SAND2022-13931C

Received 9 October 2017; | Revised ; | Accepted

DOI: xxx/xxxx

ARTICLE TYPE

Scythe: Composing MPI and OpenMP on top of a common runtime substrate

Noah Evans | Richard F. Barrett | Jacob P. Caswell

[1]Sandia National Laboratories, Albuquerque, New Mexico, USA

Correspondence
*Noah Evans Email: nevans@sandia.gov

Present Address
Sandia National Laboratories, P.O. Box 5800, MS 1319, Albuquerque, NM 87185-1319

Summary

The combination of MPI and X, two distinct mechanisms for enabling parallelism, is of significant interest as a means of extracting the performance potential from current, emerging, and expected future high performance computing architectures. A major challenge for achieving the desired performance improvement is the coordination of the two runtime systems, in particular as a means of avoiding oversubscription of a core. Additionally, the distinct separation of computation and communication induces synchronous execution, eliminating opportunities for progressing each asynchronously. We have developed a system, called Scythe, based on Lithe, that coordinates the interaction of MPI and OpenMP. In this paper we describe its impact using a miniapp from the Mantevo suite, executing on two computing systems, one composed of Intel Xeon Phi Knights Landing manycore processors, the other Intel Xeon Haswell multicore processors. This work illustrates the value of the coordination of MPI and OpenMP, and illuminates areas for further research as well as changes to OpenMP necessary for enabling this approach.

KEYWORDS:
high performance computing, parallel programming, runtime library, concurrency control

1 | INTRODUCTION

The ubiquitous means for implementing applications for execution on high performance computing systems is for each parallel process to be defined as an MPI rank, pinned to a single processor core. Increasing compute node core counts, increasingly capable node interconnects, and other capabilities has led to the exploration of alternative means for extracting parallelism from applications. OpenMP is a popular means for enabling on-node parallelism: it is specified by an open forum, it is strongly supported by vendors, it is easy to understand, and it provides mechanisms for addressing performance issues. Therefore the integration of MPI for internode communication and OpenMP for intranode data sharing provides an appealing combination.

However, since each has its own runtime system, use of the scheduler is not coordinated, and therefore competition for it will result in one thrashing while it waits for another to complete. Often this competition is unnecessary since a runtime is acting as if it owns all of the resources on the system where it could be sharing them. This is mitigated to some extent by the operating system which can schedule a new process at the end of the quanta and penalize processes that use all of the quantum. However, this behavior is detrimental to the performance of applications, especially those with high computational intensities. To prevent this while not sacrificing latency, runtimes often have set conditional back off when waiting, i.e. after a certain number of busy waiting cycles the runtime returns control to the operating system.

This leads to the issue of dynamic configuration, i.e. how often runtime resources change over the course of its execution. Therefore it becomes important that applications have a way of communicating these resource needs dynamically over the course of the execution of an application, with

[0]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

MPI handing off resources to OpenMP, and vice versa. Our work described herein targets the "+" in MPI + X. That is, it serves as the coordinator of intra- and inter-node data sharing.

This paper is organized as follows. After a discussion of related work and some background material, we describe our implementation, which we call Scythe. With this, we present some performance results, using a miniap from the Mantevo suite, executed on two distinct, yet related in important ways, computing environments. We end with a summary and discussion of our future work.

1.1 | Related work

The combination of MPI + X is an active area of research. Runtime integration and managing over-subscription in MPI applications has been studied extensively in terms of how to effectively manage resources. The primary approach is to use communication between cores as a way for things to talk to each other (1).

User level libraries also make it possible to control the resource management between systems. Exemplified by Quo (2), these systems let the user manually specify the resource sharing between the different runtimes during individual stages of the application. In Quo's case this is done by pushing cpu configurations (cpusets) onto a stack and popping back to previous configurations as stages have finished.

Another approach attempts to make things work together by having the runtimes communicate their needs to each other. The Process Management Interface (PMIX) allows additional runtimes to communicate their resources needs to the PMIX which then arbitrates resource needs between the MPI implementation and any hybrid runtimes that need access to resources that are oversubscribed. This is currently under examination by an MPI working group for possible inclusion in future specifications.

