Implementing Flexible Threading Support in Open MPI

Noah Evans
Sandia National Laboratories
Livermore, CA, USA
nevans@sandia.gov

Jan Ciesko
Sandia National Laboratories
Albuquerque, NM, USA
jciesko@sandia.gov

Stephen L. Olivier
Sandia National Laboratories
Albuquerque, NM, USA
slolivi@sandia.gov

Howard Pritchard
Los Alamos National Laboratory
Los Alamos, NM, USA
howardp@lanl.gov

Shintaro Iwasaki
Argonne National Laboratory
Lemont, IL, USA
siwasaki@anl.gov

Ken Raffenetti
Argonne National Laboratory
Lemont, IL, USA
raffenet@mcs.anl.gov

Pavan Balaji
Argonne National Laboratory
Lemont, IL, USA
balaji@anl.gov

***Abstract*—Multithreaded MPI applications are gaining popularity in scientific and high-performance computing. While the combination of programming models is suited to support current parallel hardware, it moves threading models and their interaction with MPI into focus. With the advent of new threading libraries, the flexibility to select threading implementations of choice is becoming an important usability feature. Open MPI has traditionally avoided componentizing its threading model, relying on code inlining and static initialization to minimize potential impacts on runtime fast paths and synchronization. This paper describes the implementation of a generic threading runtime support in Open MPI using the Opal Modular Component Architecture. This architecture allows the programmer to select a threading library at compile- or run-time, providing both static initialization of threading primitives as well as dynamic instantiation of threading objects. In this work, we present the implementation, define required interfaces, and discuss trade-offs of dynamic and static initialization.**

***Index Terms*—MPI, MCA, threading, hybrid programming**

I. Introduction

The end of Dennard scaling has led to an increase in node-level parallelism to achieve higher computational performance. Particular representatives of this trend are multicore processors and accelerators that together with a suitable programming model expose the additional, node-level parallelism to the developer.

The rise of these architectures has in turn increased the importance of hybrid programming models in which node-level programming models such as OpenMP [26] are coupled with Message Passing Interfaces (MPI). This is commonly referred to as *MPI+X*. In such cases, MPI is used to implement inter-node work coordination and data exchange. Sophisticated programs utilizing hybrid parallel programming commonly attempt to overlap communication and computation to achieve high performance, thus require the use of both programming models and their runtime libraries at the same time.

To increase the efficiency of such hybrid approaches, a considerable amount of research has been focused on resource management and scheduling. Central to an efficient coexistence of runtimes is the need to avoid potential resource conflicts between threads used by the MPI runtime and the node-level parallel programming model such as OpenMP. The MPICH MPI implementation [12] has made progress in this area, and a corresponding effort is required for Open MPI so that both widely-used MPI implementations can support hybrid MPI+X applications efficiently. While differences in software architectures require different solutions, a common design goal towards modular and intentional support for thread management is desirable for both.

Additional drivers for updating the Open MPI threading layer are new mechanisms anticipated in upcoming revisions of the MPI standard, like *sessions* [15]. These capabilities are designed to better support the composability of applications and libraries in coupled simulations with multiple binaries coexisting on a node.

In this use case, coordination of thread resource management among different MPI libraries is essential. Several studies have shown that the use of threading models in composed runtimes can lead to large speedups in MPI performance [2], [7], [13], [17], [22], [24], [30]. However, Open MPI [11] has reduced the number of threading models available in its implementation. Implementors have given priority to optimizing Open MPI internals, in particular, asynchronous progress [10], in the context of the *Pthreads* library at the expense of other threading models.

This development is understandable from a historical perspective. Open MPI supported multiple thread libraries in the past, including Windows Threads and Solaris Threads. However, later, as threading models settled on the Pthreads interface as a common substrate, Open MPI made Pthreads the exclusive threading model and removed support for other legacy threading models. The need for this support is driven by the development of novel, lightweight user-level threading libraries (*ULT*) that promise better support of on-node parallelism.

For a motivational insight, we measured the cost of the *Fork-Join* and *Yield* routines for three different threading libraries shown in Figure 1. Although performance differs due to

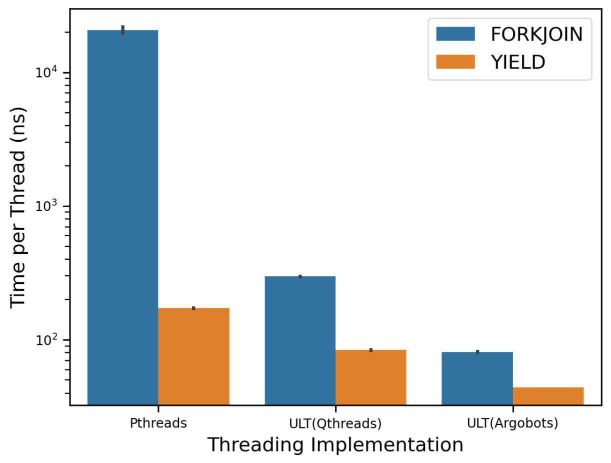


Fig. 1. Results obtained from timing frequent and successive calls to *Fork-Join* and *Yield* on an Intel Xeon E5-2699 CPU (2.30GHz) hint towards significant performance differences between Pthreads and user-level threading libraries for certain cases.

distinct design, optimization, and specialization for certain workloads, this result underlines the importance of a modular threading support. In this case, the fork-join overheads of the ULT implementations are up to 100 times smaller than that of Pthreads while their yield overheads are two to four times smaller.

Workarounds to Open MPI's narrow focus on Pthreads exist. One way to support threading libraries has been to wrap MPI calls in threading models via the `MPI_T` interface or similar means [13], [17], [30]. This indirect approach makes threading models difficult to deploy in production systems, which either need to use multiple compiled MPI libraries and different sets of modules [9] or force the users to compile custom versions of MPI for personal use.

