KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework

Brice Goglin, Stéphanie Moreaud

KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework

Brice Goglin and Stéphanie Moreaud

INRIA, Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
Corresponding author email: Brice.Goglin@inria.fr – Phone: +33 5 24 57 40 91 – Fax: +33 5 24 57 40 41

Abstract

The multiplication of cores in today's architectures raises the importance of intra-node communication in modern clusters and their impact on the overall parallel application performance. Although several proposals focused on this issue in the past, there is still a need for a portable and hardware-independent solution that addresses the requirements of both point-to-point and collective MPI operations inside shared-memory computing nodes.

This paper presents the KNEM module for the Linux kernel that provides MPI implementations with a flexible and scalable interface for performing kernel-assisted single-copy data transfers between local processes. It enables high-performance communication within most existing MPI implementations and brings significant application performance improvements thanks to more efficient point-to-point and collective operations.

Key words: MPI, Intra-node communication, Collective operations, Kernel assistance, Portability, Hardware-independence, KNEM

1. Introduction

The high performance computing landscape has been largely revised because of technology revolutions in the last decades. High-speed networks such as Myrinet or InfiniBand led to the widespread use of clusters; up to 85% of the last revision of the machines on the Top500 ranking of supercomputing sites [1] utilize these types of technology. This trend came together with the dominant use of the *Message Passing Interface* [2] which is now the *de facto* standard for communication between processes within parallel applications.

The emergence of multicore processors several years ago caused the degree of parallelism to increase inside nodes. Although some other programming models tackle these large shared-memory nodes, MPI remains widely used on multicore clusters because existing MPI applications can immediately benefit from their increased computing power without requiring a rewrite. Therefore intra-node communication is now one of the important features that MPI implementations have to offer.

Many research efforts focused on optimizing MPI intra-node communication. Most MPI implementations now use a shared-memory-based double-copy strategy that offers very low latency for small messages. Several hardware-dependent implementations also focus on large messages. We present in this article the KNEM solution (*Kernel Nemesis*) that is now available in most existing MPI implementations. KNEM is a hardware-independent kernel module that offers direct copy between processes on any existing Linux

system. It was initially designed for point-to-point send-receive operations and was later generalized to collective and remote memory access MPI operations.

Multiple MPI performance improvements were published with MPICH2 over KNEM [3, 4] and Open MPI [5, 6]. Given these successful results, this article steps back and focuses on KNEM design and implementation to explain why and how it suits MPI requirements for point-to-point and collective operations.

The remainder of this article is organized as follows. Section 2 describes the role of intra-node communication in modern parallel computing and the existing designs. Section 3 introduces our approach and design goals towards an efficient and portable intra-node communication framework, whose implementation is then detailed in Section 4. Section 5 presents a performance evaluation of our proposal from micro-benchmarks up to applications. Related works are compared in Section 6 before the conclusion and future works are presented.

2. Intra-node MPI Communication

The 1.0 revision of the *Message Passing Interface* (MPI) was introduced in 1994 [2] and became the *de facto* standard for programming parallel applications on distributed systems. Although many other programming models or languages, such as OpenMP [7], have been proposed, MPI is still widely used for *intra-node* communication inside shared-memory machines. Indeed, any existing MPI application written for distributed systems can immediately work on shared-memory nodes without any modification. On the contrary, rewriting it in a different parallel programming paradigm may require a lot of work. Moreover, mixing different paradigms for distributed and shared-memory cases makes programming more difficult and still requires significant work in the implementations before it achieves high performance [8]. In this section, we present the importance of intra-node MPI communication and the available implementations.

2.1. On the Importance of Intra-node Communication

Intra-node communication has been involved in parallel computing long before the rise of multicore processors and many-cores nodes. Dual-processor servers were already considered to have one of the best performance ratios ten years ago. For instance, Thunderbird[1], one of the last very large clusters based on single-core processors, reached the fifth rank of the Top500 [1] in 2005 while using two processors per node. In the last revision of the Top500 (June 2012), the most powerful cluster is the SuperMUC system in Leibniz[2] which notably contains 9,216 dual-processor nodes.

Since the advent of multicore processors, more than 75% of the Top500 are now clusters of dual-processor nodes with at least 4 cores per processor, making intra-node communication even more critical to the overall performance. Considering a cluster of N nodes with P processor cores, the theoretical share of local inter-process communication is $(P-1)/(NP-1)$. This ratio was $1/(2N-1)$ for dual-processor single-core machines, but it now converges towards $1/N$ thanks to P increasing beyond 8 or 12 in modern servers.

