A Deadlock Detection Concept for OpenMP Tasks and Fully Hybrid MPI-OpenMP Applications

Tobias Hilbrich*, Bronis R. de Supinski†, Andreas Knüpfer*, Robert Dietrich*
Christian Terboven‡, Felix Münchhalfen‡, Wolfgang E. Nagel*

*Technische Universität Dresden, D-01062 Dresden, Germany
Email: {tobias.hilbrich, andreas.knuepfer, robert.dietrich, wolfgang.nagel}@tu-dresden.de

†Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
Email: bronis@llnl.gov

‡RWTH Aachen University, D-52056 Aachen, Germany
JARA – High-Performance Computing, D-52062 Aachen, Germany
Email: {muenchhalfen, terboven}@itc.rwth-aachen.de

ABSTRACT

Current high performance computing applications often combine the Message Passing Interface (MPI) with threaded parallel programming paradigms, e.g., OpenMP. MPI allows fully hybrid applications in which multiple threads of a process issue MPI operations concurrently. Little study on deadlock conditions for this combined use exists. We propose a wait-for graph approach to understand and detect deadlock for such fully hybrid applications. It specifically considers OpenMP 3.0 tasking support to incorporate OpenMP's task-based execution model. Our model creates dependencies with deadlock criteria that can be visualized to support comprehensive deadlock reports. We use a model checking approach to investigate wide ranges of valid execution states of example programs to verify the soundness of our wait-for graph construction.

1. INTRODUCTION

Increasing core counts per compute node challenge purely MPI-based parallel programming. Recent compute architectures target the use of MPI along with a threading paradigm. Each process then uses multiple threads to exploit all available compute cores. OpenMP [6] is often used to introduce threads into MPI programs [1, 5, 7]. It includes a task paradigm and leads to the following nomenclature:

Processes are spawned by MPI, execute the same program, have distinct virtual memory, communicate via messages, and differ in integer *ranks*;

Threads are spawned and managed by OpenMP, they share memory within each process; and

Tasks are created and managed on each process by the OpenMP runtime, and execute on threads.

The `MPI_THREAD_MULTIPLE` support level, which we call *fully hybrid*, allows threads within a process to issue concurrent MPI operations. We provide a first investigation of deadlock conditions for these fully hybrid programs.

1.1 Lower Thread Support Levels

For hybrid OpenMP-MPI applications that are not fully hybrid, the following property holds:

> *Any application deadlock can be detected either as a pure OpenMP deadlock or as a pure MPI deadlock.*

This property has the following consequences:

- Deadlock detection for thread support levels lower than `MPI_THREAD_MULTIPLE` can be handled by independent OpenMP and MPI deadlock detection approaches, i.e., separate tools for OpenMP and MPI deadlock detection could execute side-by-side without interacting.
- Fully hybrid deadlock detection has to combine knowledge about OpenMP and MPI primitives in the most general case.

2. FULLY HYBRID DEADLOCKS

To analyze whether a fully-hybrid application is deadlocked, we construct wait-for graphs that capture wait-for dependencies of execution states of these programs. This construction must consider processes, threads, and tasks. The input for the construction is an execution state of an application. Our basic construction rules are:

(I) The AND⊕OR WFG [4] $(V, E_{\text{AND}}, E_{\text{OR}})$ for the state includes one node $i \in V$ for each process i, to which we refer as a *virtual* process node; and

(II) The virtual node of a process i waits with *OR* semantics for all of its tasks $t_0^i, t_1^i, \ldots, t_l^i$, i.e., a process is deadlocked if no task of the process can progress $(\forall t_k^i \in \{t_0^i, t_1^i, \ldots, t_l^i\} : (i, t_k^i) \in E_{\text{OR}})$.

MPI primitives consider processes, not tasks/threads. With that the virtual process nodes serve to represent dependencies of MPI primitives. If a task t^i is active in an MPI primitive and waits for a process j, we can model this by adding (t^i, j) to either E_{AND} or E_{OR}. Such a dependency can then in turn be satisfied by any task on process j. The *OR* semantic arcs of process j to its tasks (rule (II)) then models that any task of process j can satisfy the incoming MPI wait-for condition.

3. OPENMP TASKS AND DEADLOCKS

OpenMP Threads can switch the executing task by selecting an available and unscheduled task at *task scheduling points*, while adhering to OpenMP task scheduling rules. A task that is not scheduled cannot remove wait-for dependencies, which our WFG construction captures with:

(III) A task t^i of a process i that is not scheduled in the current execution state has a wait-for condition of the *AND* semantic to process i, i.e, $(t^i, i) \in E_{\text{AND}}$.

Thus, tasks that are not scheduled have a wait-for dependency that ensures that they become deadlocked if their associated process node is deadlocked.

OpenMP allows a task switch at *task scheduling points*. We must consider this when we construct WFG arcs for a task that is at a task scheduling point:

(IV) If any task of a process is at a task scheduling point, we consider all currently schedulable tasks as scheduled, irrespective of the number of threads in the team, i.e., we do not apply rule (III) for them.

Based on our construction rules we can then model individual dependencies that result from OpenMP or MPI directives onto their originating tasks.

4. EVALUATION

To show that our wait-for graph construction rules are correct, we compare whether their resulting graphs are correct for example programs. We compare the capabilities of the graph-based deadlock detection to tools such as Intel Inspector XE [2] and MUST [3].

Capturing global consistent execution states for fully hybrid applications is involved. Thus, we use models of example application to capture their valid execution states. A scheduler[1] handles this task. For our example programs, our construction rules do not create false positives/negatives and overcome limitations of Intel Inspector XE and MUST.

5. REFERENCES

[1] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost. Employing Nested OpenMP for the Parallelization of Multi-zone Computational Fluid Dynamics Applications. *J. Parallel Distrib. Comput.*, 66(5):686–697, 2006.

[2] S. Blair-Chappell and A. Stokes. *Parallel Programming with Intel Parallel Studio XE*. John Wiley & Sons, 2012.

[3] T. Hilbrich, B. R. de Supinski, W. E. Nagel, J. Protze, C. Baier, and M. S. Müller. Distributed Wait State Tracking for Runtime MPI Deadlock Detection. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13, pages 16:1–16:12, New York, NY, USA, 2013. ACM.

[4] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A Graph Based Approach for MPI Deadlock Detection. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 296–305, New York, NY, USA, 2009. ACM.

[5] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. *International Parallel and Distributed Processing Symposium*, pages 919–932, 2013.

[6] OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[7] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.

[1] `https://fusionforge.zih.tu-dresden.de/plugins/mediawiki/wiki/mpi-omp-sched`