Toward Operating System Support for Scalable Multithreaded Message Passing

Balazs Gerofi
RIKEN Advanced Institute for Computational Science
bgerofi@riken.jp

Masamichi Takagi
RIKEN Advanced Institute for Computational Science
masamichi.takagi@riken.jp

Yutaka Ishikawa
RIKEN Advanced Institute for Computational Science
yutaka.ishikawa@riken.jp

ABSTRACT

Modern CPU architectures provide a large number of processing cores and application programmers are increasingly looking at hybrid programming models, where multiple threads of a single process interact with the MPI library simultaneously. Moreover, recent high-speed interconnection networks are being designed with capabilities targeting communication explicitly from multiple processor cores. As a result, scalability of the MPI library so that multithreaded applications can efficiently drive independent network communication has become a major concern.

In this work, we propose a novel operating system level concept called the *thread private shared library* (TPSL), which enables threads of a multithreaded application to see specific shared libraries in a private fashion. Contrary to address spaces in traditional operating systems, where threads of a single process refer to the exact same set of virtual to physical mappings, our technique relies on per-thread separate page tables. Mapping the MPI library in a thread private fashion results in per-thread MPI ranks eliminating resource contention in the MPI library without the need for redesigning it. To demonstrate the benefits of our mechanism, we provide preliminary evaluation for various aspects of multithreaded MPI processing through micro-benchmarks on two widely used MPI implementations, MPICH and MVAPICH, with only minor modifications to the libraries.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design; C.2.1 [**Computer-Communication Networks**]: Network Architecture and DesignNetwork Communications

Keywords

Hybrid kernels; Per thread page tables; MPI; MPI+OpenMP; Hybrid programming; Message Rate; Threads

1. INTRODUCTION

EuroMPI '15, September 21-23, 2015, Bordeaux , France

DOI: http://dx.doi.org/10.1145/2802658.2802661

As the rate of CPU clock improvement has stalled for the last decade, primarily due to energy consumption issues, increased use of parallelism in the form of multi- and manycore CPUs have been chased to improve overall performance. Manycore processors, which come with a large number of processing cores providing relatively lower clock rates but significantly higher power efficiency, are already widespread in high performance computing. While the number of cores keeps climbing steadily, the rate of growths in other node level resources, most notably RAM, has been lacking behind. Consequently, application developers are increasingly looking at hybrid programming models, that enable better resources consolidation. On the other hand, recent high-speed interconnection networks are being designed with capabilities targeting communication from multiple processor cores, explicitly requiring applications and the runtime system to drive communication simultaneously from multiple CPUs [8].

At present, one of the most prevalent hybrid approaches is to leverage MPI for inter-node communication and to rely on OpenMP for efficient exploitation of node level parallelism. Because this model indicates sharing MPI among multiple threads, scalability of the MPI library has become a major concern. Prior research has investigated efficient multithreaded message passing thoroughly from improving MPI performance in face of multiple threads, via better integration of MPI with other programming models (e.g., OpenMP, UPC) all the way to making multiple MPI ranks available in a single multithreaded process [7, 13, 19, 11, 28, 12, 26]. Most of these efforts, however, require extensive modifications not only to MPI internals, but often to the MPI API as well.

In this paper, we approach the MPI scalability problem from an operating system (OS) perspective. In traditional operating system kernels, threads that run in the same OS process, i.e., share a single address space, usually share the same set of hardware page tables[1]. Notice, however, that an OS process is a software concept while page tables are used for address translation by the hardware and there is nothing that prevents different hardware threads in a manycore system to use a separate set of page tables even if they execute threads that logically belong to the same process. Sharing the same address space means providing the same virtual to physical mappings for all participating threads, which can be ensured through careful synchronization even if page tables are decentralized. Furthermore, per-thread

[1]We explicitly target the x86 architecture in this paper.

page tables[2] enable mapping certain parts of the address space to different physical memory across CPUs. In particular, individual threads of a multithreaded process may see a shared library in a private fashion, as if the threads were different processes, which is the main focus of this study. We summarize our contributions as follows.

- We propose *thread private shared library* (TPSL), a novel operating system level concept that enables threads of a multithreaded application to see a specific shared library in a private fashion.
- Mapping the MPI library in a thread private fashion results in per-thread MPI ranks, which eliminates resource contention inside MPI and enables threads to perform lockless communication.
- As opposed to existing threaded MPI solutions, our proposal can be applied to any implementation of the message passing interface requiring only minor modifications to their codebase.

We describe the usage model of our approach in OpenMP and provide preliminary evaluation using various micro benchmarks on two MPI implementations, MPICH [4] and MVAPICH [5]. We demonstrate TPSL mapped MPI's ability to yield the same messaging performance in a multithreaded OpenMP environment to that of the flat MPI configuration. We also show that through parallel communication our technique can yield up to 10X speed-up for a stencil computation HALO exchange using derived datatypes when running over 16 CPU cores.

The rest of this paper is organized as follows. We begin with providing some background information on hybrid kernels and IHK/McKernel in Section 2. Section 3 discusses per thread page tables, required MPI modification and the usage model. Experimental evaluation is given in Section 4. Section 5 provides further discussion, Section 6 surveys related work, and finally, Section 7 presents future plans and concludes the paper.

2. BACKGROUND

Before diving into per-thread page tables and private instance libraries, we provide an overview of IHK/McKernel [25], our hybrid kernel configuration upon which the rest of this work is built.

Traditionally, operating systems in high-end computing followed two approaches to deliver scalable performance and the reliability needed for extreme scale. In the full weight kernel (FWK) approach [2, 31], a full Linux environment is taken as the basis and features that inhibit HPC scalability and performance are removed. The light-weight kernel (LWK) approach, on the other hand, [17, 16] starts from scratch and effort is undertaken to add sufficient functionality so that it provides an API close to that of a general purpose OS, while at the same time it retains the desired scalability and reliability. However, with the considerably increased complexity of recent HPC applications, e.g., demand for components such as in-situ analysis, elaborate monitoring and performance tools, etc., the full availability of Linux APIs has gained high importance. Unfortunately, neither of the above mentioned approaches yields a fully Linux compatible environment.

[2]We use the terms *per-thread, per-core, and per-CPU page tables* interchangeably in this paper.