However, most applications communicate more with their local neighbors. Due to architectural changes, application placement is critical to performance. Users are advised to map MPI processes according to their affinities so as to benefit from cache-sharing, intra-node communication or reduced network distance as much as possible. This placement can be performed manually using the developer knowledge of the

[1] `http://www.cs.sandia.gov/platforms/Thunderbird.html`

[2] `http://www.lrz.de/services/compute/supermuc/systemdescription/`

application or using measured affinities between processes [9]. Affinity-based reordering of MPI ranks is another recent solution that lets the MPI implementation switch process roles to improve locality based on the communication pattern [10]. The actual share of local communication in most modern parallel applications on multicore clusters is therefore much higher than it may seem. Intra-node communication performance is therefore critical to the overall application performance.

2.2. Hardware-specific Optimization

Optimizing local communication was already the target of research projects thirty years ago. IP addresses starting with 127 have been reserved for loop back communication since then, and many applications used sockets for local inter-process exchanges. The loopback network interface is currently a software bypass of the network in the IP layer. Early clusters relied on this feature as an easy way to implement intra-node communication.

The advent of specialized high-performance networks such as MYRINET [11] came with the need for similar optimization. Such powerful network interface cards (NIC) have the ability to transfer data from/to the host at very high bandwidth (through DMA). An easy way to optimize local communication is therefore to add a hook in the NIC so that messages to local processes are immediately transferred back to the host memory. This strategy (depicted as *Hardware Loopback* on Figure 1) is actually still available on today's platforms and offers very good performance [12]. However it implies two DMA transfers across the memory and I/O buses, causing contention that may affect the performance of inter-node communication and of the overall application.

A pure software loopback model looks more attractive because it bypasses the NIC and I/O bus entirely. High performance networks come with dedicated drivers in the operating system. Network vendors therefore added software loopback support to these drivers [13] (see *HW Driver Assistance* on Figure 1). However, this intra-node communication support is still hardware-dependent and cannot be used on different platforms without such high-speed network technologies.

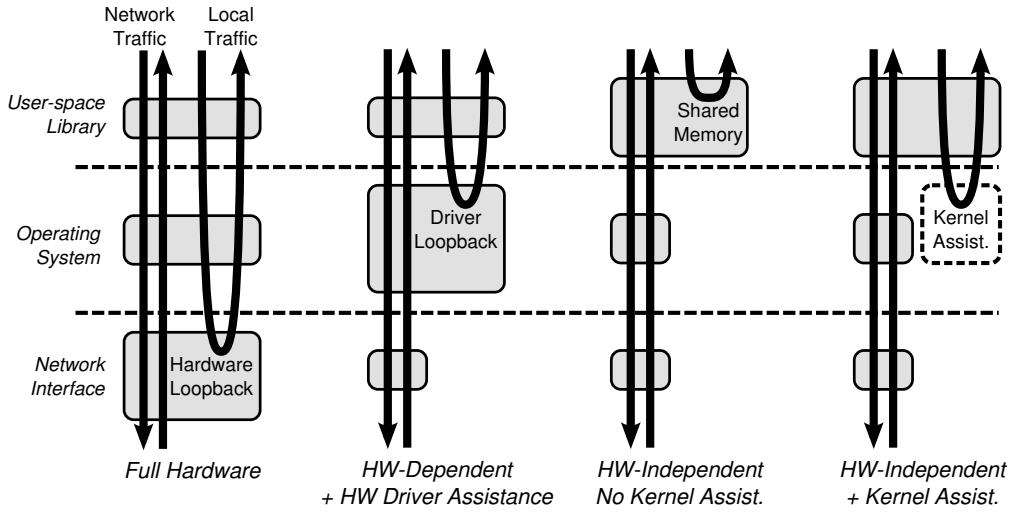


Figure 1: Comparison of possible intra-node communication supports, with or without hardware independence, and with or without kernel assistance.

All MPI stacks must offer intra-node communication support, either in hardware or in software. If a high-speed network is involved, the MPI implementation can directly benefit from its custom loopback.

This leads to surprising situations such as Open-MX[3] implementing its own loopback support [14] because it must be compatible with MPICH-MX which assumes the MX [15] loopback support is available.

2.3. The Need for a Portable Solution

Figure 1 summarizes the existing solutions for offering intra-node communication support in hardware-dependent or independent manners. Hardware-dependency is convenient because it benefits from the specialized NIC and operating system drivers to implement a dedicated loopback. However, these solutions are not portable to other platforms. MPI implementations still have to offer intra-node communication support if no specialized network is available.

