directives. The benchmark has three serialized parallel regions which are contained each fork-join construct, and a parallel region consists of 512 concurrent threads for the maximum number of simultaneously active threads during an execution and.
Figure 6 depicts the visualization for an execution result of the benchmark by our tool. The three parallel regions were rendered by the abstracted parallel region symbols, and each symbol indicates that 512 of thread segments were hidden by the abstraction. The race list view presents eight racy pairs that these data races have been located by the detector during an execution of the program.

Figure 7 shows how the visualization tool indicates where a data race ran with the program execution through the graph of the visualization view and the source code view. For a selected a racy pair in the race list view, an abstracted parallel region is extended, and two erroneous accesses on two thread segments are indicated by a race indicator symbol in the visualization view. Additionally, two lines which include each of the erroneous accesses in the source code view are highlighted to provide the facility of the tool for debugging data races.

5. Conclusion

Ensuring the reliability of structured fork-join parallel programs is difficult because the potential for subtle interactions between concurrent threads can cause concurrency bugs, such as data races, which are hard to detect, reproduce, and eliminate. The visualization techniques are introduced to help understanding and to effective debugging data races in the executions of parallel programs with a scalable graph. However, prior visualization techniques for structured fork-join parallel programs still also difficult to represent and analyze the information of programs executions.
Figure 7. The Visualization for an Execution Result of the MolecularDynamic (Indicating)

This paper presented a visualization tool that offers overall information for detecting data races by grouping and abstracting thread executions and accesses to shared variables. The grouping and abstracting concepts reduce the space complexity of thread visualization and helps to understand the large scale structure of parallel threads effectively. Moreover, the tool provides an effective approach to debug data races by indicating locations of the defects.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2011389) and was also supported by the MSIP(Ministry of Science, ICT & Future Planning), Korea, under the “SW master's course of a hiring contract” support program (NIPA-2013-HB301-13-1004) supervised by the NIPA (National IT Industry Promotion Agency).

References


Authors

Mr. Myeong-sin Kang received the BS degree in Computer Science from Gyeongsang National University (GNU), South Korea. He is currently enrolled MS degree from the Department of Informatics in GNU. He worked as an engineer of IT department in Korea industry for several years. His research interests include distributed programming and its debugging, embedded system programs for avionics, and dependable systems.
**Dr. Ok-Kyoong Ha** received the BS degree in Computer Science under the Bachelor’s Degree Examination Law for Self-Education from National Institute for Lifelong Education, and the MS and PhD degree in Informatics from Gyeongsang National University (GNU), South Korea. He is now a Research Fellow of Engineering Research Institute (ERI) in GNU. He worked as the manager of IT department in Korea industry for several years. His research interests include parallel/distributed programming and its debugging, embedded system programs, and dependable systems. Dr. Ha is a member of Korean Institute of Information Technology (KIIT) and Korea Institute of Information Scientist and Engineers (KIISE).

**Dr. Yong-Kee Jun** received the BS degree in Computer Engineering from Kyungpook National University, and the MS and PhD degree in Computer Science from Seoul National University. He is now a full professor in the Department of Informatics, Gyeongsang National University, where he had served as the first director of GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of GNU Virtual College. He is now the head of GNU Computer Science Division and the director of the GNU Embedded Software Center for Avionics (GESCA), a national IT Research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Prof. Jun is a member of Association for Computing Machinery (ACM) and IEEE Computer Society.