Part of the reluctance to modularize threading library support in MPI implementations arises from concerns about the cost of thread synchronization. Synchronization primitives and related functions are in the critical path of important runtime functionality and even minor slowdowns can be detrimental to performance. The conventional way to avoid these overheads is to use techniques such as inlining of functions and initializing data structures like mutexes statically, at compile-time, rather than relying on dynamic initialization and allocation. However, as modern CPUs and compilers show improvements in branch prediction and calling performance of indirect functions, it is worthwhile to reassess these limitations and to determine whether they are still obstructing a design towards modularity, composability, and ultimately, usability.

Interestingly, unlike threading support, other functional components in Open MPI are modular. For example, a wide breadth of interchangeable options exists to handle collective operations. Consequently, the goal of this work is to modularize threading in Open MPI and to allow the exchange of threading libraries with little effort, comparably to switching collective algorithms and other Open MPI framework components. To that end, this paper makes the following contributions:

- Design towards modular support of threading libraries in Open MPI
- Its initial implementation
- An evaluation of calling overheads

We would like to point out that recently our changes to Open MPI have been incorporated into the master branch through an accepted pull request[1]. While performance results presented to peer-reviewers of the pull request were critical for its acceptance, we believe that a discussion of this effort, some implementation details, and results are relevant to an interested reader as well.

II. Background

The requirement for heterogeneous parallel programming has led to significant efforts to research component architectures in distributed and high-performance computing. The goal is to reduce the need for duplication of code, performance optimizations, and numerical algorithms when used in different but comparable contexts. One of the most successful outcomes from this effort is the *Common Component Architecture* which used ideas, components, and interactions similar to CORBA [31]. Research into the performance cost of component architectures has quantified the performance costs of componentizing an HPC framework to be the equivalent of two indirect function calls [1]. In practice, those costs may be deemed acceptable based on software requirements and use-cases. Open MPI [11] is a product of a movement towards component architectures, specialized to the needs of the HPC community [21]. Open MPI avoids the performance overhead of some traditional component architectures by avoiding their distributed and cross-language capabilities which are not required in this setting. Instead, Open MPI uses a "bare metal" component architecture using C ABIs (application binary interfaces) and dynamic shared objects. This approach minimizes overheads in a performance-sensitive environment [4]. Open MPI's Modular Component Architecture (MCA) allows collectives [21], transports [11] and other MPI functionality to be chosen at runtime, either automatically or as specified by the user.

Despite exhibiting negligible performance overheads in many scenarios, the implementation of threading libraries in Open MPI has not been implemented as an MCA component. Instead, threading is implemented using static data initializers and function inlining. The reason for this divergence is that threading is often in the critical path of many MPI API calls were even two indirections can lead to unacceptable performance.

III. Design Goals

The *Open, Portable Access Layer* (OPAL) is a basic abstraction layer in Open MPI responsible for low-level functionality

[1]https://github.com/open-mpi/ompi/pull/6578

at the process level. To implement generic threading support, we propose the addition of a new OPAL MCA component. This component defines a generic API to interface with threading implementations in particular for thread management, thread synchronization, and thread-local storage. This approach follows the current structure of the OpenMPI threading implementation but avoids direct calls to the Pthreads library.

We list specific design goals as follows.

- Allow the selection of threading libraries for Open MPI at compile or runtime.
- Provide a common interface to threading libraries.
- Provide a common integration framework for scheduling events like MPI wait-states and synchronization yields.
- Allow the coexistence of multiple threading libraries by avoiding a shared state.
- Allow multiple threading components to coexist within applications using multiple MPI sessions.
- Implement MCA threading support incrementally, allowing static linking against threading libraries (hybrid approach).

While the presented implementation targets all the abovementioned goals, not all design goals are further discussed and evaluated in this work. In this work, we focus on the design of the MCA component, its hybrid implementation and performance implications resulting from the use of shared threading libraries.

IV. Implementation

Our reference implementation promotes threading support to an interchangeable MCA module. The MCA threading API is defined as follows. Firstly, a set of header files located in `mca/threads/*.h` define the generic API for thread abstractions and interfacing with the Open MPI MCA infrastructure. They provide all required interfaces to support mutexes, condition variables, thread-local storage, and opaque handles that are used to determine thread identity.

API calls to performance-critical functionality are declared as *static inline* but do not provide a definition (implementation). This requires to provide the corresponding definitions of those API calls at compile-time. Header files that provide definitions correspond to particular threading models and are organized in a directory structure as `mca/threads/<threading_model>/threads_<threading_model>_*.h`. Performance-critical API definitions of a given threading model such as `*_threads.h`, `*_mutex.h`, `*_tsd.h` must be declared with the same decorators and are inlined into their calling context. Non-performance critical API calls are not inlined and their definitions are provided at load time of a shared object file.

This design allows for a gradual shift towards the runtime selection of threading models. However, following requirements formulated by the HPC community, at least in the short term, threading models must be selected at compile time rather than using runtime options for this purpose. The table in Figure 2 gives an overview of the threading API

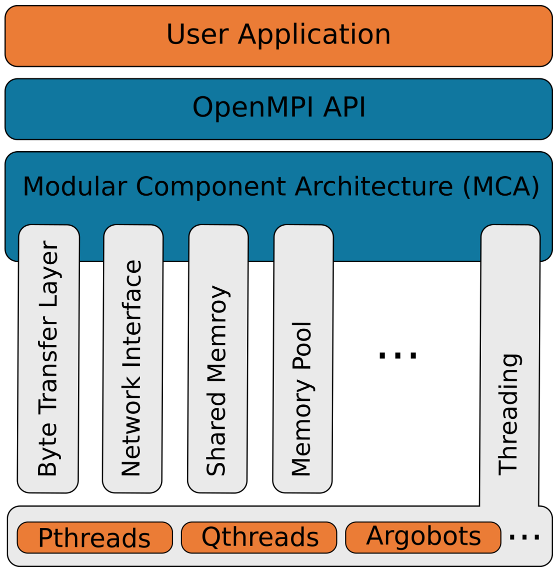


Fig. 2. Visual representation of the OPAL MCA architecture shows several MCA modules including the newly added threading support.

and marks those calls implemented as a shared library with a corresponding entry in the column named `S`.

To select a particular threading model, the configure option `--with-threads=<threading_model>` can be provided during project configuration. Currently we have added support for Pthreads and the *QThreads* [32] and *Argobots* [28] threading models. To the interested reader, we provide further insights into required code changes and considerations below. A performance evaluation of this architecture is presented in the next chapter.