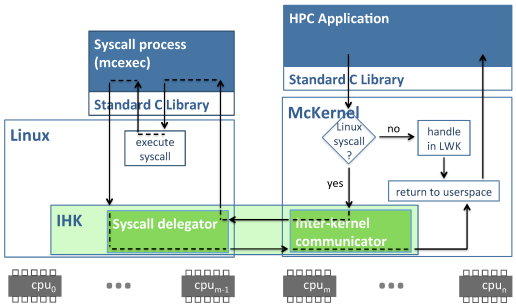


Figure 1: **Overview of the IHK/McKernel architecture and the system call delegation mechanism.**

As a result, multiple recent research projects are investigating a hybrid approach [25], [1], [30], [9], where Linux is run simultaneously with a light-weight kernel on the same node. IHK/McKernel is our approach to providing such a hybrid configuration. The IHK/McKernel architecture is shown in Figure 1. At the heart of the stack is a low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [25]. IHK is a general framework that provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. It also provides an Inter-Kernel Communication (IKC) layer, upon which system call delegation is implemented. McKernel is a lightweight kernel designed for HPC, which can be booted only from IHK.

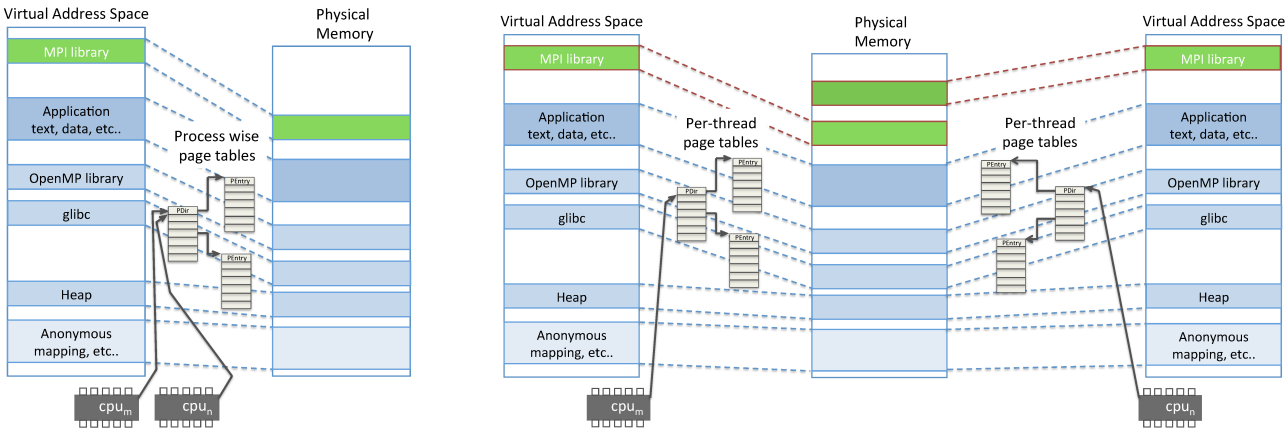
In case of IHK/McKernel, the application is run primarily on McKernel to achieve the needed scalability and reliability, but McKernel implements only performance sensitive system calls and the rest of the OS services are offloaded to Linux. With respect to this study, one of the most important attributes of McKernel is its drastically small code base, which enables rapid experimentation with OS features that would be otherwise highly intrusive and cumbersome to implement in Linux.

3. DESIGN AND IMPLEMENTATION

This section details the design of the proposed OS concept, called *thread private shared library*, it discusses the required modifications to MPI and demonstrates its usage model in an OpenMP application.

3.1 Thread Private Shared Libraries

As we pointed out earlier, in all major operating systems available currently (e.g., Linux, Windows, FreeBSD, etc.) multiple threads running in the same OS process, i.e., sharing a single address space, actually share the same set of page tables. Page tables are merely data structures that are referred by the memory management unit (MMU) to translate virtual to physical addresses. The MMU obtains the root of the page table tree via a special register in the CPU ($cr3$ on x86), which is a private resource for each hardware thread. Notice that an OS process is a software concept while page tables are used for address translation by the hardware and there is nothing that prevents different hardware threads in a many-core system to use a separate set



(a) Regular shared page tables

(b) Per-thread private page tables

Figure 2: **Address space layout of a multithreaded application using regular and per-core private page tables.** *(a.) All CPU cores see the same virtual to physical mappings, (b.) Most of the virtual address space (e.g., glibc and OpenMP) are mapped to the same physical memory, but MPI is mapped to different physical regions via thread private page tables.*

of page tables even if they conceptually belong to the same OS process. Sharing the same address space by definition means that each participating CPU follows the same set of translations from virtual to physical addresses, which can be easily guaranteed even if using distinct sets of page tables.

To this end, we have modified McKernel's memory management system to enable each thread in an OS process using its own set of page tables. Consequently, certain virtual addresses can be translated to different physical addresses on a per-thread basis. Furthermore, we have introduced a mapping table in the kernel that can be configured during boot time (the boot time of McKernel, not Linux) to indicate that certain files should be mapped in a thread private fashion even if the threads belong to the same process. Applying this mechanism to a specific shared library results in seeing a separate instance of the library by each thread as if they were different processes, hence the name thread private shared library (TPSL).

Our current prototype implementation separates library instances as follows. When the dynamic loader iterates library dependencies at application startup and request file mappings for each shared library the kernel consults the aforementioned table and maps them accordingly. To be precise, the actual separation happens after glibc initialization so that each thread acquires the library in a state where possible BSS modifications performed via glibc init hooks are reflected (see Section 3.2 for more details). In order to eliminate resource contention inside the MPI library, we instruct the kernel to map the entire library on a per-thread basis. Strictly speaking threads participating in such process do not share the entire virtual address space so one could argue that they don't belong to the same OS process any more. Indeed, TPSL blurs the concept of processes and threads and one could think of the same configuration as separate processes sharing most of their libraries. While shared mappings are commonplace, existing OS kernels unfortunately do not support transforming multiple processes into a single process so that each original process would mutate into a pthread sharing the same address space.