Maximizing the effectiveness of this approach by better integrating MPI implementations with task based runtimes has been a subject of considerable research effort. One approach, overdecomposition, attempts better overlap communication and computation by "overdecomposing" work into the smallest possible task size and then relying on the runtime scheduler to handle individual task management. This is the approach taken by MPI-Q (3) and fine grained MPI (4) which use coroutines to manage the concurrency inherent in MPI applications. Typically they are done by integrating MPI with a task based programming model (5) or multithreaded MPI (6)

However these approaches do not address *cross runtime crosstalk* in MPI applications. Traditionally when runtimes are combined with an MPI implementation it is up to the application programmer to ensure that work is evenly distributed between MPI and other runtimes. This process is often made more difficult by the assumption that many runtimes (and MPI) make that they are the sole user of system resources.

There has been considerable research in methods to compose ostensibly agnostic runtimes such as Lithe (7) and Hiper (8). Newer approaches to dealing with cross talk in MPI include the Quo (2) library which allows systems to manually specify cpu affinity and allocation between multiple runtimes in order to be able to ensure that cross runtime cross talk does not interfere with the communication and computation phases of the application.

Finally, there is an approach which attempts to use a common runtime substrate to make it possible for global runtime resource management. This requires that all of the runtimes being composed are modified to share the same underlying runtime substrate. Individual runtimes ask the substrate for resources which are then allocated by the underlying substrate. This is the approach used in this paper, specifically using the Lithe (7) runtime composition library.

2 | SCYTHE

Scythe combines MPI, OpenMP, and Lithe (7) into a single coordinated means for extracting parallelism from application programs. Some modifications to MPI and OpenMP were necessary, described below, but the APIs of each remain unchanged.

The OpenMPI threading model was configured to use Lithe rather than its default pthreads. To do this, we defined an API in `opal/src/mca/threads` which can be selected at the instantiation of the Open Runtime Environment (`orte`). We generalized the threading implementation of the Open Portable Access Layer (opal) in OpenMPI. Currently the implementation of threading in opal assumes a pthreads and Unix scheduling implementation.

Pthread calls are sprinkled throughout the code base and the implementation of threading at `opal/threads/thread.c` assumes a pthreads API implementation. We removed most of these assumptions, notably removing a call to `pthread_atfork` in opal init as well various instances of `sched_yield` in the opal progress threads. In the actual threading implementation we also removed all references to pthreads thread local storage and the relevant calls to pthreads create and join mechanisms.

The changes necessary for transferring OpenMP runtime responsibilities are provided in the Lithe distribution. The Lithe port of OpenMP is based on libGOMP from the GCC 4.4 trunk (implementing OpenMP 3.3) (7). Pthreads are replaced by the Lithe hart abstraction and team tasks

are given a set up of reusable process contexts which execute each task on a separate hart. Scheduling is done using a gang fork join model where groups of harts are requested at the beginning of a parallel region and joined using Lithe's fork join scheduler when a parallel region is complete.

2.1 | Lithe Overview

The Lithe (7) runtime system makes it possible to compose runtime systems hierarchically in the manner of hierarchical cpu scheduling (9). Individual runtimes are modified to use Lithe instead of their traditional threading mechanism which allows Lithe to mediate between their different resource needs.

Lithe also makes it possible to nest this composition, providing a global user level scheduler which then passes control to various runtimes which can then pass control to child runtimes, block or execute their own computations. Using these scheduling mechanisms runtimes can call other runtimes and provide them the necessary resources for these runtimes to operate efficiently.

To provide this mediation Lithe provides a common mechanism for sharing cores between runtimes. The pinned core threads are called "Harts" and prevent over-subscription between different runtimes running on the same machine. This is achieved by ensuring exclusive non-preemptable access to individual cores by a particular runtime. Harts must be explicitly released and requested by individual runtimes depending on the needs of a runtime at a given phase of computation.

Underlying this idea is the assumption that each runtime is acting in good faith with respect to the rest of the system and not requesting more resources than it needs. In particular this means that individual runtimes must rely on Lithe's scheduling mechanisms as much as possible rather than busy waiting for resources. Without any preemption mechanism, a malicious runtime is capable of starving the other runtimes on a system.

Using this composition mechanism, it is possible to compose MPI and OpenMP processes to inter-operate automatically. MPI progress threads are impelemented as Lithe Harts and scheduled such that they cannot interfere with OpenMP task teams. Using these facilities, it is possible to interleave communication and computation in a much finer granularity.

Therefore much of the work of implementing sharing between the runtimes is ensuring that when each runtime can no longer do useful work (e.g. OpenMPI is waiting for data or an OpenMP team is done with a compuation). Wait state behavior in the runtime must correctly yield and/or block rather than the spin waiting until its scheduler time slice is completed.