Operating systems offer several inter-process communication interfaces such as sockets (through the loopback interface) or pipes on Unix. However they are not optimized for message passing and require system calls, which dramatically increase the overhead for small messages.[4] Modern MPI implementations usually use another kind of inter-process communication: shared memory (see *No Kernel Assistance* on Figure 1). This portable solution consists of having the sender write the message in a shared buffer before the receiver reads from it. Atomic operations and pipelining enables highly optimized implementations that can reach very low latency [16]. However, using this strategy for large messages causes cache pollution, higher memory bandwidth waste, and CPU overhead due to two memory copies being involved [17]. This is the problem that is addresses by the work proposed in this paper. Better performance for these large messages may be achieved thanks to a hardware-dependent NIC or driver at the expense of portability as explained in 2.2.

Several operating system extensions have been carried out in the last several years to overcome intra-node communication issues (see Section 6). Specialized hardware such as Cray or BlueGene machines may benefit from custom hardware or software innovations. Although they do not depend on the actual underlying network technology, they are still far from being portable to commodity clusters.

The aim of this paper is to propose a generic solution that overcomes intra-node communication problems without depending on any specific hardware, as depicted by *Kernel Assistance* on Figure 1. Moreover, we focus on Linux clusters because Linux is used in almost 90% of the Top500 and in the vast majority of clusters. We introduce in this article the KNEM kernel module[5] which has been designed to tackle all these needs and features. It is already supported by all popular MPI implementations.[6] KNEM offers more flexibility than its competitors by supporting asynchronous copies, non-contiguous buffers and offloading to DMA hardware. We describe in the next sections the key design and implementation points that led to its wide adoption.

3. Design of an Efficient Intra-node MPI Communication Framework

The KNEM Linux kernel module was initially designed to address point-to-point communication inside shared-memory nodes before being extended to help collective communication as well. This work does not focus on small messages because it cannot compete with the latency of the existing shared-memory double-copy strategies. Rather, KNEM targets large messages throughput by reducing the number of copies.

[3] The Open-MX software message-passing stack implements the *Myrinet Express* interface on generic Ethernet hardware.

[4] A system call still costs about 100 nanoseconds on today's platform while optimized intra-node communication latency can be as fast as 200 ns.

[5] Available for download at `http://runtime.bordeaux.inria.fr/knem`

[6] KNEM is supported by the default MPICH2 distribution since version 1.1.1 and Open MPI since 1.5. Many derivative implementations such as MVAPICH2 (based on MPICH2) or Bullx MPI (based on Open MPI) immediately benefit from this support.

We describe in this section how the KNEM interface was designed before detailing its implementation in Section 4.

3.1. Send-Receive Interface

Offering kernel assistance to MPI implementations for intra-node communication raises several questions. One important thing to keep in mind when designing new operating system features is that the code should not be privileged unless really required. Indeed, security rules imply that privileges are granted to limited code paths so that flaws or bugs are less likely to cause serious breaches. Moreover, the logical barrier between kernel modules and user-space prevents the operating systems from having as much knowledge of the application as the user-space libraries have.

The KNEM kernel module aims at offering efficient intra-node MPI communication. It may therefore offer MPI-like interface with send and receive primitives as well as tag matching.[7] However, matching in the kernel module requires the module to maintain queues of send and receive requests and to duplicate the existing user-space MPI matching code. Relying on the user-space MPI implementation to do the matching avoids these constraints while increasing security thanks to the reduction of privileged code. Moreover, it has the advantage of not requiring the kernel module to be aware of all MPI send modes (*buffered*, *ready*, *synchronous*, *blocking* or not, *etc.*). KNEM therefore exposes a **very simple programming interface that assumes that the caller knows which send and receive requests matched**.

The next design issue relates to the way remote buffers are identified. Indeed, direct copy through the operating system requires the driver to know the memory address and layout of the source and destination buffers. Should the process that performs the actual copy be aware of the other process's addresses and layout? If so, the remote process must first pass the description of its memory buffer before the local process can actually perform the copy. This may not be very convenient in cases of non-contiguous buffers such as MPI datatypes. Moreover, a rogue process could modify this information to read other pieces of the remote process memory. The KNEM approach works around these problems by **hiding the remote buffer layout behind an opaque identifier** (a *cookie*). This idea also has the advantage of enabling some optimizations in case of buffer reuse (see Section 4.1).

