Project Proposal:
OpenMP Debugger

Jack Wei Luca Borletti

1 URL

This is the URL to our project repo: https://github.com/JackWei33/OpenMP-Debugger

2 Summary

We're developing a powerful debugging tool for C++ code using OpenMP. This tool will help developers identify bottlenecks, optimize performance, and ensure correctness by detecting potential race conditions and synchronization issues.

3 Background

We plan on implementing several features for our debugging tool:

1. Post-execution Analysis on Thread Workload Distribution: This feature will graph the workload distribution across threads and provide statistics for when threads are sitting idle or busy or waiting for other threads to synchronize. This data will be gathered through the OMPT tool (a first-party API for performance tools) and we will use callbacks related to OpenMP commands to inject our own code into the execution. One option here is to log relevant data to be analyzed after the code is finished executing. This feature will allow users to better distribute their workloads and quickly identify possible performance bottlenecks.
2. Post-execution Analysis on Intrathread Workloads: Even within threads, workload analysis could be immensly helpful. Our tool could provide data on how long each thread spends moving data to memory versus performing arithmetic operations. It could also display how long a thread spends in each function, allowing users to immediately identify bottlenecks in performance. This data could be collected through our own custom logging functions or OMPT.
3. Race Condition Checks: We want our tool to help users identify and debug race conditions. We will first look into detecting race conditions at runtime. One option is to write our own memory access tracer using frameworks like Intel PIN. We could also cleverly use the data provided by OMPT to identify race conditions, although OMPT does not natively support memory accessing tracing. Then we will try to identify race conditions statically. There are many tools like this and we can try to implement our own by looking at their algorithms.
4. Deadlock Detection: Similar to race conditions, we want our code to identify deadlocks and show users how to fix them. Using OMPT data, we can keep track of lock acquisition and release patterns by building a dependency graph. If a circular dependency is identified, we can stop the program and notify the user the offending threads and locks causing the issue.

5. Other possible features:

   (a) Debugger that controls executions using breakpoints and stepping (e.g. gdb)

   (b) Memory leak detector

   (c) Automatic performance optimization suggester

4 The Challenge

The problem is overall a new one for us. Neither of us have experience writing these kinds of debugging tools that interact with our code. Memory leak detectors like Valgrind and race condition detectors like ThreadSanitizer often work like magical black boxs and understanding how they work is one of the main benefits of this project. Furthermore, interacting with OpenMP and its implementation will teach us a lot about how parallel programs are creted and managed.

Developing this debugging tool presents several unique challenges. First, we need to build a lightweight and non-intrusive debugger; our tool's performance monitoring must have minimal impact on the execution of the code itself. We want to measure the standalone code performance, not how it runs with the debugger's overhead. Achieving this will require researching efficient algorithms and methods used by other debugging tools to maintain functionality while minimizing interference.

Additionally, we want a tool that is easy to use. The debugger should require minimal setup, ideally involving only a few lines of additional code, making usability and integration with OpenMP straightforward. We'll need to deepen our understanding of OpenMP's functionality, including OMPT (OpenMP Tools API), to inject our analysis and monitoring code precisely where needed.

Finally, we'll need to understand how to collect and analyze the necessary data effectively. While OMPT will be a primary resource, much of the in-depth analysis will require additional processing beyond the raw data OMPT provides. For instance, detecting deadlocks will involve tracking patterns of lock acquisition and release, and identifying race conditions will demand tracking memory accesses across threads. Conducting this level of analysis will require us to study and implement many advanced algorithms designed for debugging parallel programs.

5 Resources

We will likely rely heavily on an OpenMP API called OMPT — a first-party API for performance tools. This API track events within OpenMP programs, such as thread creation, task scheduling, synchronization, and other runtime activities. It can also collect detailed performance data, such as timing and resource utilization. With this tool, we should be able to extract the data we need and display it to the user. OpenMP has another tool OMPD — a shared-library plugin for debuggers that enables a debugger to inspect and control execution of an OpenMP program. Although we are not intending to build a debugger with breakpoints that controls the code execution, this is another resource we could look into.

We will also spend some time looking at existing debugging tools for OpenMP, the two most popular being TotalView and DDT. These debuggers are more general purpose parallel debuggers, but we will be creating a debugger specifically for OpenMP. However, they will still guide us in designing our own as well as show us what information is most useful to the user.

6 Goals and Deliverables

6.1 Plan to Achieve

We plan to achieve the four features outlined in the background: post-execution analysis on thread workload distribution, post-execution analysis on intrathread workloads, race condition checks and deadlock detection. With just these four features, we believe we will already have a very impactful product. Thinking back to Assignment 3, having a debugger with these features would have greatly streamlined the process of speeding up our code. If we manage to finish these parts of the project, we hope to implement some of the other possible features in the background: a debugger that controls executions of diffent threads using breakpoints and stepping, a memory leak detector, and an automatic performance optimization suggester.

6.2 Demo

We plan to demo our debugger on our version of Assignment 3: Parallel VLSI Wire Routing via OpenMP. We have many different commits with our code achieving variable degrees of success. At each commit, we will use our debugger to quickly and easily identify where our performance bottlenecks are. We can also build some example programs with subtle race conditions or deadlocks. This will allow us to show the correctness aspect of our debugger. On our poster, we can display our different features and how they are implemented.

6.3 System Capabilities

The system will allow users to identify performance bottlenecks in C++ code using OpenMP. Additionally, the debugger will be able to catch correctness errors in race conditions and deadlocks. The system will ideally be light-weight and not effect code execution, as well as require minimal extra lines of codes from the user to use.

7 Platform Choice

We chose to write a debugger for C++ and OpenMP as they are the standard language and framework chosen to write high performance parallel code. Our debugger will largely be written in C++ for the same reason. We will be running our code on the Gates Machines in order to test and debug code written for multi-core processors.

8 Schedule

1. **November 13 - November 19**
    - **Goal**: Set up development environment and initial research.
    - Familiarize with OMPT (OpenMP Tools API) documentation.
    - Research memory tracing techniques and identify potential algorithms for race condition detection.
    - Draft a high-level architecture and outline core features of the debugger.
2. **November 20 - November 26**
    - **Goal**: Begin implementation of basic data collection.
    - Develop initial setup for OMPT integration, focusing on collecting basic thread and task scheduling data.

- Implement preliminary data logging functions to gather workload distribution data.
- Build testing to verify that data is being accurately collected.

3. **November 27 - December 3 (Intermediate Milestone on November 27)**
    - **Goal**: Use collected data to provide post-execution analysis and begin implementing race condition detection.
    - Add graphing functions to convert collected data to post-execution analysis for thread workload distribution and intrathread workloads.
    - Begin implementing basic race condition detection using memory tracing, starting with runtime detection.
    - Test memory tracing functions to ensure accurate data on memory access patterns.
4. **December 4 - December 10**
    - **Goal**: Implement deadlock detection and continue testing race condition detection.
    - Develop deadlock detection logic by tracking lock acquisition and release patterns.
    - Integrate dependency graph analysis to identify potential circular dependencies and deadlocks.
    - Test the combined race condition and deadlock detection functions with sample OpenMP code.
    - Collect test results and adjust detection algorithms as needed for accuracy and performance.
5. **December 11 - December 15**
    - **Goal**: User-friendly output and initial UI for debugging data.
    - Design a simple user interface for displaying debugging information, focusing on readability and usability.
    - Integrate visualizations for thread workload distribution and identified bottlenecks.
    - Develop output formats for race condition and deadlock reports to guide the user through debugging.