Figure 2 contrasts memory mappings between the traditional process wise shared page table configuration and the proposed per-thread based model. As seen in Figure 2b, the MPI library is mapped in a thread private fashion (emphasized by red lines), so from MPI's point of view these threads appear much like separate processes. On the other hand, glibc and OpenMP remain shared resulting in a global shared heap and the availability of OpenMP constructs as in any regular multithreaded process.

3.2 MPI Modifications

There is a minimal set of modifications required to make MPI aware of the proposed thread private configuration. First, the process management interface (PMI) needs to be able to handle these special processes slightly differently, because they constitute multiple MPI ranks in a single OS process. Using the HYDRA process manager, each application process obtains its MPI rank and a file descriptor for an established PMI connection through environment variables (PMI_RANK and PMI_FD, respectively) and by default the PMI proxy passes a single pair of values to each process. We have modified the HYDRA process manager so that it transfers a set of ranks and PMI fds to a process when TPSL is used. In turn, the PMI initialization function in the MPI library has been also extended to understand the new format and obtain its corresponding MPI rank and fd based on thread IDs. To let MPI know which thread ID it is dealing with, the MPI_Init routine expects an additional thread ID argument in our model. See Section 3.3 for an actual example how this is used in application code.

The second change is not strictly MPI related, but rather an Infiniband [3] specific modification. The low-level Infiniband API requires applications to explicitly register memory regions which are to be accessed via RDMA operations. The registration and deregistration operations are relatively expensive so Infiniband modules in MPI implementations often maintain a registration cache in order to minimize the number of calls to these functions. Unfortunately, to keep the cache consistent with the application's heap, MPI needs to track memory allocations and deallocations (i.e., the malloc() and free() family of routines). For this purpose, MPI

implementations usually provide their own heap manager by overriding the standard libc calls (for example, MVAPICH relies on a modified version of ptmalloc). By default, internal data structures of the heap manager are located in the BSS section of the MPI library. Because TPSL maps MPI in a thread private fashion, manipulating these data structures independently results in an inconsistent state. To overcome this problem, we simply moved them to the heap (leaving only pointers in BSS as reference), which is shared across all threads of the application even with per-core page tables. As we will see later in Section 4, we applied our mechanism to two MPI implementations, MPICH [4] and MVAPICH [5]. The above described BSS modifications amount to approximately 20 lines of code change in case of MVAPICH and even less within MPICH.

3.3 Usage Model

We will now demonstrate the usage model of MPI when mapped through TPSL in an OpenMP multithreaded application. The first difference compared to regular MPI is that in case of TPSL, each OpenMP thread needs to initialize its private instance of the MPI library via *MPI_Init()*. As we mentioned earlier, we extended *MPI_Init()* so that it accepts an additional thread ID argument.

```
#pragma omp parallel
{
  MPI_Init(&argc, &argv, omp_get_thread_num());
}
```

Listing 1: MPI Initialization using TPSL.

Listing 1 demonstrates the initialization call. As seen, each OpenMP thread passes its thread ID which it obtains from the OpenMP library. Note that there is no explicit requirement on how threads agree on their respective IDs, but the number should be in the interval of $[0, N)$, where N is the number of ranks in one TPSL process.

The second example shows the usage of derived data types in the context of a 3D stencil computation HALO area exchange [29].

```
  MPI_Datatype sub_xz;
  int sub_xz_size[3];
  int sub_xz_start[3];
  int xz_target;
  ...
#pragma omp parallel private(xz_target,
  sub_xz_size, sub_xz_start)
{
  /* Subarray type creation */
  sub_xz_size[0] = Z_SIZE / omp_get_num_threads();
  sub_xz_size[1] = 2;
  sub_xz_size[2] = X_SIZE;
  sub_xz_start[0] = omp_thread_id *
    (Z_SIZE / omp_get_num_threads());
  sub_xz_start[1] = 0;
  sub_xz_start[2] = 0;

  MPI_Type_create_subarray(3, sizes,
    sub_xz_size, sub_xz_start,
    MPI_ORDER_C, MPI_DOUBLE, &sub_xz);

  MPI_Type_commit(&sub_xz);

  xz_target = (rank + omp_get_num_threads())
    % num_ranks;

  /* Main loop */
  for (iter = 0; iter < NR_ITERS; ++iter) {
    /* Computation */
    ...
    /* HALO exchange */
    MPI_Isend(data, 1, sub_xz, xz_target, ...);
    ...
  }
}
```

Listing 2: HALO exchange with derived datatypes using TPSL.

Listing 2 indicates the code snippet. In particular, it shows how each OpenMP thread creates a datatype to represent its own piece from the X-Z plane. One can see that the Z dimension of the size and start arguments to the *MPI_Type_create_subarray()* function are computed according to the number of OMP threads (Z_SIZE / *omp_get_num_threads*()) and the thread ID (*omp_thread_id* $*$ (Z_SIZE / *omp_get_num_threads*())), respectively. The variable *xz_target* denotes the rank of the thread on the target node, which for the sake of simplicity is just an offset by the number of threads in one process, assuming that in the original code each rank would have sent the entire X-Z plane simply to its subsequent rank in the MPI job.

The most important observation here is that the data distribution among OpenMP threads results in implicit parallelization of the underlying datatype packing and unpacking routines as well as of the data transfer. In Section 4.3 we will provide measurements on how such parallelization impacts performance.

4. EVALUATION

This section provides preliminary evaluation of the proposed mechanism using various message passing micro-benchmarks.