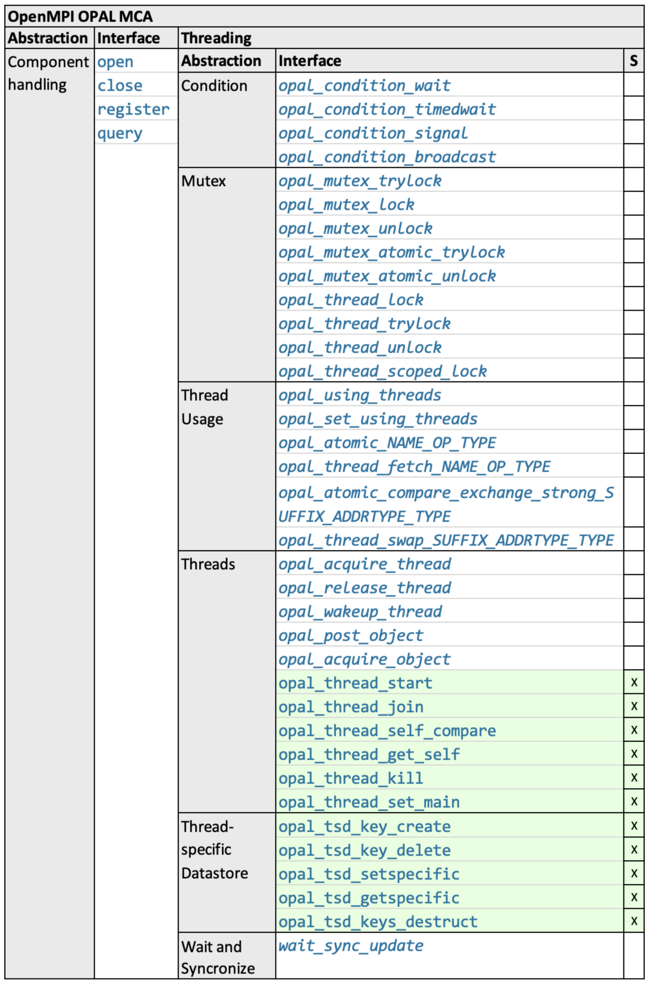
A. Other Code Changes

The original Open MPI implementation assumed that threading was a static, top-level OPAL portability library. Therefore, all previous references to Pthreads had to be changed to the MCA threading header files `mca/threads/*.h`. This includes all occurrences of direct Pthread usage in the Open MPI code base. In particular, process management and event handling (e.g., the OPAL PMIx module and Libevent support) were using Pthreads calls directly and these calls needed to be factored out and placed behind the common OPAL interface.

B. Further Considerations

The implemented MCA threading support requires considering implications on handling thread identity, event handling, and key-value management. We discuss this as follows.

The proposed threading architecture maintains assumptions from the OPAL implementation regarding the nature of thread identity. In OPAL threads are identified using opaque handles and compared via special functions. Our reference implementation maintains that assumption. However, this requirement places a certain burden on scalable threading libraries that have no concept of thread identity to maintain a minimal amount



OpenMPI OPAL MCA				
Abstraction	Interface	Threading		
		Abstraction	Interface	S
Component handling	open	Condition	opal_condition_wait	
	close		opal_condition_timedwait	
	register		opal_condition_signal	
	query		opal_condition_broadcast	
		Mutex	opal_mutex_trylock	
			opal_mutex_lock	
			opal_mutex_unlock	
			opal_mutex_atomic_trylock	
			opal_mutex_atomic_unlock	
			opal_thread_lock	
			opal_thread_trylock	
			opal_thread_unlock	
			opal_thread_scoped_lock	
		Thread Usage	opal_using_threads	
			opal_set_using_threads	
			opal_atomic_NAME_OP_TYPE	
			opal_thread_fetch_NAME_OP_TYPE	
			opal_atomic_compare_exchange_strong_SUFFIX_ADDRTYPE_TYPE	
			opal_thread_swap_SUFFIX_ADDRTYPE_TYPE	
		Threads	opal_acquire_thread	
			opal_release_thread	
			opal_wakeup_thread	
			opal_post_object	
			opal_acquire_object	
			opal_thread_start	x
			opal_thread_join	x
			opal_thread_self_compare	x
			opal_thread_get_self	x
			opal_thread_kill	x
			opal_thread_set_main	x
		Thread-specific Datastore	opal_tsd_key_create	x
			opal_tsd_key_delete	x
			opal_tsd_setspecific	x
			opal_tsd_getspecific	x
			opal_tsd_keys_destruct	x
		Wait and Syncronize	wait_sync_update	

Fig. 3. The threading module exposes a common API that is subdivided into different abstraction groups. Calls to shared libraries are marked in the respective fields of the column *S*.

of state. The implications of this fact are to be considered in future work.

To simplify the implementation of asynchronous progress, Open MPI uses the Libevent library [23] to handle asynchronous progress and polling for event management functionality. Note that Libevent is not involved in the critical path for sending and receiving MPI messages. Libevent relies on a set of custom callback functions to implement object allocation and synchronization with its parent runtime system. Our threading implementation generalizes the previous Pthreads interface and requires that any compatible threading runtime provides a set of callback functions and wrappers to ensure that Libevent is properly integrated into the imported runtime system. However, as Libevent currently assumes preemption and calls to the `sleep()` routine when yielding, a custom yield function is needed to exploit task context switching in user-level threading runtimes. Designing and evaluating this support is subject to future work.

Open MPI relies heavily on thread-local storage which is used even without `MPI_THREAD_MULTIPLE`. Thread local storage is used in the Open MPI's Open Run-Time Environment (ORTE) and the OPAL Unified Communication X (UCX) [29], hardware locality (hwloc) [5] and Process Management Interface - Exascale (PMIx) [6] modules. Therefore, each threading interface must provide an `opal_tsd_key` module to implement thread-local storage.