The KNEM send-receive interface and usage may thus be summarized as follows:

- The sender process declares a send buffer (contiguous or not) in the KNEM driver and passes the corresponding identifier cookie to the receive process.
- The receiver process receives the cookie and requests the KNEM driver to copy from the cookie buffers into its local buffers (contiguous or not).

This model may be easily mapped under the existing MPI implementations because the cookie can be passed along the existing control messages such as in the *rendez-vous* semantics. When the cookie is received, MPI matching is performed as usual to find the corresponding receiver buffer and the KNEM copy can be initiated. The ports of MPI implementations over KNEM will be further detailed in Section 4.4.

3.2. Beyond Send-Receive

The cookie model was initially designed for send-receive communication (*two-sided*) and later generalized to cope with most MPI operation requirements, including RMA (*Remote Memory Access*) and collective operations. Indeed, collectives involve multiple data transfers between the same buffers or part of them. It

[7] Matching is the ability to filter incoming messages based on their tag and associate them to the right receive request.

led us to the idea of reusing the same cookie multiple times without re-declaring it again. It reduces the need for system calls and memory pinning, and also enables *registration-cache* optimizations as explained in Section 4.1.

Implementing a broadcast over the old KNEM send-receive interface requires the MPI implementation to declare the same send buffer once per destination process. The KNEM model actually enables the factorization of these common steps. Then implementing a scatter can be simplified as sending different parts of the same cookie to different peers. Finally this model can be easily reversed to have processes write into the remote region instead of reading. The new extended KNEM interface therefore replaces the existing send cookie with a **generic declared memory region, which can be read or written, partially or entirely, by multiple processes and multiple times .**

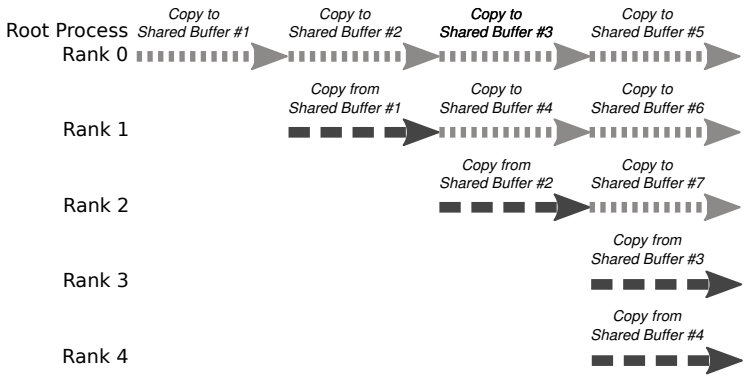
The ability to read or write into remote memory region is very convenient because it **enables the control of which process performs the actual (expensive) copy** while the old send-receive interface always had the copy initiated by the receiver process. It is useful when trying to avoid serialization of independent copies by moving their expensive processing away from bottleneck processes. For instance, rooted collective operations such as broadcast or gather should not let the root process serialize copies, but rather have the leaf processes perform the copies in parallel. A broadcast would therefore keep the copy processing in the receiver processes while a gather would move them into the sender processes.

Another reason for extending the KNEM interface lies in the need of **letting MPI implementations optimize their strategies using all available context information**. The former KNEM send-receive interface was obviously only used by the two-sided back-ends of MPI implementations. Even if other operations such as collectives or RMA can be implemented on top of send-receive, this intermediate layer prevents optimizing KNEM usage because the lower layer cannot easily know what the upper layer is actually doing. Should the copy be moved to the sender process? Will the memory region be reused later? The extended KNEM interface solves this problem by exposing a simple but generic interface that lets the upper layer directly use KNEM without having two-sided constraints in the middle. This idea was also considered in MVAPICH2 by adding yet another kernel module for intra-node RMA instead of using the existing send-receive-only LiMIC module [18]. Our KNEM approach goes further by **supporting send-receive, RMA and collective needs simultaneously**.

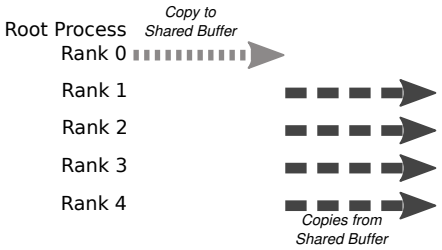
3.3. Opportunities for Optimizing Collective Operations

Figure 2 presents the execution time-line of several implementations of a broadcast operation. Many collective operation algorithms have been proposed in the past [19] and most MPI implementations may now select the right one dynamically [20]. The figure shows that direct copy indeed removes the serialization in the root process and enables the full parallelization of the data transfers. Moreover, large shared caches in modern architectures reduce the number of remote memory reads to one per socket, thus preventing the memory bus from becoming an obvious bottleneck. Commodity implementations require additional steps because each data transfer requires its sender process to perform a memory copy before the receiver can read and use it.