For example, in the Lithe model instead of yielding to the scheduler after busy waiting a short time for input, the MPI implementation yields immediately to the runtime for rescheduling at which point any runtime waiting for resources can then take control of that core. This approach is similar to the way that MPIQ (3) reschedules wait states into the Qthreads runtime instead of spin-waiting until a resource is free.

2.2 | Implementing OpenMPI and OpenMP composition with Lithe

To make it possible to compose the OpenMPI and OpenMP runtimes with Lithe, we needed to change OpenMPI to use Lithe's abstractions rather than its traditional pthread model.

Lithe is implemented using two fundamental abstractions, harts described in 2.1 and contexts – continuations representing the state of a runtime at a scheduling event. The contexts can be saved, restored and passed to other schedulers for execution.

Lithe threads are implemented using the parlib (10) library, which provides a common threading abstraction that can be used across multiple operating systems (notably Akaros (11) and 64 bit Linux). Parlib provides functionality similar to a traditional user level threading system, including thread local storage and cooperatively scheduled coroutines.

Using parlib as a base, Lithe implements harts as the representation of individual hyperthreads of a system (derived from `get_nprocs()` in linux implementations) by pinning each thread to a core.

Hierarchical schedulers are implemented on top of these threads as structures of function pointers of callback functions which provide a set of common behaviors, specifically how to request and cleanup hardware threads, handling of child processes and finally blocking, unblocking, yielding and exiting coroutines.

This combination of features allows conformant runtimes to be composed by forming a tree of schedulers which arbitrates between the different runtimes and allows these runtimes to use other runtimes by delegating their allocated resources to whichever library it provisions work to.

In turn, this allows the overlapping of computation and communication. When a runtime wishes to wait for an event (such as I/O), it blocks its scheduler which relinquishes its hardware thread to another runtime which then uses it until it no longer needs the CPU, which prevents the system from oversubscribing cpu resources.

Lithe provides two default schedulers: the base scheduler which acts as a set of defaults (effectively acting as the parent scheduler of all other runtimes), and a "fork join" scheduler which allows basic fork join parallelism similar to cilk (12) and other fork join based runtimes.

Lithe also provides a modified version of libGOMP (13) as its OpenMP implementation. The Lithe libGOMP substitutes pthreads based threads teams for hart based teams and implements a custom scheduler to reuse contexts between thread team computations.

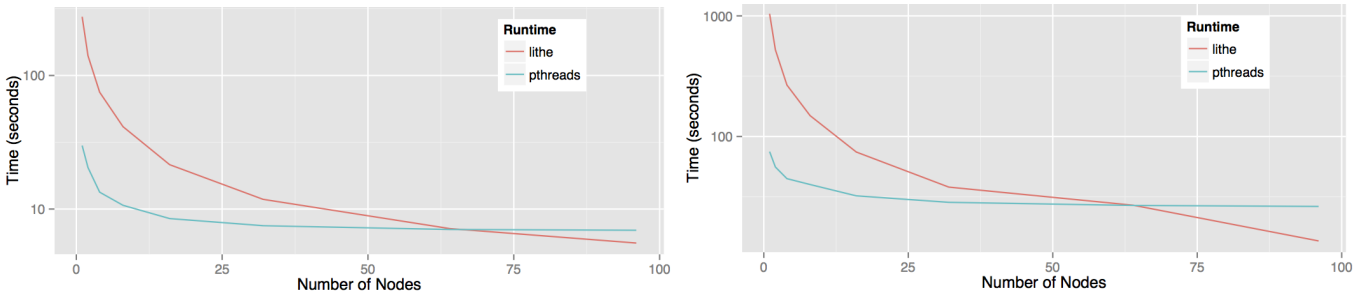

FIGURE 1 miniFE performance on Mutrino
The figure on the left is from the Mutrino Haswell partition. The figure on the right is from the KNL partition.

Thus, much of the effort in composing MPI and OpenMP is changing an MPI implementation to support Lithe's hart model. To make these changes we modified OpenMPI (14) by removing the assumption of pthreads from its Open Portability Layer (opal) and substituting a Lithe threading model. With the assumption of pthreads support thus removed we used Lithe's fork join scheduler implementation to implement similar runtime aware functionality in Opal. Since OpenMPI's threading model also relies heavily on thread local storage, we used the thread local storage implementation from parlib to satisfy OpenMPI's thread local storage semantics.