V. Evaluation

For evaluation, we use the Blake and Voltrino clusters at Sandia National Laboratories[2] and the Lassen supercomputer at Lawrence Livermore National Laboratory[3]. Blake is a 40 node Intel Xeon Platinum (Skylake) 8160, 2.10GHz - based cluster with two sockets per node and 24 cores per socket connected over the Intel OmniPath interconnect. Voltrino is a Cray XC40m supercomputer consisting of 24 nodes of two Intel Xeon E5-2698 v3, 2.6GHz (Haswell) processors with 16 cores per socket equipped with an Intel Xeon Phi 7250 processor with 68 cores with 4-way SMT per node. The Lassen is an IBM POWER9-based supercomputer installed at the Lawrence Livermore National Laboratory. It consists of 684 nodes with two POWER9, 2.3 GHz processors per node with 22 cores each.

A. Methodology

For benchmarking we use the function call overhead measurement benchmark *FNbench* [4] and a multi-threaded benchmark for MPI's one-sided communication interface *RMA-MT* [8]. It is designed to measure the threading impact on latency and throughput between two MPI processes. Both benchmarks were compiled with the Intel C compiler (18.2.199) and the GCC compiler (gcc/7.3.1) and manually disassembled to ensure expected code generation and linking.

The function call measurement application microbenchmark uses the same technique from [4], a library function with an empty body called either from a shared or static library. We use the CPU time stamp counter instruction (`rdtsc`) that returns an approximated cycle count of the execution of the kernel that calls the empty body function 10^9 times. We output the cycle count normalized to one loop iteration. The performance of this microbenchmark should only depend on the cost of an L1 instruction cache hit and the `callq` and `retv` instructions in the static case while the shared library's trampoline function and offset function should exercise to the function implementation.

To test the performance impact of these calls on threading libraries within Open MPI, we used the RMA-MT benchmark suite. The RMA-MT benchmarks measure latency, bandwidth, and bi-directional behavior between a pair of MPI processes. RMA-MT benchmarks create a user-specified number of threads where each thread performs a single or multiple `MPI_Put` or `MPI_Get` operation(s) thus effectively measuring latency and bandwidth. A master thread handles all synchronization. Locking, flushing, post-start-complete-wait (PSCW), or fencing behavior can be chosen at runtime.

[2]https://www.sandia.gov/asc/computational_systems/HAAPS.html
[3]https://hpc.llnl.gov/hardware/platforms/lassen

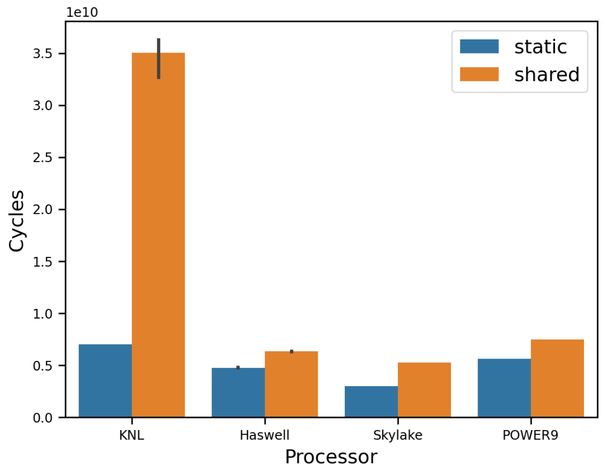


Fig. 4. Function call overhead for one billion function calls as measured by the *FNbench* benchmark. On systems with high single-thread performance, calls to shared libraries are on average 1.5x slower than calls to static libraries. Multicore systems like the Xeon Phi, show a higher performance penalty of 5.8x for calls to shared libraries.

For testing we use the Pthread module from the MCA threading library. Since the reference implementation of modular threading is a hybrid implementation where the threading model is chosen and statically linked at compile-time as described previously, it was necessary to factor out the currently inlined synchronization functions and move them to the Pthread threading module. We compare this shared library implementation against the original hybrid implementation to evaluate if the cost of calling overhead is sufficiently large to cause a performance penalty.

B. Results

1) FNbench: Figure 4 shows the calling overhead for one billion function calls obtained from the function call measurement application microbenchmark on the aforementioned architectures. On systems with high single-thread performance, the use of shared libraries results in a performance slowdown of 1.7x on the Intel Xeon Skylake and 1.3x on POWER9. Executions on simpler cores that are typical for large multicore processors like the Xeon Phi show a performance penalty of approximately 5.7x for shared library calls.

A more detailed analysis of the generated assembly code shows that performance results on the POWER9 proportionally correspond to the increased number of instructions in case of calling a shared library. Figure 5 shows the assembly code of the test function *noop* of the FNbench benchmark compiled as a shared library and how the return address for the branch instruction (*bctr*) is computed. The assembler code of the static library contains only one branch instruction (*blr*) to an address specified in the link register. This additional code in the case of the shared library accounts for approximately 30% of the total code executed in one loop iteration of the benchmark.

```
void __attribute__ ((noinline)) noop() {
    asm volatile("":::);
}

100005c0 <00000038.plt_call.noop>:
  100005c0: 18 00 41 f8 std r2,24(r1)
  100005c4: 10 81 82 e9 ld r12,-32496(r2)
  100005c8: a6 03 89 7d mtctr r12
  100005cc: 20 04 80 4e bctr
```

Fig. 5. Assembly code of the *noop* test function in *FNbench* compiled as a shared library for the IBM POWER9 processor shows the explicit computation of the return address and the branch instruction. This is opposed to the single branching instruction generated by the compiler in case of calling a static function.

This corresponds to the proportionally higher instructions per iteration count and the resulting slowdown shown in Figure 4.

The assembly code of the same test function compiled as a shared library for Intel's instruction set architecture shows a single return instruction that internally implements the return semantics. However, its implementation accounts for a significant increase in instructions per iteration of the benchmark application which is proportional to the slowdown as reported previously.

Even though the assembly code for both, the Intel Xeon and Xeon Phi are similar, the Xeon Phi shows a significant performance difference. The Xeon Phi misses 49% of branches leading to a 4.2x loss of branch throughput (684.4 M/sec compared to 161.8 M/sec) as opposed to 0.02% branch misses on the Intel Xeon Skylake. Further, the variation in observed cycles per iteration expressed as the standard deviation in 4 is potentially due to variations of library placement in virtual address space which can affect performance [18].