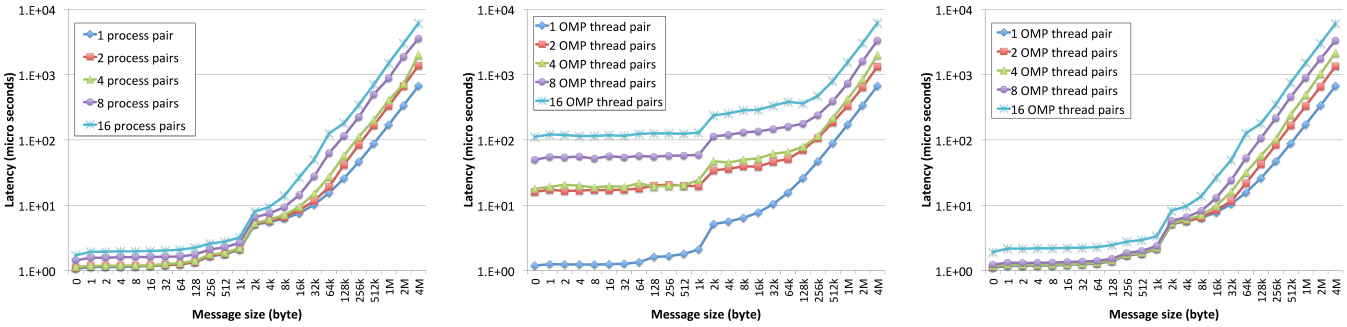
4.1 Experimental Setup

Our experiments were conducted on a small cluster where each node consists of Intel Xeon Ivy Bridge (E5-2670 v2 @ 2.50GHz) CPUs with two sockets, ten cores per socket and two hardware threads per core. The nodes are also equipped with 64GB RAM organized in two NUMA domains and with Mellanox Infiniband QDR (MT27500 ConnectX-3) interconnection network. In all experiments where Linux' performance is compared to IHK/McKernel it is always ensured that the Linux run is scheduled to the exact same set of CPU cores and to the same NUMA domain with the McKernel partition when our stack is used. To eliminate NUMA effects we restricted all evaluation to NUMA node 0.

To demonstrate the transparency of our proposal, most measurements were performed on two different MPI implementations, MPICH version 3.1.3 [4] and MVAPICH version 2.1 [5]. It is worth pointing out that MVAPICH itself was derived from MPICH, however, it has gone through significant changes over the course of time.

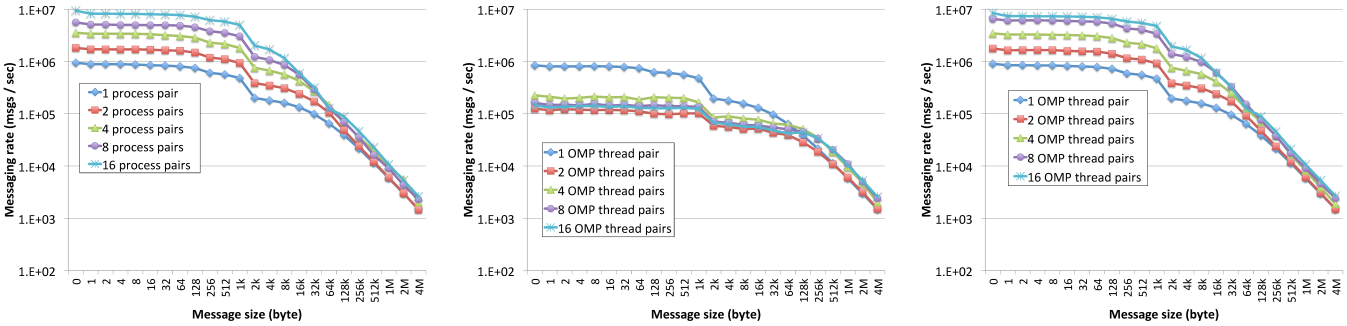
4.2 Latency, Message Rate and Bandwidth

In the first set of experiments we are focusing on latency, message rate and bandwidth between communicating pairs of processes or threads. We compare three configurations. First, communicating process pairs between two nodes where each MPI process is bound to a separate CPU core, denoted by flatMPI. Second, communicating thread pairs using multithreaded MPI and OpenMP hybrid programming, indicated by multithreaded MPI. These two experiments were run on top of Linux. Finally, we evaluate communicating



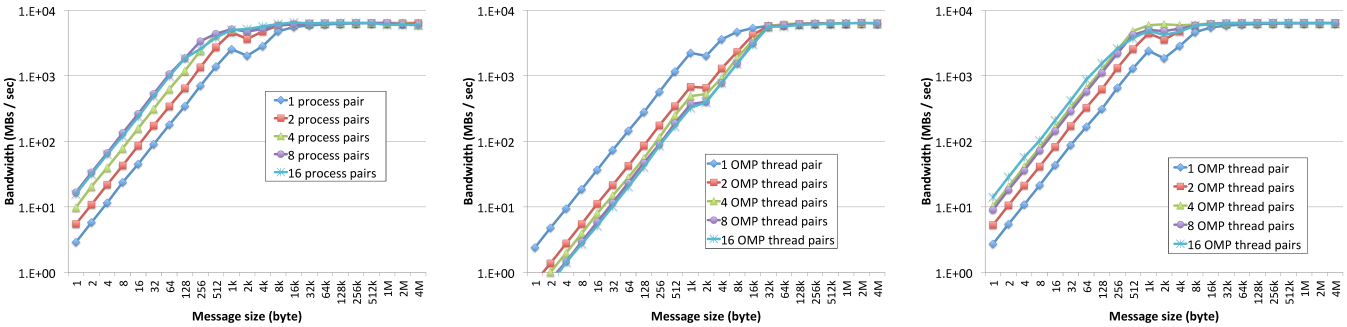
(a) Linux+MVAPICH flat MPI (b) Linux+MVAPICH multithreaded MPI (c) IHK/McK+MVAPICH TPSL MPI

Figure 3: **MVAPICH Latency.**



(a) Linux+MVAPICH flat MPI (b) Linux+MVAPICH multithreaded MPI (c) IHK/McK+MVAPICH TPSL MPI

Figure 4: **MVAPICH Message Rate.**



(a) Linux+MVAPICH flat MPI (b) Linux+MVAPICH multithreaded MPI (c) IHK/McK+MVAPICH TPSL MPI

