Overcoming Distributed Debugging Challenges in the MPI+OpenMP Programming Model

Lai Wei[†], Ignacio Laguna[*], Dong H. Ahn[*], Matthew P. LeGendre[*], Gregory L. Lee[*]
[*]Lawrence Livermore National Laboratory, [†]Department of Computer Science, Rice University

I. Introduction

There is a general consensus that exascale computing will employ a wide range of programming models to harness the many levels of architectural parallelism [1], including models to exploit parallelism in CPUs and devices, such as OpenMP. To aid programmers in managing the complexities arising from multiple programming models, debugging tools must enable programmers to identify errors at the level of the programming model. However, the question of what the effective levels for debugging in hybrid distributed models are, remains unanswered. In this work, we present a novel framework to build an intuitive stack trace view of MPI+OpenMP programs. We develop a methodology to reconstruct call stacks for OpenMP threads and share our lessons learned from incorporating OpenMP awareness into a highly-scalable, lightweight debugging tool for MPI applications: the Stack Trace Analysis Tool (STAT) [2]. Our framework leverages OMPD [3], an emerging debugging interface for OpenMP, so that we can evaluate the effective levels of debugging for MPI+OpenMP. Our easy-to-understand stack trace views help users debug MPI+OpenMP programs at the user code level by mapping the stack traces to the high-level abstractions provided by programming models.

II. Problem Statement

Although OpenMP is commonly used in shared-memory parallel programs, debugging OpenMP programs is challenging. For example, if we run the OpenMP program shown in Fig. 1, attach a debugger to it, and print the stacks of all threads when they are in the sleeping state, the result would be what we see in Fig. 2. This example illustrates two challenges when debugging OpenMP programs. First, OpenMP worker threads don't have stack frames generated before their thread creation, providing only partial calling context. Second, OpenMP runtime libraries generate stack frames that are not part of the user code, which could confuse users when debugging. To address these challenges, a debugger needs to retrieve additional information from the OpenMP runtime library through OMPD.

In addition, we must address two debugging challenges in the MPI+OpenMP programming model. First, the debugger needs to collect information efficiently from thousands, if not millions, of OpenMP threads which distribute among different MPI processes. Second, the debugger needs to provide the user with an intuitive representation to help him pinpoint a bug from this huge thread pool.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. [LLNL-ABS-676023]

```
7  void bar()
8  {
9    #pragma omp parallel for num_threads(2)
10   for (int i = 0; i < 10; i++)
11     sleep(18);
12 }
13
14 void foo()
15 {
16   omp_set_nested(1);
17   #pragma omp parallel num_threads(2)
18   {
19     if (omp_get_thread_num() == 0)
20       sleep(90);
21     else
22       bar();
23   }
24 }
25
26 int main(int argc, char *argv[])
27 {
28 // MPI_Init(&argc, &argv);
29   printf("Application: Process %d started.\n", getpid());
30   foo();
31 // MPI_Finalize();
32   return 0;
33 }
```

Fig. 1. An Example OpenMP Program

III. Background

OMPD [3] is an emerging debugging interface—encapsulated in a shared library—that enables debuggers to understand the state of an OpenMP program and the OpenMP runtime in a live process or a core file. Debuggers can interact with OMPD to get information, such as a thread's current status or a thread's current parallel region or task region. To construct the full calling context of OpenMP threads, one needs to exploit OMPT task inquiry analogues in OMPD. The OMPT technical report [4] has a good explanation about how OMPT task inquiries work.

The Stack Trace Analysis Tool (STAT) [2] gathers and merges stack traces from a parallel application's processes. It can help users quickly locate problems in a large MPI application. While it has a good support for MPI applications, stack traces gathered for MPI+OpenMP applications are inaccurate due to the aforementioned challenges in OpenMP debugging. Therefore, we develop a call stack reconstructing framework for OpenMP threads and implement it in STAT to provide intuitive stack trace view of MPI+OpenMP programs.

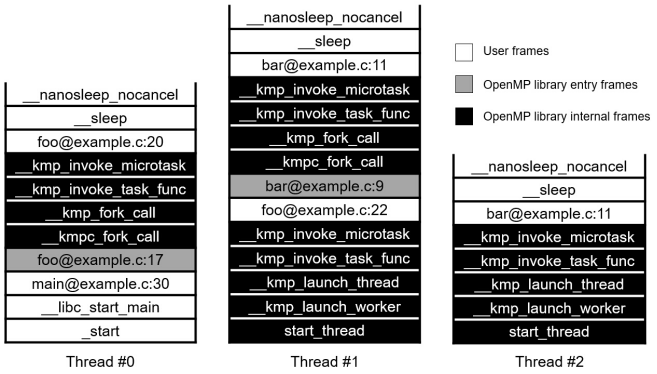

Fig. 2. Stacks of OpenMP threads for the code shown in Fig. 1. We divide stack frames into three categories. User frames corresponds to routines in the user program. OpenMP library entry frames are user frames that enter a new parallel construct. OpenMP library internal frames are created by routines inside an OpenMP library.