2) RMA-MT: To test parallel MPI with the proposed architecture, we invoke the RMA-MT benchmark as below.

```
mpirun --np 2 --map-by ppr:<1,2>:node
 --bind-to socket \
    rmamt_<bw,lat> -x -t <num_threads>\
     -o put -s fence
```

This allows us to measure unidirectional *put*-performance with fencing synchronization to exercise the effects of locking behavior on the MPI implementation and to observe any changes relating to function call behavior.

Figure 6 shows the transport layer configuration parameters for these experiments. The RMA-MT benchmark consists of the bandwidth and latency applications (rmamt_bw and rmamt_lat). We show performance results, both in terms of bandwidth (MB/sec) and latency (microseconds) in Figures 7, 8, 9, and 10.

Given the overheads of using a shared library function calls, we expected a similar performance penalty when switching to shared libraries in RMA-MT. However, this is not the case. Shared libraries are within the margin of error for most cases and are slightly faster on Intel Xeon Phi and Haswell architectures.

System	Transports
Intel Xeon Phi	`Cisco usNIC: no`
Intel Haswell	`Cray uGNI (Gemini/Aries): yes`
Voltrino.sandia.gov	`Intel Omnipath (PSM2): no`
	`Open UCX: no`
	`OpenFabrics OFI Libfabric: no`
	`Portals4: no`
	`Shared memory/copy in+copy out: yes`
	`Shared memory/Linux CMA: yes`
	`Shared memory/Linux KNEM: no`
	`Shared memory/XPMEM: yes`
	`TCP: yes`
Intel Skylake	`Cisco usNIC: no`
Blake.sandia.gov	`Cray uGNI (Gemini/Aries): no`
	`Intel Omnipath (PSM2): yes`
	`Open UCX: yes`
	`OpenFabrics OFI Libfabric: yes`
	`Portals4: no`
	`Shared memory/copy in+copy out: yes`
	`Shared memory/Linux CMA: yes`
	`Shared memory/Linux KNEM: no`
	`Shared memory/XPMEM: no`
	`TCP: yes`
IBM POWER9	`Cisco usNIC: yes`
Lassen.llnl.gov	`Cray uGNI (Gemini/Aries): no`
	`Intel Omnipath (PSM2): no`
	`Intel TrueScale (PSM): no`
	`Mellanox MXM: no`
	`Open UCX: yes`
	`OpenFabrics OFI Libfabric: yes`
	`Portals4: no`
	`Shared memory/copy in+copy out: yes`
	`Shared memory/Linux CMA: yes`
	`Shared memory/Linux KNEM: no`
	`Shared memory/XPMEM: no`
	`TCP: yes`

Fig. 6. OpenMPI transport layer configuration for RMA-MT experiments

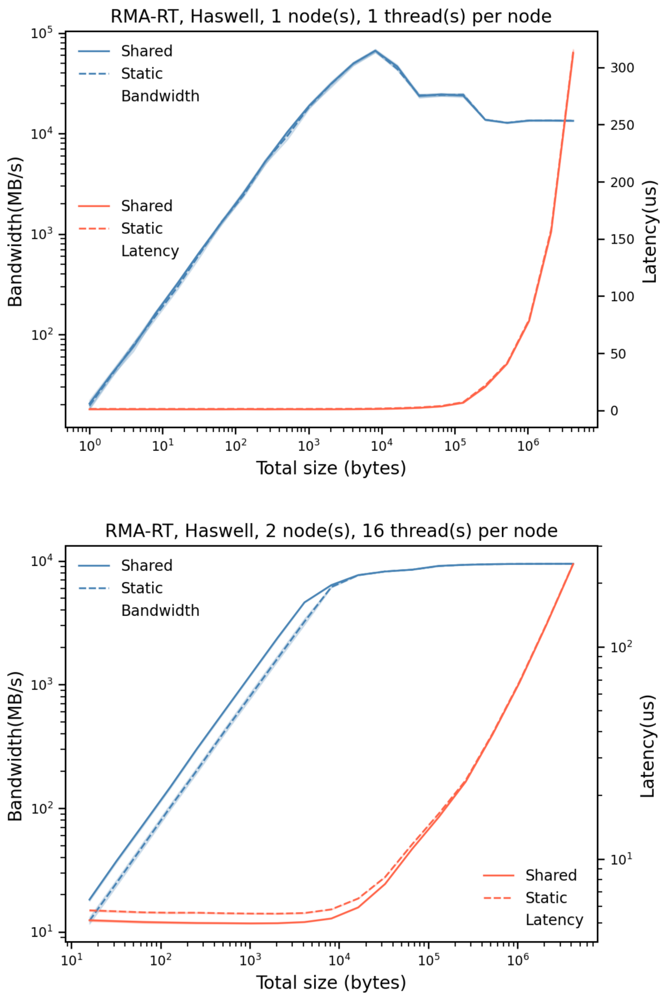


Fig. 7. Executions of the RMA-MT benchmark on th Intel Xeon E5-2698-based system (Voltrino) show a performance increase in terms of bandwidth in case of using shared libraries. Bandwidth peaks at 13,5 GB/sec and 9,47GB/sec for the respective execution scenario.

The reason for the much lower performance penalty of shared library calls in RMA-MT is the amount of work per call. In case of the RMA-MT benchmark, substantially more code is executed per function call, making the overheads less prevalent.

Further, on Haswell, shown in Figure 7, we observed that for 16 threads per process on two nodes, Open MPI spends less time in the UGNI BTL [4] progress engine. However, the performance counters show roughly equivalent values aside for context switches. This is possibly due to changes in user-space mutex (futex) behavior for shared library executions leading to fewer kernel traps. This manifests especially for executions with smaller message sizes with a higher overhead-to-payload ratio.

On the Xeon Phi architecture shown in Figure 8 we see roughly equivalent bandwidth performance but better message latency in case of a shared library. We posit that this is due to better instruction cache locality on less capable nodes. The benefits of inlining are counterbalanced by the lack of cache locality. Concurrency matter more on cores with lower single-threaded performance because there are typically more cores per node leading to greater thread contention. Again, this manifests in the case of smaller messages sizes with a higher overhead-to-payload ratio.