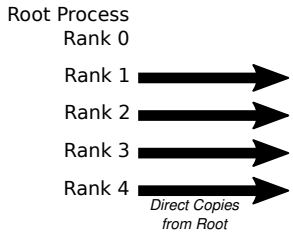
The ability to transfer messages as multiple chunks may further improve the model. Multilevel pipelined algorithms have been widely used in collective operations because it reduces the overall startup time. Figure 3 presents the execution timeline of a pipelined transfer between two processes across an intermediate process. It shows that chunked transfers dramatically reduce the overall time because the second transfer of each chunk can be overlapped with the first transfer of the next chunk. In the shared-memory case the intermediate process is involved in both transfers, and thus causes the sender and receiver processes to wait half the time.



(a) Binary tree with shared-memory based point-to-point operations.

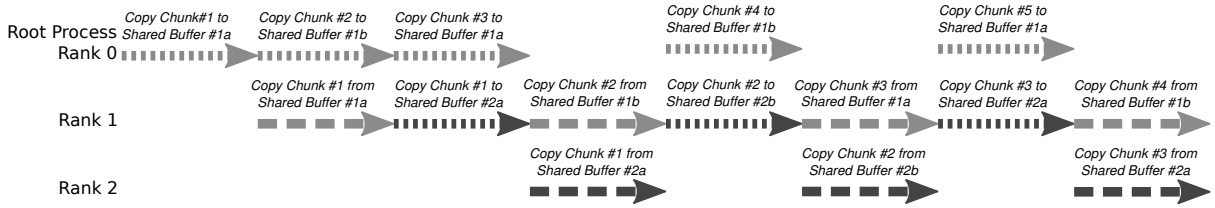


(b) Shared-memory between all processes.

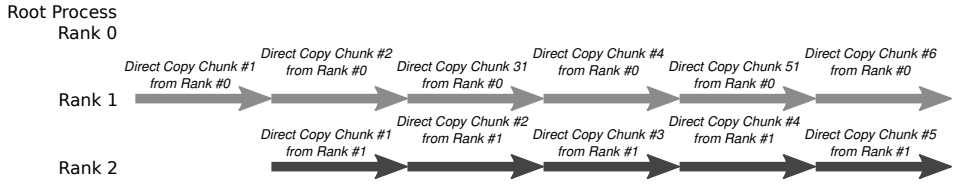


(c) Direct copy.

Figure 2: Copy processing over time among involved processes during a Broadcast operation depending on the implementation.



(a) Shared-memory based point-to-point operations.



(b) Direct copy.

Figure 3: Copy processing over time among involved processes during a two-step pipelined scheme. Process rank #1 serves as an intermediate between #0 and #2.

3.4. KNEM Interface Details

```
knem_create_region
 IN  /dev/knem file descriptor
 IN  memory segments
 IN  protection (read, write)
 IN  flags (singleuse)
 OUT cookie

knem_destroy_region
 IN  /dev/knem file descriptor
 IN  cookie

knem_poll_copy_status
 IN  status address
 OUT status (success, pending, error)
```

```
knem_region_copy
 IN  /dev/knem file descriptor
 IN  source region cookie + offset
 IN  destination region cookie + offset
 IN  length
 IN  flags (asynchronous, I/O AT, etc.)
 IN  status address

knem_inline_copy
 IN  /dev/knem file descriptor
 IN  local memory segments
 IN  remote region cookie + offset
 IN  length
 IN  direction
 IN  flags (asynchronous, I/O AT, etc.)
 IN  status address
```

Figure 4: Summary of the extended KNEM interface.

Now that all requirements have been listed, we describe the extended KNEM interface which is also summarized in Figure 4.[8] As expected, the interface is based on cookie identifiers. Memory regions are created by passing an array of memory segments to the KNEM driver (`knem_create_region`). The array is stored in the driver and a cookie is returned to the caller. The memory region may then be accessed by any process that knows this opaque identifier until it is destroyed. Each region may also optionally be protected for read or write so that a send buffer cannot be written or the receive buffer cannot be read by the peer.