In addition to these superficial changes there are other modifications necessary to support Lithe thread semantics in OpenMPI. Lithe contexts differ from traditional pthreads or lightweight threading models in that they make no assumptions about their locations and state of their stacks (similar to the more primitive `clone` system call in Linux). To ensure that each thread maintains its own state information and its context is independent, we `mmap` each stack into the address space of the parent MPI process.

3 | PERFORMANCE EVALUATION

We evaluated Scythe using a miniapp from the Mantevo suite[1]. MiniFE is designed to serve as a proxy for unstructured implicit finite element based applications (15). In particular, it is designed to represent the Krylov subspace iterative solver for a finite element discretization on unstructured meshes. It assembles finite element matrices into a global matrix and vector, then solves the linear system using the Conjugate Gradient (CG) method. The problem is structured for simplicity, but that structure is not exploited.

Experiments were run on Mutrino, a Cray XC40. Mutrino consists of two distinct partitions, based on two different processors. The first partition consists of Intel Xeon Haswell processors, configured as 16 core dual socket nodes, 128 GBytes of DDR4 memory, and a 2.3 GHz clock. The second partition consists of Intel Xeon Phi Knights Landing processors, configured as 16 core dual socket nodes, 96 GBytes of memory, with a 1.4 GHz clock. Each partition consists of 100 nodes, connected using the Cray Aries interconnect. Compute nodes are configured using the Cray CLE, based on SLES running Rhine/Redwood, runing a Lightweight Operating System (LWOS).

The performance of miniFE, operating on a 256 x 256 x 256 grid, is illustrated in Figure 1 . At small node counts, Lithe's overhead degraded performance. However, as node counts increased, our coordinated approach continued to scale where as the uncoordinated approach remained flat. The cross over point on the Mutrino Haswell and KNL partitions occurred at 64 nodes. Figure 2 illustrates the time spent in the computational phases when executing on the Haswell partition. Total time includes the finite element assembly resulting in the linear system. Total CG is further broken down by the time spent in the matrix-vector products, dot products, and vector updates.

4 | SUMMARY

The combination of MPI and OpenMP is a desirable means for extracting parallelism on high performance computing resources. However, MPI and OpenMP use distinct runtimes, preventing coordination of the activities necessary for extracting the performance potential from those resources. Scythe takes over many of the responsibilities for managing the threads, resulting in improved performance as the number of MPI processes increases.

[1] http://mantevo.org

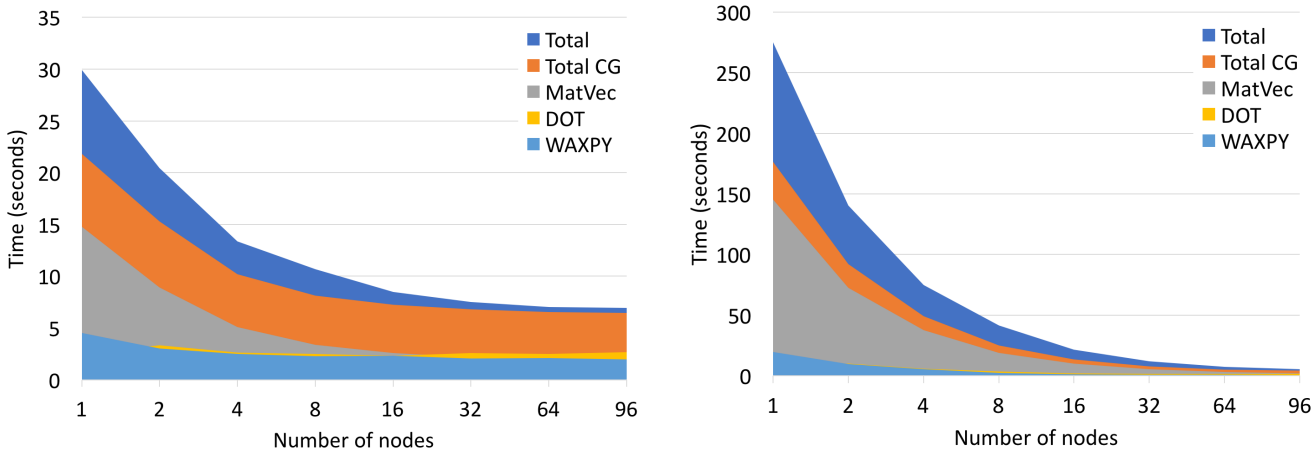

FIGURE 2 miniFE phases performance

The figure on the left uses standard MPI + OpenMP. The figure on the right uses MPI + OpenMP coordinated using Scythe.