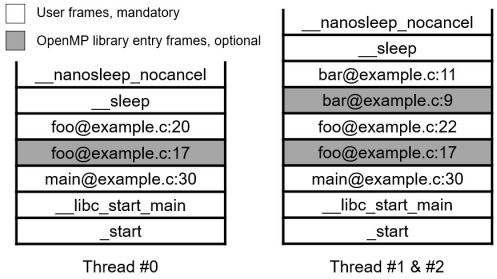

Fig. 3. Reconstructed call stacks of OpenMP threads for the code shown in Fig. 1. User code frames are always presented to the user while OpenMP library entry frames are optional depending on the user's choice.

IV. Approach

We first implement an OpenMP stack walker that uses OMPD and DynInst API [5] to rebuild call stacks for OpenMP threads. Fig. 3 shows the result when we attach the OpenMP stack walker to the sample program in Fig. 1. Our OpenMP stack walker successfully reconstructs the full calling context and removes OpenMP library internal frames.

We then update STAT to collect stack traces of OpenMP threads with the OpenMP stack walker. When we enable all MPI related lines in our example program as shown in Fig. 1, we obtain a result from STAT as shown in Fig. 4.

V. Summary and Future Work

In our work, we leverage OMPD to reconstruct call stacks for OpenMP threads in STAT. Our results show that we are able to present users with intuitive stack trace views for MPI+OpenMP programs.

As future work, we plan to refine STAT call stack views for MPI+OpenMP programs and evaluate STAT on large MPI+OpenMP applications. We also want to extend our framework to support debugging of OpenMP 4.0 programs, which include target constructs to express parallelism in devices. We aim to explore new use cases for our tool, such as presenting views of OpenMP tasks as well as identifying master and worker threads in MPI+OpenMP programs.

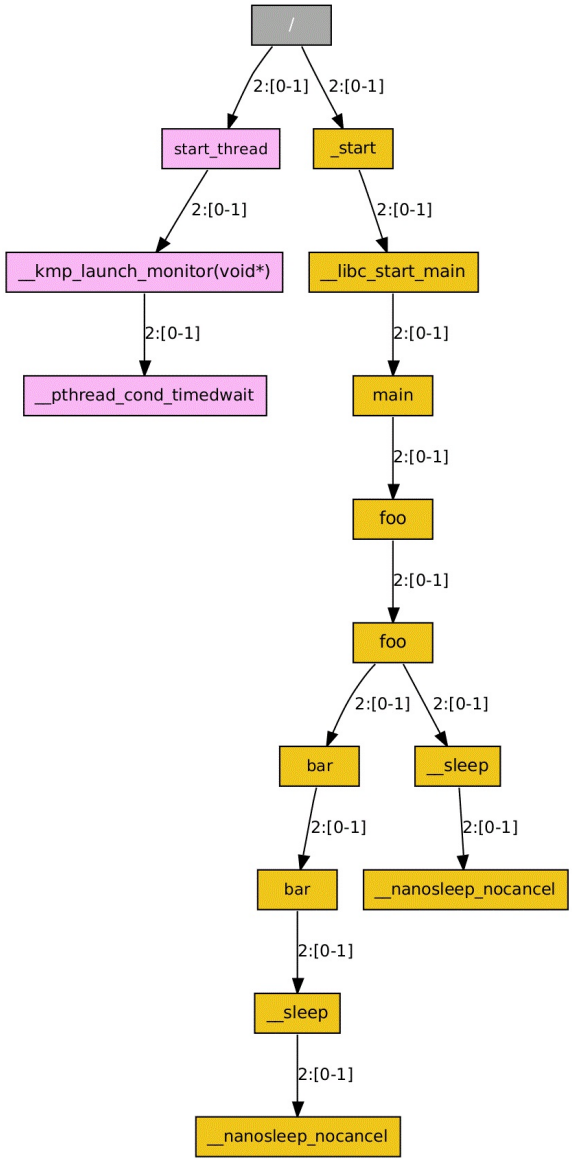

Fig. 4. STAT 2D spatial call graph for the example program shown in Fig. 1 after enabling all MPI related lines. The result corresponds to two MPI nodes where there are three OpenMP threads and one OpenMP helper thread per node.

References

[1] Jack Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[2] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[3] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. Cownie, R. Dietrich, J. Signore, E. Loh, and D. Lorenz. OMPD: An application programming interface for a debugger support library for OpenMP. Technical report, 2014.

[4] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. Cownie, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OpenMP technical report 2 on the OMPT interface. Technical report, 2014.

[5] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. Toward the deconstruction of dyninst. Technical report, 2007.