# Comprehensive OpenMP Analysis and System Supervision (COMPASS): An OMPT Tool for Program Analysis and Visualization

JACK WEI[*] and LUCA BORLETTI[*], Carnegie Mellon University, USA

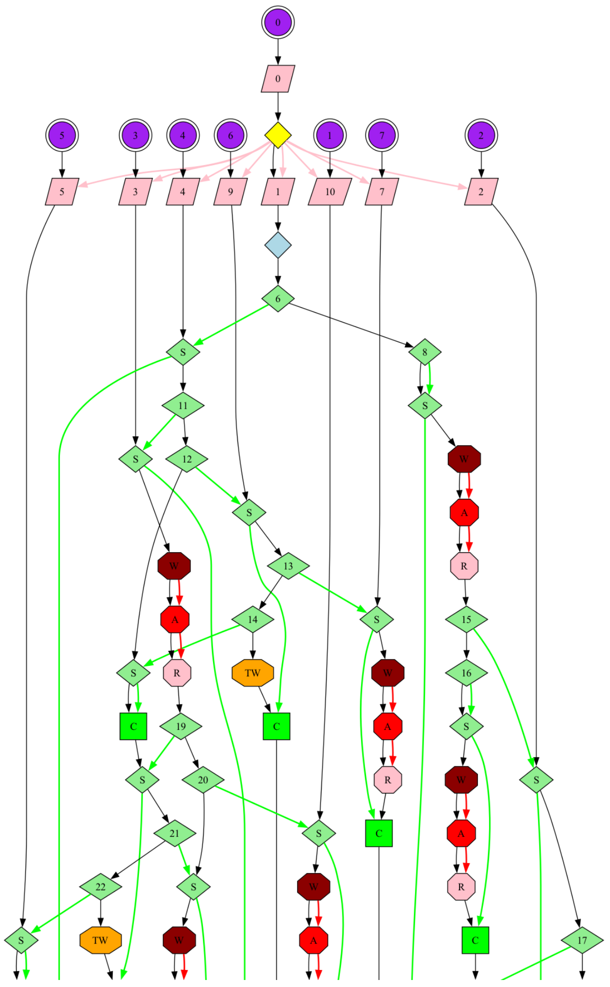


Fig. 1. A runtime execution graph of a C++ OpenMP program (task-based Fibonacci solver with mutex-locked memoization) generated by COMPASS.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: C++, OpenMP, Debugger, OMPT, Deadlock Detection, Execution Graph

[*]Both authors contributed equally to this research.

Authors' Contact Information: Jack Wei, jackwei@andrew; Luca Borletti, lborlett@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

## 1 Github Repo Link

https://github.com/JackWei33/OpenMP-Debugger.git

## 2 Summary

We implemented an OMPT performance analysis and debugging tool for C++ programs with OpenMP directives. COMPASS performs event logging and deadlock detection at runtime. After program termination, postmortem analysis is displayed through two visualizations: an execution graph as well as a detailed breakdown of time spent across critical regions, parallel work, barrier waiting, and other key areas.

## 3 Motivation

Getting a clear picture of the performance and computational structure of an OpenMP program can be extremely difficult. While working on Assignment 3, we found that we would often have hypotheses on what was contributing to our program's latency, but actually testing these hypotheses' correctness was laborious and time-consuming. It was often easier to blindly try solving a problem, than to investigate if it was truly a problem in the first place. Our tool attempts to provide the visibility to identify problems in OpenMP programs quickly. With little effort required from the user and minimal impact on code performance, we provide an easy way to visualize parallel execution as well as what parts of the program contribute to the end-to-end latency—be it workload imbalance, excess task spawning, lock synchronization time, or any code snippet that the user would like to investigate.

## 4 Features and their Implementation

### 4.1 OMPT

OMPT [1] (OpenMP Tools API) is a first-party API introduced by OpenMP to facilitate the development of performance analysis and debugging tools. It provides a standard interface for 'tools' to interact with OpenMP applications at runtime, allowing them to observe and collect detailed information about program execution. In particular, tool-builders have two main resources when utilizing OMPT. The first are synchronous callbacks. Tool-builders can attach functions to callbacks that correspond to different OMPT events, such as a thread entering a parallel region, a thread acquiring a lock, or the scheduling of a task onto that thread. The second feature is OpenMP data management. OMPT maintains data attached to important objects like threads, mutexes, and parallel regions. Tool-builders are able to set the value of these data and retrieve them at any time, allowing information passing between callbacks. Our application is built primarily using this API—OMPT.