Accessing a remote memory region consists of submitting a copy request to the KNEM driver (`knem_inline_copy`). This request is made of the remote cookie, an offset within its contents, an array of local memory segments, and a copy direction. It is also possible to copy between two cookies, even if none of these memory regions belongs to the caller process (`knem_region_copy`). This variant mostly serves as an optimization for the cases where both source and destination buffers have previously been declared as KNEM regions.

The old KNEM send-receive interface is easily emulated on top of the extended interface. Declaring a send buffer consists of creating a read-only region. Posting a receive request consists of copying from this region. To ensure that the message can only be read from the send buffer once, the region is created with the special `SINGLE_USE` flag. When the copy request is submitted, this flag guarantees that the send region will be destroyed before anybody else can try to access it.

Copies can be processed synchronously or asynchronously. When synchronous, a success or error code is returned at the end of the call. When asynchronous, the caller passes a status address where the success or failure will be written on copy completion. This asynchronous mode opens the opportunity for offloading the copy to dedicated tasks or hardware and communication overlap. All these variants can be requested by the application through request flags.

A copy completion notification is only given to the process that submitted the request. The extended

[8] The KNEM interface is also detailed online at `http://runtime.bordeaux.inria.fr/knem/doc/knem-api.html`.

KNEM interface looks very similar to the *Remote Memory Access* model. Such models have been criticized in the past because implementing MPI on top of them raises scalability and synchronization issues[9]. Fortunately, KNEM targeting intra-node communication makes these problems mostly irrelevant because the flexibility of the pure software implementation avoids hardware scalability problems, and because the shared-memory between all processes makes synchronization much easier.

4. Implementation

In this section, we detail the interface which has been implemented in the KNEM kernel module. The extended interface was first implemented in KNEM release 0.7 and further optimized in release 0.9, which we describe below.

4.1. Core Implementation

KNEM is made of a LINUX kernel module and therefore requires administrator privileges at initialization. This kernel module is accessed by user applications through system calls (*ioctls* on the `/dev/knem` special device file) such as creating a region, destroying a cookie, or requesting a copy.

The KNEM module is primarily based on two object types. A **Context** is an instance of KNEM that a user process owns when opening the `/dev/knem` device file. These entry points are then used to manipulate cookies and submit copy requests. A process cannot use KNEM without any context, but a process can also use multiple contexts simultaneously if different libraries use KNEM for different reasons.[10]

The other important KNEM object is a **Cookie** which corresponds to the opaque identifier of a declared memory region. Cookies are created through the `CREATE` ioctl and attached to the caller context. They may then be accessed by any other context, but only its creator may explicitly destroy it. Cookies and contexts are both stored in *radix-tree* structures[11] that can be read without locking.[12] This ensures scalability when multiple processes submit many copy requests on numerous contexts simultaneously. We indeed observed no significant lookup performance decrease (less than 100 nanoseconds, much less than the duration of a copy) when thousands of contexts or regions are being used simultaneously.

Creating a cookie requires to the driver *pin pages down* in physical memory so that virtual memory can be accessed by other processes. This expensive operation is responsible for most of the overhead of creating a cookie because pinning each physical page costs about 100 nanoseconds on modern machines[13] (30 μs per megabyte). The other creation steps only consist of a system call, allocating a data structure, locking, and inserting in a radix-tree (several hundred nanoseconds total).

The basic implementation of a send-receive over the KNEM kernel driver is summarized in Figure 5. Once the region has been created by the driver (2), the cookie is stored in the driver's internal radix-tree for later use (3). The cookie identifier is then returned to the application (4) which passes it to the remote process (5) (in an out-of-band message such as a usual shared-memory control message). The remote process then submits an inline copy request consisting of its local buffer and the cookie (8). The driver finds the send buffer pages by looking for the cookie in its internal radix-tree and checking that the copy direction

[9] `http://www.hpcwire.com/hpcwire/2006-08-18/a_critique_of_rdma-1.html`

[10] OPEN MPI may open up to one KNEM context per process and per local MPI communicator because each communicator can use different collective implementations depending on its size and placement.

[11] The LINUX *IDR* type is used to translate KNEM integer identifiers into pointers. It scales to thousands of entries with very little overhead during insertion, lookup or deletion.

[12] KNEM makes intensive use of LINUX *Read-Copy-Update* [21] mechanism to avoid locking overhead.

[13] 2.67 GHz INTEL WESTMERE XEON X5650 processors.