[4]user Generic Network Interface Byte Transfer Layer (BTL)

On the Intel Skylake and IBM POWER9 systems, the additional calling overhead is marginal, as shown in Figures 9 and 10.

C. *Summary and Discussion*

The change in HPC hardware architectures has led to a major difference in the cost of function calls via static or shared libraries, especially as hardware architects attempt to scale performance by adding simpler cores rather than aiming for greater single-thread performance. This is a fundamentally different result when comparing to early work in [4] which showed negligible performance difference on higher performance single-core executions on previous generations of hardware.

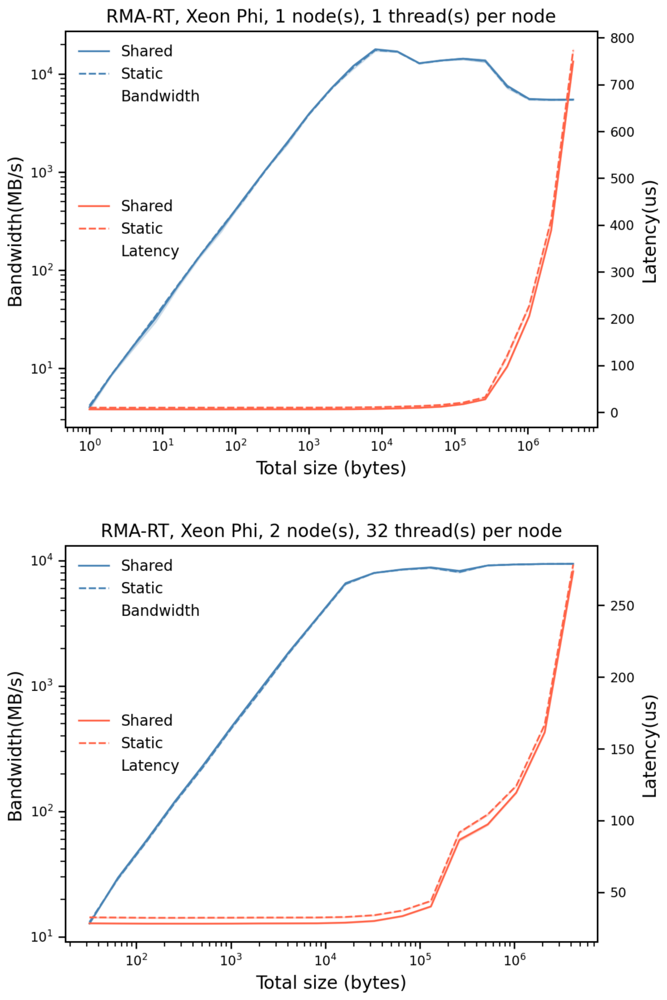


Fig. 8. Performance results obtained on the Intel Xeon Phi 7250-based system (Voltrino) show a minor performance increase in case of using shared libraries. The achieved bandwidth peaks at 5.4GB/sec and 9.42GB/sec for the respective execution scenario.

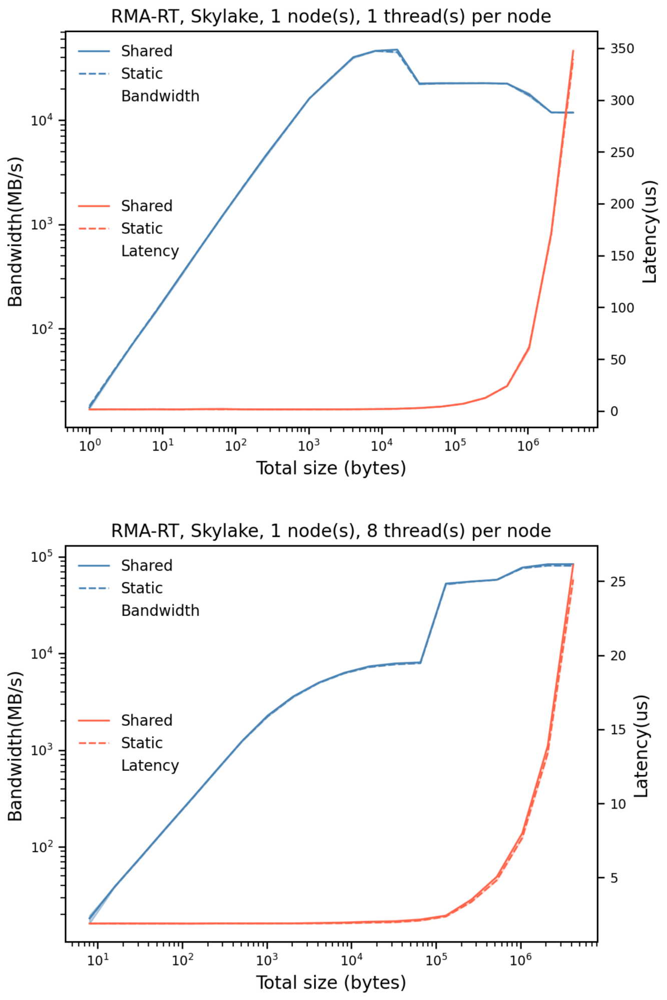


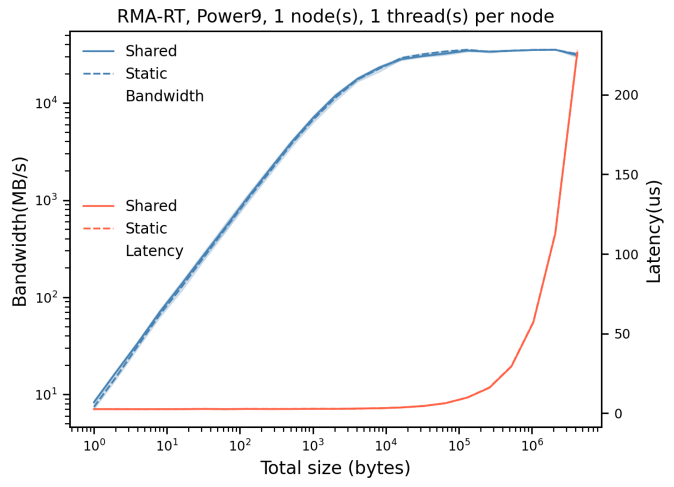
Fig. 9. Performance results obtained from on the Xeon Platinum 8160 system (Blake) show results for single-node executions for 1 and 12 threads. Bandwidth peaks at 11,2GB/sec and 86GB/sec for single and multi-threaded (single node, shared memory BTL) executions respectively.