### 4.2 Logging

Inside of each OMPT event callback, our OMPT tool gathers whatever data it can about the event and logs that information to log a log file—one for each active thread[1]. In these logs, we record information like the event type, timestamp, and thread number. For certain events, we keep track of extra information: for example, in the `ompt_callback_sync_region` callback, we additionally log whether the region was a barrier, taskwait, or another kind of sync region; for `ompt_callback_task_create`, we atomically increment a counter and assign its value to the task's data, allowing us to track which thread(s) that task gets scheduled onto. After program termination, our Python scripts ingest these logs to generate each of our

[1]Technically, the number of active threads can fluctuate at runtime (e.g., before vs. after the start of a parallel region, between different parallel regions, etc.), thus, the number of log files corresponds, precisely, to the maximum number of threads entering any OpenMP parallel region over the course of the program.

postmortem visualizations.

In the first versions of our tool, all log writes were synchronous and non-batched. This was incredibly detrimental to latency, as each callback took around $20\mu s$ which we attribute primarily to I/O from writing to log files. For programs or problems that required a non-trivial number of parallel regions, tasks, or active threads, our tool was not only unusable, but also innaccurate. Since our callbacks uniformly slow down all 'events,' some events which are low latency and occur often—such as mutex acquisition—change the behavior of the program when their latency is drastically increased (e.g., higher waiting times in our visualizations). While some latency is unavoidable due to the nature of live execution analysis, we believed we could do better.

We tried various tracing methodologies to bring down this latency, including OTF2, spdlog, and building our own batched, asynchronous logging solution. However, for the sake of finishing our project on time, we decided to go with Quill, a popular high-performance asynchronous logging library which, according to the creators, "is particularly suited for performance-critical applications where every microsecond counts." Initially, we faced an obstacle in integration, which was that the program would unavoidably hang prior to termination. After some inspection, this turned out to be because of a common feature built into asynchronous logging libraries, which is that the logging is disabled after `main()`[2]. Unfortunately, OMPT can (and does) frequently trigger callbacks after `main()`[3]. We were able to fix this by having a fallback synchronous logger, which is typically only activated at the end for a few logs per thread. As shown in Table 1, asynchronous logging with Quill demonstrates improved performance over synchronous logging.

| Statistic | Synchronous Logs | Batched/Asynchronous Logging with Quill |
|---|---|---|
| p50 | $17.0\mu s$ | $1.0\mu s$ |
| p90 | $22.0\mu s$ | $2.0\mu s$ |

Table 1. Logging Time Percentiles for Synchronous and Asynchronous Logging

## 4.3 Parallel Execution Graph

After the logs have been generated, and the program has terminated, the user is then able to make use of COMPASS's visualization features. One of our most useful ones is the ability to generate a runtime compute graph of a parallel program. Below, we will go through the process of going from a set of linear event sequences (i.e., logs) to a dependency graph representing the chronology as well as the task, lock, and parallel dependencies of an OpenMP program.

First, we should note that the execution graph will trivially always take the form of a DAG (trivially, as all dependencies flow 'forwards' in time). At a high level, the process of constructing this graph can be outlined as follows:

(1) Extracting Events: We begin by parsing the log files for each thread to create Python class objects that fit into several categories of event boundaries, including: Parallel regions, Implicit tasks, Custom Callbacks, Explicit tasks, and Synchronization. Events can be associated with certain identifiers, such as a *Parallel ID* for parallel regions, a *Task ID* for tasks, and a *Lock ID* for mutex locks, allowing us to link related events across threads.

(2) Creating Nodes: We create a graph node for each event, applying filters as needed. Not all low-level events may be interesting for the user or relevant to understanding the parallel execution structure, so some may be excluded based on user preference or event type (e.g., `ompt_work_single_other` is triggered when another

[2]Here is a GitHub thread documenting this issue for spdlog, and the docs that describe it for Quill.
[3]We believe that this is because the root thread reaches the end of main before the worker threads wrap up their OMPT 'implicit' tasks

thread enters a `ompt_work_single_executor`).