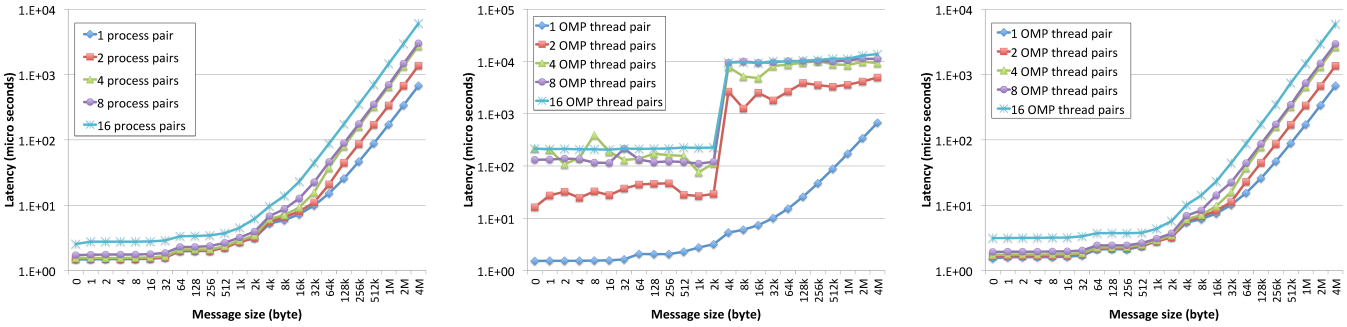
Figure 5: **MVAPICH Bandwidth.**

thread pairs using TPSL mapped MPI running on top of IHK/McKernel also driven from OpenMP threads.

For all benchmarks we took the OSU benchmark set as reference [6]. We modified the latency and bandwidth benchmarks to suit our needs. The communicating process pairs is a straightforward extension of the original benchmarks. As for the multithreaded version, it is worth noting that we used dedicated tags for each thread pair in the MPI send/receive routines. TPSL requires similar usage model to threads (as we showed in Section 3.3), but tags are not necessary because the communicating thread pairs have their respective MPI ranks.

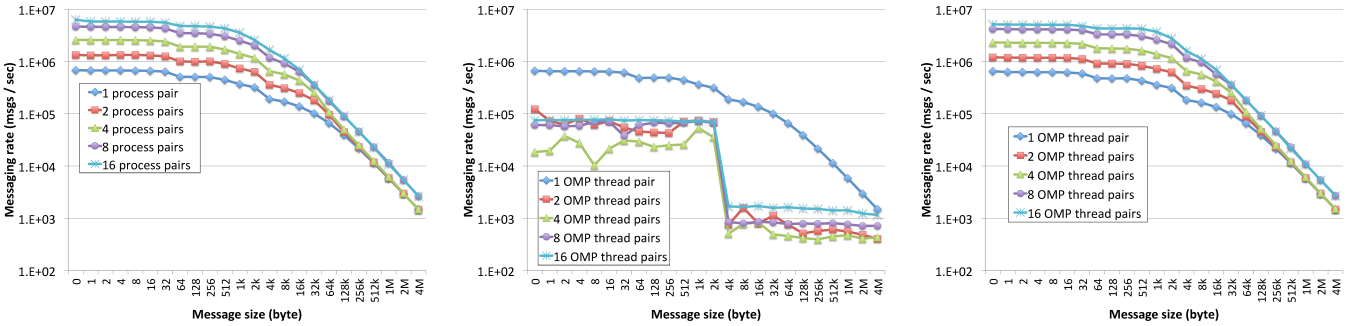
Our primary intention is to verify that TPSL mapped MPI performs on par with flat MPI (i.e., separate MPI processes on each CPU core) even in the context of hybrid MPI+OpenMP model, where on the other hand multithreaded MPI experiences heavy resource contention due to competing threads.

Figure 3, Figure 4 and Figure 5 indicate results measured on MVAPICH for latency, message rate and bandwidth, respectively. The X axis denotes message size, while Y axis shows microseconds in case of latency, number of messages



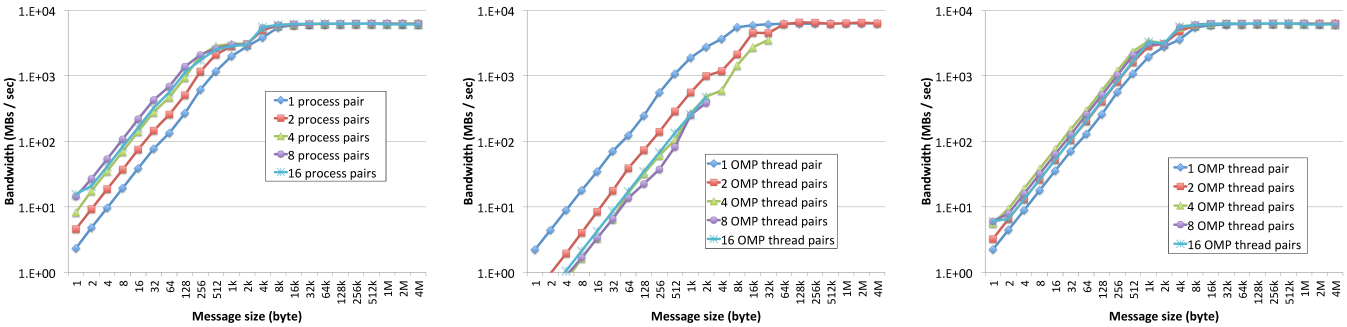
(a) Linux+MPICH flat MPI (b) Linux+MPICH multithreaded MPI (c) IHK/McK+MPICH TPSL MPI

Figure 6: **MPICH Latency.**



(a) Linux+MPICH flat MPI (b) Linux+MPICH multithreaded MPI (c) IHK/McK+MPICH TPSL MPI

Figure 7: **MPICH Messaging Rate.**



(a) Linux+MPICH flat MPI (b) Linux+MPICH multithreaded MPI (c) IHK/McK+MPICH TPSL MPI

Figure 8: **MPICH Bandwidth.**

per second for message rate and MBs/sec for bandwidth. Each figure provides results from 1 to 16 communicating pairs of processes/threads.

For easy reference, the Y axis in all figures is adjusted so that results for one communicating pair of processes/threads are displayed at the same position. As indicated, while 16 process pairs using flat MPI only experience approximately 70% latency increase compared to 1 process pair, letting multiple threads interact with the MPI library results in over two orders of magnitude increase in latency. In contrast, the private nature of TPSL mapped MPI eliminates any resource/lock contention inside MPI and consequently enables the (almost) identical multi-threaded benchmark to yield the same performance as separate communicating processes.