In practice, shared libraries are comparable to static libraries. The primary driver for this result is the higher amount of work per call in real-world examples as well as differences in caching behavior and fewer context switches. On KNL the smaller instruction cache footprint of shared libraries better utilizes KNL's smaller instruction caches while on Haswell these differences come from better progress behavior due to fewer kernel calls for synchronization.

VI. Related Work

The idea of integrating threading models in MPI is not new. There is extensive literature examining how to manage contention for programs that couple MPI communication and on-node threading using user-level threading runtimes. MPICH+ULT [20] integrates MPICH [12] with the Argobots [28] user-level threading runtime and aims to minimize lock contention. MPIQ [30] uses the Qthreads [32] runtime to co-schedule MPI and application tasks. OMPSs [22] provides an OpenMP-like task model that can interact with MPI. However, these integration mechanisms differ from the presented work. They emphasize the integration of MPI and a particular programming model runtime, while the current work explores the practicality of an initial *decoupling* of MPI and pluggable runtimes.

The composition of pluggable architectures has been addressed by Lithe [27] and HiPER [13]. Each proposes a modular system to compose and integrate multiple runtimes. Lithe provides a common hierarchical scheduling substrate that is the target of runtime shims to mimic the APIs of other runtimes while scheduling them together. HiPER specifies a common module interface and user-level scheduling mechanism that allows users to write wrappers around HPC libraries and co-schedule them through defined module behaviors. Our approach is most similar to the Lithe approach, but rather than implementing a specific scheduling mechanism we provide a shim API to plug in runtimes and access their capabilities



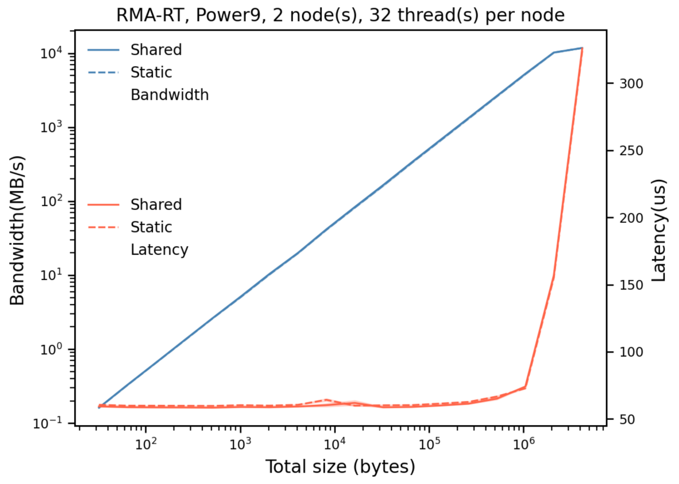


Fig. 10. Performance results obtained from the RMA-MT benchmark on the IBM POWER9 system (Lassen) for two process-to-node mapping scenarios. Results show comparable performance in both cases for shared and statically linked threading libraries peaking equally at a bandwidth of 32GB/sec and 12GB/sec in each scenario.

through a library interface.

An alternative approach is the one used by MassiveThreads [25], Process-In-Process [16] and Lithe's Pthreads [27] front-end. Rather than making a shared shim library, they expose the Pthreads API and implement novel runtimes under it. This scheme fundamentally changes the mechanisms of implementation, promoting kernel-level threading to the user-level. Our approach is compatible with all of these mechanisms. Modularizing the Pthreads MCA makes it possible to add any of these runtimes, potentially making them compatible with proposed MPI standards enhancements like *sessions* [15].

VII. Future Work

To better implement asynchronous progress in the MCA threading module, an improved Libevent library integration with lightweight threading models is desirable. Currently, the Libevent library assumes preemption and does not yield by default. Instead, it enters a sleep state in cases such as waiting for a file descriptor until woken up by the operating system. This behavior interferes with lightweight threading libraries that typically require tasks to be cooperative and to voluntarily yield after some time. It may be necessary to enhance the existing Libevent MCA framework to allow for use of event handling libraries optimized for a particular threading model.

The internal synchronization of Open MPI critical sections also makes assumptions of a Pthreads-based, coarse synchronization model. Finer grained locking of critical sections in the runtime [3], [19] would help support user-level and other fine-grained threading models.

MCA threading support would also benefit from alternative ways to represent thread identity. Savings in the per-thread state would make user-level threading models like Qthreads and Argobots more scalable if individual lightweight threads would no longer require per-thread data structures.

A good test of the robustness of this work would be to integrate MPI Sessions [15]. This integration would exercise the composability of different runtimes, possibly requiring a more substantial runtime integration as in Lithe [27] or Quo [14].

Finally, as seen in Section V, there are still aspects of the performance gains from using shared libraries for thread synchronization that would benefit from further investigation, including instruction cache and futex behavior. Exploring such issues would give a deeper understanding of the trade-offs in the future use of shared versus static libraries in MPI implementations.

VIII. Conclusion

The shift towards heterogeneous and multicore architectures in HPC has made threading and the associated requirement for efficient thread management a challenging problem.

A growing body of research has attempted to address the issue of coordinating application and MPI threading runtimes. However, prior work has neither been designed to fit nor been integrated into a component-based system like Open MPI's MCA. Thus, in practice, the use of threading runtimes remains stuck in a fixed compile-time approach centric to Pthreads or subject to the use of wrappers and other workarounds. In this work, we have described a new MCA architecture enabling the integration of MPI and threading libraries at run time. We also implemented a reference version of the integrated runtime system and evaluated its performance characteristics.