We also assigned shapes, colors, and labels to each type of event:

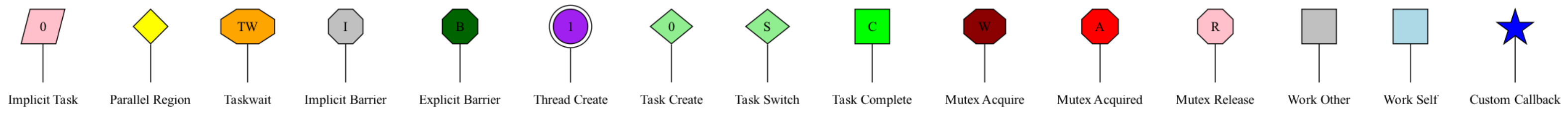


Fig. 2. Node design for each OMPT event

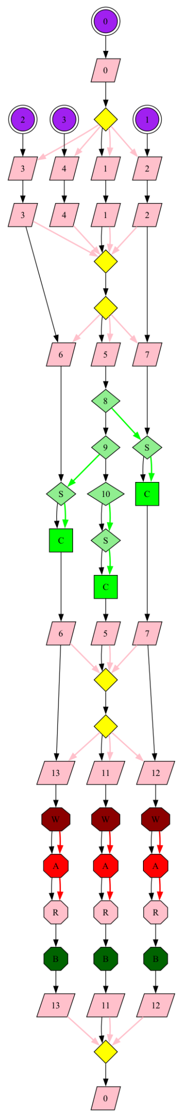


Fig. 3. Caption for image

(3) Connecting Nodes by Chronological Edges: Within each thread's log, events are listed in the chronological order they occurred. We connect these events with directed edges to reflect the temporal progression on a per-thread basis (see the black edges connecting nodes in Figure 3):

$$\text{Event}_1 \longrightarrow \text{Event}_2 \longrightarrow \text{Event}_3 \longrightarrow \cdots$$

(4) Identifying Parallel Regions and Implicit Tasks: The key to capturing the parallel execution structure is to identify which parts of the event streams correspond to the same parallel region. By using the *Parallel ID* metadata for each parallel region, we can match all *Parallel Begin / End* events in the root thread to the *Implicit Task Begin / End* events of worker threads in that region[a].

(5) Adding Task Dependencies: Next, we incorporate explicit task relationships. Each explicit task has at least three key events: a *Task Create* (on the creator thread), a *Task Switch* (on the switched-to thread), and a *Task Complete* (on the thread that completed the task). By identifying all events that share the same *Task ID*, we can connect them as follows:

$$\text{Task Create} \longrightarrow \text{Task Switch} \longrightarrow \text{Task Complete}$$

(6) Adding Synchronization Dependencies: To represent the behavior of locks and other synchronization primitives, we look at events like *Mutex Acquire*, *Mutex Acquired*, and *Mutex Released* events, all associated with a particular *Lock ID*. This allows us to illustrate the synchronization dependencies across threads:

$$\text{Acquire} \longrightarrow \text{Acquired} \longrightarrow \text{Released}$$

[a]Due to issues we experimentally encountered with inconsistent data conservation in *Implicit Task* events, we applied an algorithm similar to the Parenthesis Matching Algorithm to find the correct *End* events.

After all these steps are applied, the final structure is a DAG that faithfully represents not only the time-ordered sequence of events per thread, but also how those sequences are interwoven through parallel regions, implicit and explicit tasks, and synchronization points, potentially allowing performance engineers to:

- Identify concurrency patterns and bottlenecks in their OpenMP program.
- Understand how (and where) tasks are created, distributed, and completed.
- See where threads engage in synchronization vs. parallel work.