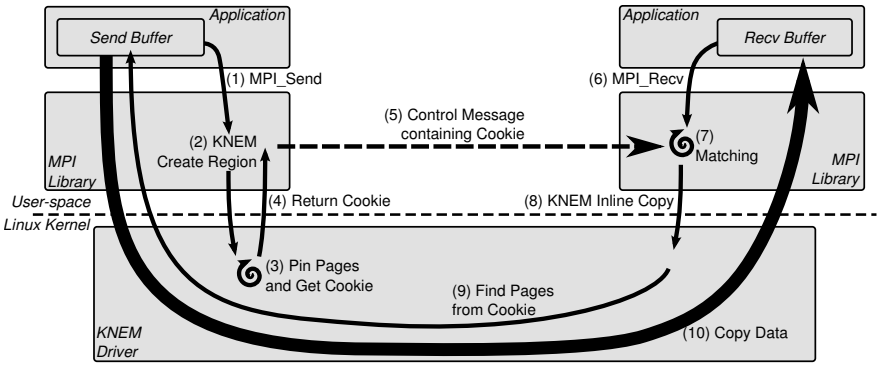


Figure 5: Summary of the implementation of a KNEM send-receive operation.

and cookie protection match (9). If the `SINGLE_USE` flag was specified on region creation, the cookie is immediately removed from the radix-tree when found so that nobody else can use it. The driver then starts copying data between the remote and local buffers (10).

4.2. Security Model

Allowing other processes to read or write in our local memory raises some security issues. The idea behind the KNEM security model is that only memory regions that were explicitly declared to the kernel module can be accessed remotely. Indeed, remote tasks have to use cookies to identify target regions when submitting copy requests. Other memory buffers cannot be accessed unless they were declared and the corresponding cookie was shared.

The opaque cookie identifier actually consists of a KNEM context and memory region. Both of them are checked to verify that the target process indeed opened a KNEM context (enabling some remote access to its memory) and created this exact memory region (enabling actual access to this region). KNEM security checks prevent malicious accesses and return errors if an invalid buffer identifier is passed.

This model is in fact very similar to System V shared memory keys (with `shmget`). A very lucky malicious user trying random 64 bits cookie values may successfully access random memory regions. System V lets sensitive programs reduce this risk by setting Unix group and other access permissions. Although this may be easily implemented in KNEM, we did not do so because KNEM targets HPC platforms. Indeed, a security-sensitive HPC user usually reserves machines in exclusive mode; once no other job can run at the same time, there is no point in trying to prevent them from accessing our memory. Although KNEM could easily restrict access rights to the processes belonging to the user that created the memory region, this idea is also not needed as soon as the node is reserved in exclusive mode. Moreover, access to the KNEM special file may be restricted to only trusted users if needed.

4.3. Synchronous and Asynchronous Data Transfers

Once a KNEM copy request is submitted to the kernel driver and the target cookie has been verified, KNEM initiates the actual data transfer between the remote and local memory buffers. The default strategy consists of copying synchronously during the system call and returning a success code to the user application. This copy can be implemented by temporarily remapping remote region pages into the kernel address space[14] before performing the copy between the local mappings.

[14]The kernel offers the `kmap()` and `kmap_atomic()` functions for temporary mapping of physical pages in the kernel virtual memory.

KNEM also offers the ability to offload this copy to a dedicated kernel thread. This enables overlapping of data transfers with computation by using other cores (or hardware threads) to perform the copy. This can be relevant when the application does not place one task per processing unit. However, if the application uses all cores, having a kernel thread and the application task compete for a single core will likely result in performance decrease.

Memory copies may also be offloaded to hardware *DMA Engines* such as those found in INTEL *I/O Acceleration Technology* (I/O AT[15]). This strategy enables overlapping of data transfers with computation even when all processing units are used by the application.

The KNEM asynchronous model first involves exposing the asynchronous behavior to the user-application. It returns a pending code to the user application at the end of the system call. The actual completion status is returned later at a memory location specified when submitting the copy request. In the thread offload case, the thread performs the copy before writing the success code to this memory location. In the I/O AT case, the success code is directly deposited in user-space by the I/O AT hardware asynchronously, enabling the full overlap of the copy processing [3].

The main drawback of this model is that it is incompatible with some existing notification systems (such as LINUX `epoll`) that are based on a queue of completion events. Moreover there is no way to block until a KNEM copy completes. Fortunately such a feature is rarely useful because most MPI implementations busy poll instead of sleeping when they wait for completions.