Message rate is even more expressive than latency because it truly demonstrates the performance characteristics of today's network interfaces in response to multiple CPU cores. As Figure 4 indicates, in case of flat MPI message rate increases almost linearly with the number of communicating process pairs yielding almost an order of magnitude higher number of exchanged messages for 16 pairs compared to

one. In contrast, due to resource contention in the MPI library, the performance of multithreaded message rate actually decreases with the number of threads. Again, TPSL mapped MPI exhibits the same behavior as regular process pairs, even though communication is driven from the same OpenMP multithreaded environment.

Moving on to bandwidth, Figure 5 depicts the measurements for all three scenarios. As seen, for large messages the attained bandwidth is rather independent from the MPI mechanism used, yielding basically the same value across all measurements. On the other hand, small messages are more interesting and give similar results to message rate. With respect to the number of participating cores, an almost linear bandwidth increase can be observed in case of flat MPI. Contrary, the multithreaded MPI configuration suffers heavily from the increasing number of threads. Nevertheless, similarly to message rate, TPSL mapped MPI delivers the same bandwidth benefits as flat MPI, regardless the multithreaded nature of the benchmark.

We conducted the same set of measurements for MPICH as well and Figure 6, Figure 7 and Figure 8 indicate the results. In general, the same conclusions can be drawn as for MVAPICH, however, it's worth pointing out that we used a relatively recent, not yet optimized Infiniband module in this experiments. We believe this is the main reason why multithreaded measurements across the board perform rather irregularly. Setting this aspect of the data aside, TPSL clearly demonstrates its agnostic nature of the underlying MPI implementation.

4.3 HALO Exchange using Derived Datatypes

In order to demonstrate the benefits of a TPSL mapped MPI library, we have developed a set of micro-benchmarks that mimic HALO exchange in three dimensional stencil computation [29]. The computation data is a 3D grid of double precision floating points, and we provide 4 arrangements varying the size of X, Y, and Z dimensions as follows. *Cube* denotes a grid with 512 elements in each dimension, *Large X*, *Large Y*, and *Large Z* stand for 16K elements in the large dimension and the remaining dimensions (in alphabetical order) are set to 128 and 64, respectively. The array is layed out in row major order, implying that the X-Y plane is contiguous in memory.

We used the subarray derived data type for exchanging the X-Z and Y-Z planes. When employing TPSL MPI ranks, OpenMP threads divide the surface among each other and send their respective slice of the data. A code snippet demonstrating this arrangement has been shown previously in Section 3.3. Note that since the X-Y plane is contiguous in memory, neither using derived datatypes nor parallelizing the data transfer yield any benefits.

On the other hand, Figure 9 and Figure 10 indicate the speedup gained according to the number of threads involved in the data transfer for exchanging plane X-Z and Y-Z, respectively. As seen, the X-Z HALO exchange performs best when applied to the large Z grid arrangement, for which 16 threads yield almost 10X speedup compared to using only one thread. Because the large Z dimension implies a lot of small data chunks scattered non-contiguously in memory, parallelizing the derived datatype packing operation yields substantial improvements.

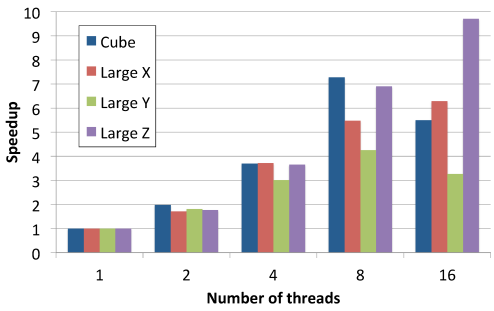


Figure 9: **HALO exchange of the X-Z plane with TPSL ranks.**

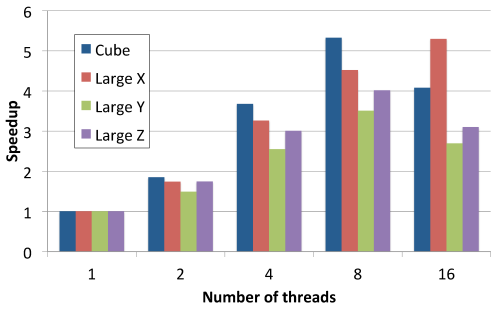


Figure 10: **HALO exchange of the Y-Z plane with TPSL ranks.**

As shown in Figure 10, parallelizing the Y-Z plane exchange proved to be most beneficial when using large X dimension, in which case 16 threads improve performance by over 5X. To the contrary, with less suitable grid shapes increasing the number of threads too far occasionally implies a slowdown compared to fewer threads. Nevertheless, we observe improvements across all measurements when utilizing up to 8 threads for parallelizing HALO exchange.

5. DISCUSSION

We cover a couple of additional topics related to thread private shared libraries, focusing primarily on their restrictions and weaknesses related to helper threads, memory consumption and TLB contention.

5.1 Helper Threads

Throughout our experiments we deployed the MPI library without utilizing helper threads. This isn't necessary a restriction by itself, because MPICH for instance does not use helper threads by default. When it comes to TPSL, however, the main problem with helper threads is that the operating system has no knowledge of whether a particular thread should see the same mappings of a per-thread library as another existing thread or have its own page tables set up. We foresee multiple solutions to this problem. One would be to introduce a dedicated flag to the clone() system call that would indicate whether or not the thread should see the same set of mappings with its parent. Another solution would be to allow threads switching between mappings, which could also enable a thread to access arbitrary MPI ranks in the given process.

5.2 Memory Consumption

While the TPSL approach eliminates lock contention in multi-threaded environments enabling threads to perform lockless MPI communication, it also consumes extra resources. First, because it maps libraries in a private fashion it consumes additional memory for page tables. However, the additional amount of memory required is not as severe as one might imagine due to aggressive sharing of page tables whenever it is possible. The page table sharing mechanism works as follows. Although each thread has its own page global directory (PGD), TPSL separates page tables only for a predefined region of the virtual address space. For mappings that are identical process-wise, the page table tree is entirely shared from the page middle directory (PMD) all the way down to the actual page table entries. It is also worth pointing out that physical pages backing the MPI library are managed through copy-on-write, making extra copies only when thread private modifications make it necessary.