## 4.4 Task-based Performance Analysis and Custom Callbacks

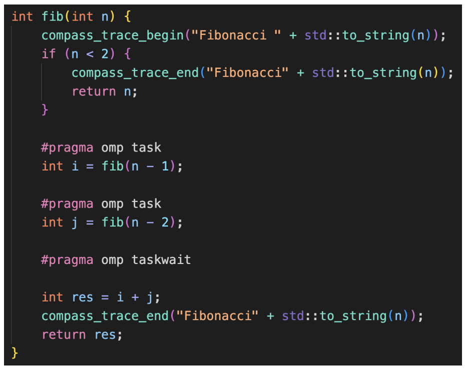
This graph is designed to output how long each thread spends working on its tasks as well as within user-defined regions we call ‘custom callbacks,‘ but below we will conflate them with tasks. This visualization utilizes the logs in a similar way to the Parallel Execution Graph. First, boundary events are identified for each parallel region. Then, for each parallel region, each threads’ logs are parsed while maintaining a stack of current tasks being worked on. When the parser reaches the end of a task, the parser checks for when the task was started and computes how long the thread was working on the task. Our tool allows users to insert benchmarking callbacks anywhere into their source code by using the following functions:

Listing 1. Example usage of compass_trace functions

```
#include "compass.h"
#include <omp.h>
int main() {
    #pragma omp parallel num_threads(8)
    {
        #pragma omp for
        for (int i = 0; i < 42; i++) {
            // Insert a trace point at the beginning of a code section
            // (could be within a parallel region)
            compass_trace_begin("Taskloop", { {"i", std::to_string(j)}});

            // Code to benchmark
            do_some_computation();

            // Insert a trace point at the end of a code section
            compass_trace_end("Some␣computation");
        }
    }
}
```

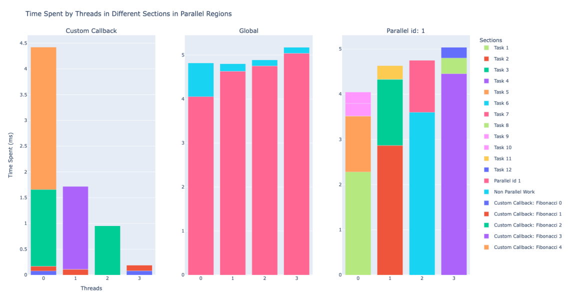
Here, `compass_trace_begin(std::string name)` marks the start of a traced region, and `compass_trace_end(std::string name)` marks its end. The ‘name‘ parameter helps identify the traced sections uniquely. `compass_trace_begin` also has an optional parameter, `std::initializer_list<std::pair<std::string, std::string>> optional_details`, which allows the user to log attributes into our visualizations; for example, if the program does repeated work on a set of structs over the course of the program, the user may want to log which struct each callback corresponds to.

Figure 4 shows an example of the graphs produced by COMPASS on a task-based Fibonacci implementation (specifically, calling `fib(4)`). As seen in the code, the implementation creates new tasks for each recursive call and waits for both tasks to complete before returning. The function is also marked at the beginning and ends with custom callbacks. The rightmost subplot of the outputted graph shows how long each of the four threads spend on its tasks. Note that each of the four threads work on more than one task, handled by OpenMP's task pool and context switching. Here, there is only one parallel region. Thus, the global subplot displays how long each thread spends working on this parallel region (pink) and doing non-parallel work (light blue). The leftmost subplot displays the time spent in custom callbacks. We can clearly see the custom callback representing 'Fibonacci 4' (orange) takes the longest to complete as it wraps all of the recursive function calls. Likewise, the base cases of 'Fibonacci 1' and 'Fibonacci 0' (red and blue) return practically instantly.



```
int fib(int n) {
    compass_trace_begin("Fibonacci " + std::to_string(n));
    if (n < 2) {
        compass_trace_end("Fibonacci" + std::to_string(n));
        return n;
    }

    #pragma omp task
    int i = fib(n - 1);

    #pragma omp task
    int j = fib(n - 2);

    #pragma omp taskwait

    int res = i + j;
    compass_trace_end("Fibonacci" + std::to_string(n));
    return res;
}
```

(a) Task-Based Fibonacci with Custom Callbacks



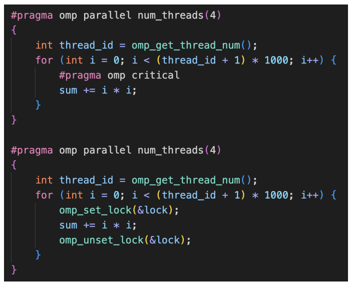
(b) Task-Based Performance Graph

Fig. 4. Task-Based Performance Graph generated from Fibonacci Code with Custom Callbacks