MPI and OpenMP APIs required no changes to accomplish this. Some modifications to MPI and OpenMP were of course necessary to transfer thread management responsibilities to Lithe. We modified OpenMPI for these purposes and used the Lithe version of OpenMP, based on gcc. However, the necessary changes could be incorporated into any MPI and OpenMP implementations.

We explored the performance impact of this approach using miniapps from the Mantevo suite, designed to represent common algorithms found in scientific and engineering applications. At small node counts, Lithe's overhead degraded performance. However, as node counts increased, our coordinated approach continued to scale where as the uncoordinated approach remained flat. The cross over point on the Mutrino Haswell occurred at 64 nodes.

4.1 | Future work

Moving the threading implementation to an OpenMPI Modular Component Architecture (MCA) module would make it possible to choose between the traditional OpenMPI pthreads implementation or a custom threading implementation at runtime. This would also benefit OpenMPI by moving calls to `sched_yield` into the opal runtime.

Our current implementation of progress threads is invasive. For example, generalizing the yield and other behavior would make it possible to do the necessary work easily.

Further, the OpenMPI implementation would benefit from greater use of fine grained parallelism. The current implementation makes assumptions about various parts of the system that would benefit from being addressed in a finer grained manner. Incorporating many of these approaches in the MPI runtime (e.g. yielding to the runtime instead of spin waiting) would make it easier to implement fine grained approaches using a variety of runtimes in MPI.

MiniFE provided straightforward means for exploring these initial ideas. We intend to explore other scheduling mechanisms and other applications, for example Particle in Cell (PIC) and Adaptive Mesh Refinement (AMR) codes. The complexity of these algorithms is expected to provide even more opportunities for Scythe to avoid oversubscription.

References

[1] Rabenseifner Rolf, Hager Georg, Jost Gabriele. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: :427–436IEEE; 2009.

[2] Gutiérrez Samuel K., Davis Kei, Arnold Dorian C., et al. Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. In: ; 2017; Orlando, Florida.

[3] Stark Dylan T, Barrett Richard F, Grant Ryan E, Olivier Stephen L, Pedretti Kevin T, Vaughan Courtenay T. Early experiences co-scheduling work and communication tasks for hybrid MPI+ X applications. In: :9–19IEEE Press; 2014.

[4] Kamal Humaira, Wagner Alan. An integrated fine-grain runtime system for MPI. *Computing*. 2014;96(4):293–309.

[5] Chatterjee Sanjay, Tasirlar Sagnak, Budimlic Zoran, et al. Integrating asynchronous task parallelism with MPI. In: :712–725IEEE; 2013.

[6] Si Min, Peña Antonio J, Balaji Pavan, Takagi Masamichi, Ishikawa Yutaka. MT-MPI: multithreaded MPI for many-core environments. In: :125–134ACM; 2014.

[7] Pan Heidi, Hindman Benjamin, Asanović Krste. Composing parallel software efficiently with lithe. *ACM Sigplan Notices.* 2010;45(6):376–387.

[8] Grossman Max, Kumar Vivek, Vrvilo Nick, Budimlić Zoran, Sarkar Vivek. A Pluggable Framework for Composable HPC Scheduling Libraries. In: :723–732IEEE; 2017.

[9] Ford Bryan, Susarla Sai. CPU inheritance scheduling. In: 22, vol. 96: :91–105; 1996.

[10] Klues Kevin Alan. *OS and Runtime Support for Efficiently Managing Cores in Parallel Applications.* University of California, Berkeley; 2015.

[11] Rhoden Barret, Kles Kevin, Zhu David, Brewer Eric. Improving per-node efficiency in the datacenter with new OS abstractions. In: :25ACM; 2011.

[12] Blumofe Robert D, Joerg Christopher F, Kuszmaul Bradley C, Leiserson Charles E, Randall Keith H, Zhou Yuli. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing.* 1996;37(1):55–69.

[13] Novillo Diego. OpenMP and automatic parallelization in GCC. *the Proceedings of the GCC Developers Summit.* 2006;.

[14] Gabriel Edgar, Fagg Graham E, Bosilca George, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: :97–104Springer; 2004.

[15] Lin Paul T, Shadid John N. Towards large-scale multi-socket, multicore parallel simulations: Performance of an MPI-only semiconductor device simulator. *Journal of Computational Physics.* 2010;229(19):6804–6818.