Second, TPSL utilizes thread private buffers for unexpected messages and other MPI internals the same way as flat MPI in case of multiple processes. In an optimized MPI implementation such resources can be shared among threads and thus memory consumption can be decreased. Again, the MPI implementation could be made aware of TPSL mappings and make threads share buffers the same way as in a multithreaded implementation, however this would clearly require additional modifications.

5.3 TLB Implications

Modern CPU architectures employ Translation Lookaside Buffers (TLBs) to cache recent translations from virtual to physical memory, and thus to speed up lookup operations. While TLB entries are typically private to hardware threads, some modern CPUs enhance TLB performance by introducing additional levels of the TLB cache that can be shared among hardware threads or CPU cores. Mapping MPI in a thread private fashion makes threads utilized separate mappings from the hardware's point of view, which in turn can increase TLB resource contention in higher level caches.

6. RELATED WORK

This section discusses related work in the domains of HPC operating systems (particularly focusing on lightweight and hybrid kernels) and multithreaded message passing.

6.1 Operating Systems for HPC

Lightweight kernels developed from scratch and designed explicitly for HPC workloads date back for over two decades now. Notably, Catamount [17] from Sandia National Laboratories was one of the first systems which has been successfully deployed on a large scale system. The IBM BlueGene line of supercomputers have also been running an HPC targeted lightweight kernel called CNK [16]. On the other end of the lightweight kernel spectrum are kernels which originate from Linux, but have been heavily modified to meet HPC requirements ensuring low noise, scalability and predictable application performance. Cray's Extreme Scale Linux [2] and ZeptoOS [31] follow this approach. Neither of these approaches, however, succeed in retaining full Linux compatibility and achieving high scalability at the same time, which has become highly desired due to the increasing complexity of large scale systems and applications.

To mitigate this issue, FusedOS [22] was the first to propose combining Linux with an LWK and since then, along with IHK/McKernel [25], a whole new breed of OS research (ANL's Argo [1], Intel's mOS [30], and Sandia National Laboratories' Hobbes [9]) have been investigating how to provide LWK performance while retaining the Linux APIs. One of the main arguments in favor of lightweight kernels in these hybrid configurations is its ability to nimbly be adapted to new technologies. Page table and virtual memory management in general is a complex subsystem of the Linux kernel and per-thread page tables would be a major development effort in Linux. In contrast, McKernel is simple enough to incorporate changes for TPSL in a reasonable amount of time, which we believe further justifies the need for lightweight kernels in high-end computing.

Nevertheless, per-thread (or per-CPU) page tables have been studied before in the context of scalable virtual address spaces [10], as well as in our previous work for hierarchical memory management in heterogeneous architectures [14, 15]. TPSL also relies on per-core page tables, however, it leverages them to implement a novel OS concept, the thread private library.

6.2 MPI and Threads

Prior research has considered efficient multithreaded message passing extensively from improving MPI performance in face of multiple threads, through better integration of MPI with other programming models (e.g., OpenMP, UPC) all the way to making multiple MPI ranks available in a single multithreaded process.

Balaji et al. investigated building a fully thread-safe MPI implementation with decreasing levels of critical-section granularity comparing coarse-grain locks, fine-grain locks and lock-free operations [7]. Dozsa and colleagues used a combination of a multichannel-enabled network interface, fine-grained locks, lock-free atomic operations, and specially designed queues to provide a high degree of concurrent access on BlueGene/P systems [13]. While these studies focus on improving the performance of a shared MPI rank across multiple threads, we eliminate resource contention at the OS level by exposing separate instances of the MPI library.

Luo et al. proposed opening multiple network endpoints within the MPI library and driving communication in parallel relying on lock-free algorithms while retaining the single MPI rank interface to the application [18, 19]. Although their solution can be user transparent, contrary to TPSL it requires extensive modifications to the MPI library. Moreover, this approach also inhibits performance degradation when used by multiple threads due to the requirement for keeping MPI's FIFO message matching order which enforces implicit serialization among threads using a shared rank.

With respect to the MPI+OpenMP hybrid programming model, multiple previous studies have concluded that applications indeed can benefit from hybridization [20, 24]. In response, Min et al. demonstrated how MPI can leverage idle OpenMP threads that otherwise would not be involved in communication [26]. They proposed various optimizations in different parts of MPI implementation, including derived datatype processing, shared-memory communication, etc. Generally, these improvements are effective at involving multiple CPU cores in an application's communication without the need for any modifications to the application, but at the same time they shift the burden of extracting and

managing communication parallelism entirely to the MPI library.

MPI implementations, where MPI processes are implemented as threads have been also considered previously [11, 28, 23, 21], however, these proposals require extensive modifications, if not a complete rewrite of the MPI library. From a usage point of view, the most similar study to our work is the recent MPI Endpoints proposal [12]. The Endpoints proposal calls for an extensions to the MPI standard that would allow implementations to minimize contention and improve performance by explicitly enabling multiple MPI ranks in a multithreaded process. Again, the most important difference here is the need for an extensive redesign of the MPI internals. Although Dinan et al. recently demonstrated a library based approach to MPI endpoints [27], where ghost processes running an unmodified MPI stack can transparently serve as MPI endpoints to the actual application, their solution requires notifications via crossing process boundaries, which decreases performance for small messages. On the contrary, TPSL blurs the notion of threads and processes at the OS level and consequently enables separate MPI ranks to be seen from different threads without crossing address spaces.

7. CONCLUSION AND FUTURE WORK

The relatively higher rate of increase in the number of CPU cores compared to other resources (e.g., memory) promotes hybrid MPI+X programming models, where multiple threads sharing the MPI library are utilized in a single address space to exploit node-level parallelism. On the other hand, to attain high-performance communication on modern NICs, multiple CPU cores are expected to drive the network, which calls for multiple communication endpoints. To bridge this gap without the need for redesigning the message passing library, we have proposed a novel OS level concept, which we call thread private shared library (TPSL).

Mapping the MPI library in a thread private fashion results in per-thread MPI ranks, which eliminates resource contention inside MPI and enables threads to perform lockless communication. At the same time, it seamlessly retains the multithreaded programming model that application developers are acquainted with. We have shown that TPSL mapped MPI can yield similar messaging performance in an OpenMP environment to that of the flat MPI configuration and demonstrated how our mechanism can yield up to 10X speed-up when using derived datatypes for HALO exchange on 16 CPU cores. As we have shown for two MPI implementations (MPICH and MVAPICH), our technique requires only a minimal set of changes to the libraries.