## 4.5 Sychronization-based Performance Analysis

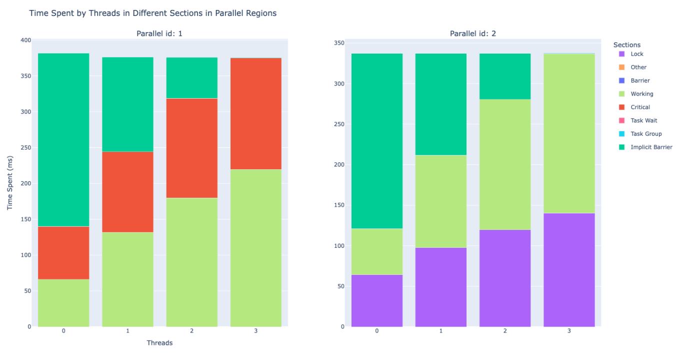
Another crucial aspect of benchmarking parallel programs is tracking synchronization primitives. Below is a visualization that displays time spent by each thread in a variety of time categories. Two important ones are Barrier Waiting and Lock Waiting, but our tool is configured to track standard locks, nest locks, critical sections, implicit and explicit barriers, task waits, and task groups.

Much like the previous graph, this graph will also create subplots for each parallel region. It parses the logs by, again, identifying boundary events for each parallel region, and then traverses the logs and tracks when synchronization periods begin and end. Figure 5 shows an example of the visualization. In the code snippet on the left, we can see two parallel regions. The first one utilizes critical sections, while the second one uses a lock. In addition, each thread has a variable amount of workload with the lower threads having less work and the higher threads having more work. The generated graph has two subplots, one for each parallel region. The first subplot shows a per-thread breakdown of time spent working, waiting for implicit barriers, and waiting for critical sections. The second subplot shows the same regions, but with lock regions instead of critical regions. We can also see that the earlier threads complete their work early and wait for the other threads to finish through the implicit barrier. This graph allows users to identify when synchronization is hurting code performance as well as display imbalances in workload distribution.



```
#pragma omp parallel num_threads(4)
{
    int thread_id = omp_get_thread_num();
    for (int i = 0; i < (thread_id + 1) * 1000; i++) {
        #pragma omp critical
        sum += i * i;
    }
}

#pragma omp parallel num_threads(4)
{
    int thread_id = omp_get_thread_num();
    for (int i = 0; i < (thread_id + 1) * 1000; i++) {
        omp_set_lock(&lock);
        sum += i * i;
        omp_unset_lock(&lock);
    }
}
```

(a) Code Snippet



(b) Performance Graph

Fig. 5. Synchronization-Based Performance Graph

As an aside, both the Task-based and Synchronization-based performance graphs are created using Plotly. Plotly is a python graphing library desgined to create interactive graphs. Plotly generates an html file that can be loaded in the browser. In our graphs specifically, users can hover over any section of the bar graph to gain more information on the thread, timed length, and region name. Additionally, they can use the legend to remove certain regions from the graph. In Figure 6, we show the cursor hovering over the "Implicit Barrier" section of thread 2. The "Working" section is greyed out in the legend and thus not showed on the bar graph.

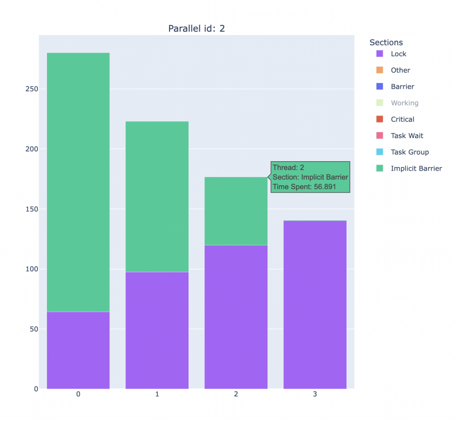


Fig. 6. Plotly Graph showing the Hover Template and Removed Working Region

## 4.6 Deadlock Detector

Besides performance graphs, we also added a correctness checking feature to our tool. The deadlock detector will spin up an additional thread on startup and maintain a 'wait-for' graph. When threads enter certain callbacks, they will push the event onto a lockfree queue implemented by the Boost library. The additional thread will sleep until the queue is not empty, then wake up and proccess the event by updating the 'wait-for' graph.