Our evaluation addresses two of the most pressing issues surrounding such componentization of threading in MPI implementations: understanding implications on calling overheads when using shared libraries and performance implications of the proposed solution in a multi-threaded, latency- and bandwidth-sensitive microbenchmark. Our evaluation shows that despite the increasing cost of function calls in shared libraries, especially on many-core systems with slower single-thread performance, shared libraries can perform comparably to static libraries in Open MPI.

Our work on modular threading support can be extended to cover runtime composition and co-scheduling of MPI runtimes using various user-level threading libraries. Because our work has been contributed to the Open MPI code base, the community can further participate in continuous development towards generic support of threading in MPI.

Acknowledgments

Matthew Dosanjh provided valuable feedback on an early draft and advice on RMA-MT. National Nuclear Security Administration under contract DE-NA-0003525. Los Alamos National Laboratory is operated by Triad National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy (contract no. 89233218CNA000001). This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work used computer systems provided by the National Nuclear Security Administration's Advanced Simulation and Computing Program. 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-NA-0003525.

References

[1] R Armstrong, D Gannon, A Geist, K Keahey, S Kohn, L McInnes, S Parker, and B Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*, pages 115–124, August 1999.

[2] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *Recent Advances in the Message Passing Interface*, pages 298–299. Springer Berlin Heidelberg, 2012.

[3] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Fine-Grained multithreading support for hybrid threaded MPI programming. *Int. J. High Perform. Comput. Appl.*, 24(1):49–57, February 2010.

[4] B Barrett, J M Squyres, A Lumsdaine, R L Graham, and G Bosilca. Analysis of the component architecture overhead in open MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 175–182. Springer Berlin Heidelberg, 2005.

[5] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.

[6] Ralph H. Castain, Joshua Hursey, Aurelien Bouteiller, and David Solt. Pmix: Process management for exascale environments. *Parallel Computing*, 79:9 – 29, 2018.

[7] S Chatterjee, S Tasirlar, Z Budimlic, V Cavé, M Chabbi, M Grossman, V Sarkar, and Y Yan. Integrating asynchronous task parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 712–725, May 2013.

[8] M G F Dosanjh, T Groves, R E Grant, R Brightwell, and P G Bridges. RMA-MT: A benchmark suite for assessing MPI multi-threaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559. ieeexplore.ieee.org, May 2016.

[9] John L Furlani. Modules: Providing a flexible user environment. In *Proceedings of the fifth large installation systems administration conference (LISA V)*, pages 141–152. gnu-darwin.org, 1991.

[10] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer Berlin Heidelberg, 2004.

[11] R L Graham, G M Shipman, B W Barrett, R H Castain, G Bosilca, and A Lumsdaine. Open MPI: A High-Performance, heterogeneous MPI. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9. IEEE, September 2006.

[12] William Gropp. MPICH2: A new start for MPI implementations. In Dieter Kranzlmüller, Jens Volkert, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474 of *Lecture Notes in Computer Science*, page 7, Berlin, Heidelberg, September 2002. Springer Berlin Heidelberg.

[13] M Grossman, V Kumar, N Vrvilo, Z Budimlic, and V Sarkar. A pluggable framework for composable HPC scheduling libraries. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 723–732, May 2017.

[14] S K Gutiérrez, K Davis, D C Arnold, R S Baker, R W Robey, P McCormick, D Holladay, J A Dahl, R J Zerr, F Weik, and C Junghans. Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 469–478, Orlando, Florida, May 2017.

[15] N Hjelm, H Pritchard, S Gutierrez, D Holmes, R Castain, and A Skjellum. MPI sessions: Evaluation of an implementation in open MPI. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[16] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. Process-in-process: Techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 131–143, New York, NY, USA, 2018. ACM.

[17] Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F Adams, Francesco Miniati, Chao Mei, and Laxmikant V Kale. Charm++ and MPI: Combining the best of both worlds. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 655–664, 2015.

[18] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016.

[19] H Kamal and A Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. ieeexplore.ieee.org, April 2010.

[20] Huiwei Lu, Sagmin Seo, and Pavan Balaji. MPI+ULT: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications*, pages 444–454. IEEE, 2015.

[21] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of open MPI: Enabling Third-Party collective algorithms*. In Vladimir Getov and Thilo Kielmann, editors, *Component Models and Systems for Grid Applications*, volume 27, pages 167–185. Kluwer Academic Publishers, Boston, 2005.

[22] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

[23] Nick Mathewson, Azat Khuzhin, and Niels Provos. libevent - an event notification library. https://libevent.org, 2019.

[24] Q Meng, A Humphrey, and M Berzins. The uintah framework: a unified heterogeneous task scheduling and runtime system. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 2441–2448. ieeexplore.ieee.org, November 2012.

[25] Jun Nakashima and Kenjiro Taura. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, volume 8665 of *LNCS*, pages 222–238. Springer, 2014.

[26] OpenMP Architecture Review Board. OpenMP application programming interface, version 5.0, November 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[27] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. *SIGPLAN Not.*, 45(6):376–387, June 2010.

[28] S Seo, A Amer, P Balaji, C Bordage, G Bosilca, A Brooks, P Carns, A Castelló, D Genet, T Herault, S Iwasaki, P Jindal, L V Kalé, S Krishnamoorthy, J Lifflander, H Lu, E Meneses, M Snir, Y Sun, K Taura, and P Beckman. Argobots: A lightweight Low-Level threading and tasking framework. *IEEE Trans. Parallel Distrib. Syst.*, 29(3):512–526, March 2018.

[29] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. Ucx: An open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, 2015.

[30] Dylan T Stark, Richard F Barrett, Ryan E Grant, Stephen L Olivier, Kevin T Pedretti, and Courtenay T Vaughan. Early experiences co-scheduling work and communication tasks for hybrid MPI+ X applications. In *2014 Workshop on Exascale MPI at Supercomputing Conference*, pages 9–19. ieeexplore.ieee.org, 2014.

[31] Steve Vinoski and Others. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Commun. Mag.*, 35(2):46–55, 1997.

[32] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPSW 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing Workshops*, pages 1–8. IEEE, 2008.