In the near future we intend to further evaluate our proposal using real applications running on scale, and possibly applying it to other MPI implementations as well. We are also excited to explore whether or not TPSL could be employed to tackle multithreaded scalability in other programming models, such as PGAS.

Taking into account the amount of changes needed for implementing per-thread page tables in McKernel, which would be a major development effort in Linux, we believe that TPSL very well demonstrates a lightweight kernel's ability to rapidly adapt to new software/hardware requirements.

Acknowledgment

This work is partially funded by MEXT's program for the Development and Improvement for the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities.

Part of this work has been performed while visiting the mOS group at Intel Labs. We are grateful for all the valuable discussions with Rolf Riesen, Evan Powers and David Van Dresser.

We also acknowledge the McKernel development efforts of Tomoki Shirasawa and Gou Nakamura from Hitachi.

8. REFERENCES

[1] Argo: An Exascale Operating System (Accessed: Jan, 2015). http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[2] Cray Linux Environment (CLE) 4.0 Software Release Overview (Accessed: Jan, 2015). http://docs.cray.com/books/S-2425-40/S-2425-40.pdf.

[3] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.

[4] MPICH: High-Performance Portable MPI (Accessed: May, 2015). https://www.mpich.org/.

[5] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE (Accessed: May, 2015). http://mvapich.cse.ohio-state.edu/.

[6] Network-Based Computing Laboratory, The Ohio State University: OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks.

[7] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., and Thakur, R. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds., vol. 5205 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 120–129.

[8] Barrett, B. W., Hammond, S. D., Brightwell, R., and Hemmert, K. S. The Impact of Hybrid-core Processors on MPI Message Rate. In *Proceedings of the 20th European MPI Users' Group Meeting* (New York, NY, USA, 2013), EuroMPI '13, ACM, pp. 67–71.

[9] Brightwell, R., Oldfield, R., Maccabe, A. B., and Bernholdt, D. E. Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2013), ROSS '13, ACM, pp. 2:1–2:8.

[10] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 211–224.

[11] Demaine, E. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *In: Proceedings of the 11th International Symposium on High Performance Computing Systems* (1997), pp. 153–163.

[12] Dinan, J., Balaji, P., Goodell, D., Miller, D., Snir, M., and Thakur, R. Enabling mpi

interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting* (New York, NY, USA, 2013), EuroMPI '13, ACM, pp. 13–18.

[13] Dózsa, G., Kumar, S., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Ratterman, J., and Thakur, R. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface* (Berlin, Heidelberg, 2010), EuroMPI'10, Springer-Verlag, pp. 11–20.

[14] Gerofi, B., Shimada, A., Hori, A., and Ishikawa, Y. Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (May 2013), pp. 360–368.

[15] Gerofi, B., Shimada, A., Hori, A., Masamichi, T., and Ishikawa, Y. CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 73–84.

[16] Giampapa, M., Gooding, T., Inglett, T., and Wisniewski, R. W. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–10.

[17] Kelly, S. M., and Brightwell, R. Software architecture of the light weight kernel, Catamount. In *In Cray User Group* (2005), pp. 16–19.

[18] Luo, M., Jose, J., Sur, S., and Panda, D. Multi-threaded UPC runtime with network endpoints: Design alternatives and evaluation on multi-core architectures. In *High Performance Computing (HiPC), 2011 18th International Conference on* (Dec 2011), pp. 1–10.

[19] Luo, M., Lu, X., Hamidouche, K., Kandalla, K., and Panda, D. K. Initial Study of Multi-endpoint Runtime for MPI+OpenMP Hybrid Programming Model on Multi-core Systems. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2014), PPoPP '14, ACM, pp. 395–396.

[20] Lusk, E., and Chan, A. Early experiments with the openmp/mpi hybrid programming model. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism* (Berlin, Heidelberg, 2008), IWOMP'08, Springer-Verlag, pp. 36–47.

[21] Negara, S., Zheng, G., Pan, K.-C., Negara, N., Johnson, R. E., Kalé, L. V., and Ricker, P. M. Automatic MPI to AMPI Program Transformation Using Photran. In *Proceedings of the 2010 Conference on Parallel Processing* (Berlin, Heidelberg, 2011), Euro-Par 2010, Springer-Verlag, pp. 531–539.

[22] Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D., and Wisniewski, R. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on* (Oct 2012), pp. 211–218.

[23] PÃL'rache, M., Carribault, P., and Jourdren, H. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 5759 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 94–103.

[24] Rabenseifner, R., Hager, G., and Jost, G. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on* (Feb 2009), pp. 427–436.

[25] Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A., and Ishikawa, Y. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures. In *High Performance Computing (HiPC), 2014 21st Int. Conf. on High Performance Computing* (Dec 2014), HiPC '14, pp. 1–10.

[26] Si, M., Peña, A. J., Balaji, P., Takagi, M., and Ishikawa, Y. MT-MPI: Multithreaded MPI for Many-core Environments. In *Proceedings of the 28th ACM International Conference on Supercomputing* (New York, NY, USA, 2014), ICS '14, ACM, pp. 125–134.

[27] Sridharan, S., Dinan, J., and Kalamkar, D. D. Enabling Efficient Multithreaded MPI Communication Through a Library-based Implementation of MPI Endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2014), SC '14, IEEE Press, pp. 487–498.

[28] Tang, H., and Yang, T. Optimizing Threaded MPI Execution on SMP Clusters. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM, pp. 381–392.

[29] Wallcraft, A. J., and Moore, D. R. The NRL Layered Ocean Model. In *Parallel Computing* (Amsterdam, The Netherlands, The Netherlands, 1997), vol. 23, Elsevier Science Publishers B. V., pp. 2227–2242.

[30] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An Architecture for Extreme-scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2014), ROSS '14, ACM, pp. 2:1–2:8.

[31] Yoshii, K., Iskra, K., Naik, H., Beckmanm, P., and Broekema, P. C. Characterizing the Performance of Big Memory on Blue Gene Linux. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops* (2009), ICPPW '09, IEEE Computer Society, pp. 65–72.