The 'wait-for' graph represents both threads and synchronization primitives as nodes. An edge from node A to node B represents node A is 'waiting' for node B. For example, if a thread is waiting to acquire a lock, the thread is ‘waiting‘ on the lock. Similarly if a lock is being held by a thread, a lock is waiting on a ‘thread.‘ The deadlock detector will maintain this graph and check for cycles. A cycle indicates that nodes are waiting in a circular dependency and there is a deadlock. In this case, the deadlock detector will stop the program and output the corresponding graph. The user can then examine the graph and debug their program.

The deadlock detector not only supports locks and critical sections, but also barriers. One can imagine if thread 0 enters a barrier, then thread 0 is waiting on the barrier while the barrier waits on every other thread. Then, say thread 1 joins the barrier. This would flip the edge between the barrier and thread 1. Once a thread releases the barrier, the dependency between the two nodes is completely removed. The implementation of this feature is non-trivial as there are certain edge cases to consider. For example, all threads could join the barrier, thread 0 releases the barrier, and then rejoins the barrier before any other threads release the barrier. In this case with a naive solution, if all acquires and releases are processed in time order, we would produce an incorrect ‘wait-for’ graph. This is a similar problem to how the OS implements barriers, as we discussed in class. The solution then was to use sense reversal, where each thread maintains a local ‘sense’ of the state of the barrier and only proceeds once the true state matches their ‘sense.’ We implement a similar solution with a global *barrier_iterations* variable and local *thread_iterations* variables. The first thread that enters the barrier initiates the barrier and creates edges between the barrier and all other threads. Other joining threads get processed normally. Then, the first releasing thread removes dependencies between the barrier and all other threads and increments *barrier_iterations*. When other threads release the barrier, they will have a *thread_iterations* value smaller than the global *barrier_iterations* value. These threads simply increment their local *thread_iterations* value and move on, as their dependencies were already moved by the first releasing thread. This way, we correctly manage the edge case, allowing for correct and consistent ‘wait-for’ graphs.

In Figure 7, we have an example output deadlock graph. Here, thread 0 grabs the lock and enters the barrier. Then, thread 2 start waiting on the lock. This creates a deadlock since thread 0 will not release the lock until it exits the barrier, but it cannot exit the barrier until thread 2 acquires the lock and enters the barrier. The resulting cycle is highlighted in red.

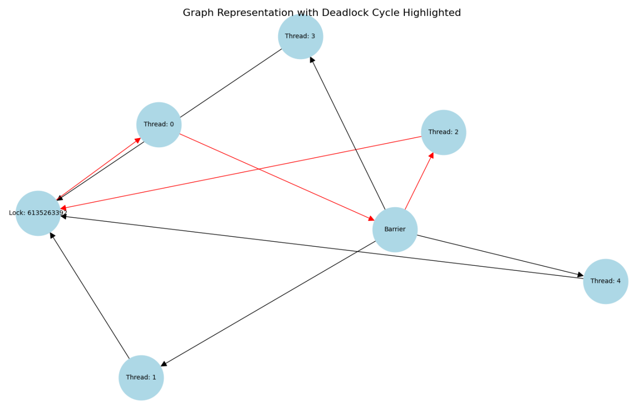


Fig. 7. Deadlock Detector Example Graph

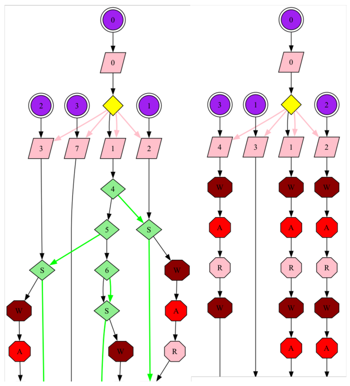
## 5 Case Study: "Parallel VLSI Wire Routing via OpenMP"

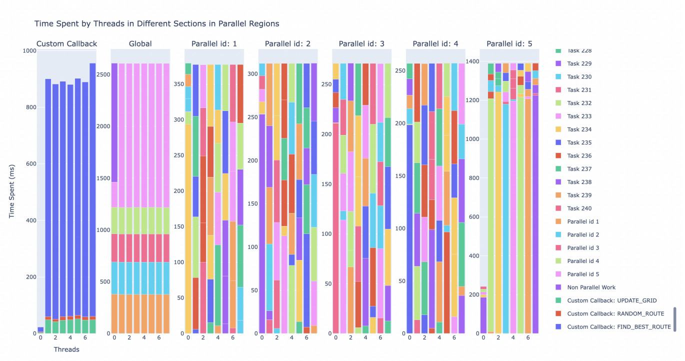
We wanted to look at some possible applications of our tool. Here, we look at a few different approaches to solving Assignment 3's Wire Routing and how our tool can painlessly show off the trade offs between these approaches. We mainly look at variants of the "across wires" approach, where each thread picks off a batch of wires to compute and uses a lock to write to shared memory.

### 5.1 Task-based Approach

One possible approach we examined was a task-based approach. Here, instead of each thread grabbing a batch of wires, thread 0 instead creates an explicit task for each batch. Then, any available thread will context switch to compute the batch and update the wire routing. On the left in Figure 8, we see the task graph generated by this approach as opposed to the standard approach. In the task-based approach thread 0 immediately begins creating tasks, which other threads are then scheduled to complete. This is reflected in the associated performance graph. The five different parallel ids each represent a simulated annealing iteration. Each bar segment represents a different task. Using the graph, it's clear which tasks take extra time and could be separated into multiple tasks to optimize workload imbalance.

On the leftmost subplot of the performance graph, we see the custom callback timings. In blue is work done to find the best route, in red is work done to simply set a random route, and in green is work done to update the occupancy matrix. It's clear that most of the work being done is calculating the best route and therefore should be the main focus when optimizing this code. An interesting tidbit is thread 0 seems to do little to no work. This is because it spends all its time generating explicit tasks! This can also be seen in each parallel id subplot, where thread 0 works on significantly fewer tasks than any other thread and instead spends most of its time in its implicit task. The overhead of creating explicit tasks can be seen as a major downside to this approach. On the other hand, the work is quite evenly distributed among the threads as the scheduling is essentially handled by the OS.


(a) Task-based (left) vs Across-wires (right)



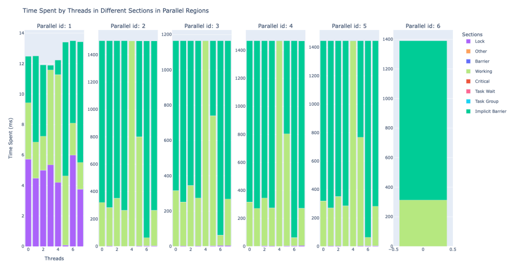
(b) Performance Graph

Fig. 8. Assignment 3 Task Based Approach

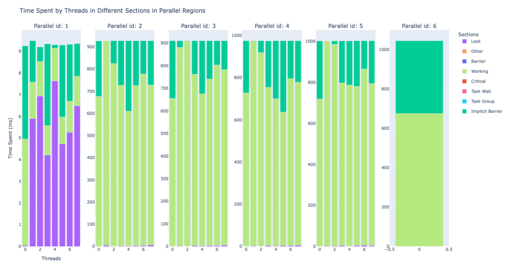
### 5.2 Static Allocation of Wires

Another possible approach to wire routing is statically assigning each thread a set of wires to work on. This approach is practically required for Asssignment 4's MPI solution, since threads have access to much less communication in a message passing environment. As shown in Figure 9, we first randomly assigned each thread a set of wires. Here, the light green represents a thread working and the purple represents waiting for lock acquisition. The dark green represents waiting on an implicit barrier. When exiting a parallel region, all threads must reach the end together before continuing. This means that threads that finish their work early sit in the implicit barrier until other threads catch up. In this example, we clearly see some threads like thread 4 receive a more than 10x workload as compared to other threads like thread 6. This is consistent across each annealing iteration.

In an attempt to rectify this imbalance, we sort the wires by their size. Then we alternate allocate each thread one wire at a time starting from the biggest wire down to the smallest wire. This significantly balances out workload between threads resulting in much faster runtimes. The approach of statically assigning wires does have its benefits. For one, its simpler to implement and debug. Secondly, it results in very low lock contention as compared to the next approach.



(a) Random allocation of wires



(b) Sorted allocation of wires

Fig. 9. Static Assignment of Wires

### 5.3 Dynamic Allocation of Wires

The approach most students implement and the one outlined in the writeup is a dynamic allocation of wires. Each thread grabs a batch of wires, computes new routes for those wires, then goes back and grabs another batch. In Figure 10, we run dynamic allocation with a low batch size. We see a near perfect workload distribution, with no thread waiting long in an implicit barrier. However, this approach leads to high contention on the lock. Due to the low batch size, threads are constantly finishing work and moving to grab a new batch of wires to work on. As shown through these examples, COMPASS allows users to quickly and painlessly examine the trade offs of different approaches, as well as identify important regions of code to optimize.

## 6 Future Works and Limitations

### 6.1 Usability Across Platforms

For now, the tool is designed specifically to run on M-series Macs, as that is the machine both team members own. In addition, the tool is built around LLVM's clang compiler and their implementation of OMPT. This was initially

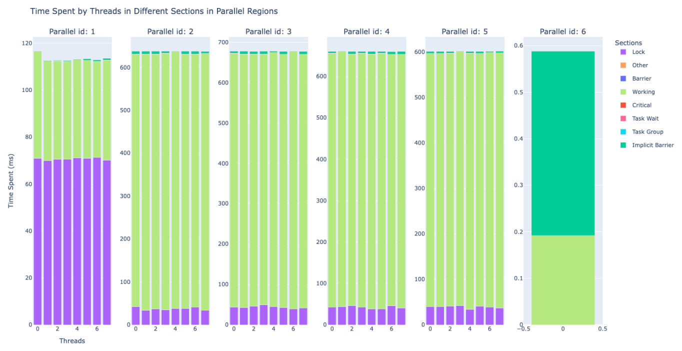


Fig. 10. Dynamic Assignment of Wires

chosen as clang offers greater support to OpenMP and OMPT compared to the gcc compiler. We would love to expand COMPASS to be usable by everyone including Linux, Window and gcc compiler users.

### 6.2 Tracing Tools

COMPASS currently relies on asynchronous logging to text files. While better than synchronous logging, it introduces measurable latency into the program and affects the accuracy of performance measurements. Other similar tools we have looked into like Otter[2] use a tracing tool called OTF2 (Open Trace Format 2). We believe this could slightly improve the logging latency and, as a result, the accuracy of our tool.

### 6.3 Nested Parallel Regions

In an OpenMP program, it is possible to spawn parallel regions inside of parallel regions. However, we found the resulting logs of nested parallel regions to be incredibly confusing to parse with little consistency across multiple program executions. Thus, we implemented our tool with the assumption that the user would not be using nested parallel regions. This is not a completely unreasonable assumption as nested parallel regions are supported by the default in OpenMP [4]. If given more time, though, we would love to work towards a solution that properly logs events in nested parallel regions.

### 6.4 Data Access Pattern Analysis

An important part of performance analysis and optimization is proper memory usage. Thus, a great extension onto our tool would be tracking how long threads spend doing arithmetic work vs memory operations (i.e., arithmetic intensity). We also would like to add another feature of correctness analysis through race condition checking. Both of these features would require the incorporation of other tools like Linux's perf or Intel's VTune. Creating data access pattern analysis would be a logical next step in COMPASS's development.

[4]One must first call omp_set_nested(1) to toggle nested parallelism on.

## 7 Distribution of Credit

Total credit should be split evenly between the two group members.

Work completed by Jack:

(1) Built Task-based Performance Analysis Graph
(2) Built Synchronization-based Performance Analysis
(3) Built Deadlock Detector
(4) Worked on running experiments with Assignment 3
(5) Wrote the project proposal and midway report
(6) Wrote the majority of the final report

Work completed by Luca:

(1) Built the Parallel Execution Graph
(2) Implemented asynchronous logging
(3) Worked on running experiments with Assignment 3
(4) Designed the poster
(5) Wrote parts of the final report

## References

[1] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. 2013. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In *Proceedings of the 9th International Workshop on OpenMP (IWOMP 2013) (Lecture Notes in Computer Science, Vol. 8122)*. Springer-Verlag, 171–185. https://doi.org/10.1007/978-3-642-40698-0_13

[2] Adam Tuft. 2021. Otter: An OMPT Tool for Tracing and Visualising OpenMP Tasks. In *CIUK 2021: Computational and Information UK Conference*. Durham University. https://tobiasweinzierl.webspace.durham.ac.uk/wp-content/uploads/sites/288/2022/02/2021_CIUK.pdf Accessed: 2024-12-15.