4.4. Porting MPI over KNEM

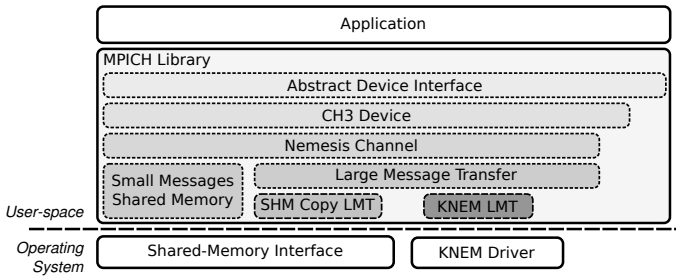


Figure 6: Implementation of point-to-point operations in MPICH2 over KNEM.

KNEM stands for *Kernel Nemesis*. It was initially developed for MPICH2 as a new communication strategy for the NEMESIS communication subsystem. NEMESIS uses two different code paths for small (up to 64 KiB) and large messages. The large message transfer strategy can be chosen at compile time by selecting one of the available *Large Message Transfer* backends (LMT) as depicted on Figure 6. The default LMT consists in copying across a shared memory buffer with a pipeline of 64 KiB chunks.

Each LMT only implements two-sided point-to-point communication. It uses one of the available communication models such as *Put*, *Get*, and *Cooperative*, which differs by the control messages they involve. We added a new KNEM LMT based on the *Get* model [3]: the sender creates a send region and passes the corresponding cookie to the receiver through the *Ready-to-Send* control message. The receiver then initiates a copy using the incoming cookie before sending a *Done* control message back to the sender. These control messages are passed between processes using the shared-memory small messaging strategy which is known

[15] I/O AT [22] is available in INTEL server platforms since 2006. It was advertised as a way to improve the receive side of network stacks [23, 24] but later showed advantages for local copies [25].

to offer very good latency (up to 200 nanoseconds on modern processors), making this overhead mostly negligible against the actual data transfer.

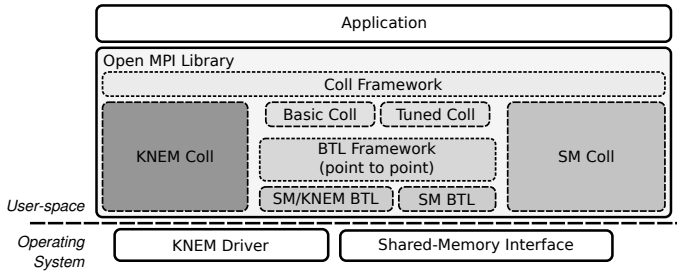


Figure 7: Implementation of point-to-point and collective operations in Open MPI over KNEM.

KNEM was later added to Open MPI as part of the existing *Shared-Memory* point-to-point component (`sm` *Byte Transfer Layer*). The Open MPI dynamic model lets the user select components and configure them at runtime. It is therefore possible to easily configure which message sizes will be transferred with KNEM or not, and with I/O AT copy offload or not.

Both Open MPI and MPICH2 switch between a network-specific backend for inter-node communication and a local backend with KNEM support for intra-node. This ability to optimize each data transfer offers an efficient answer to the need for high-performance intra-node communication. Tools such as hwloc may be used to detect the local topology so as to adapt KNEM use to shared caches or NUMA distance by having the threshold vary with the peer locality [26, 5].

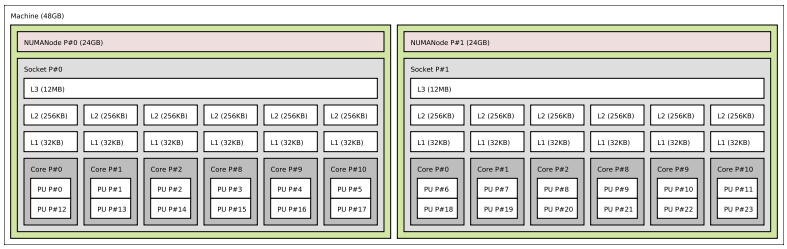
As shown by Figure 7, the component architecture of Open MPI supports having multiple collective operation implementations at the same time. The usual collective components (*Basic* and *Tuned*) are available on top of point-to-point messages (BTL), with or without KNEM. Similarly, MPICH2 implements one-sided communication and collective operations on top of the aforementioned point-to-point model. Therefore, they cannot benefit from the KNEM extended interface because all operations are broken down to two-sided communication with contiguous buffers. We added a dedicated Open MPI KNEM collective component (*knem-coll*) that bypasses point-to-point messages and uses the extended KNEM interface directly [6]. It is then possible to implement advanced uses of KNEM to optimize collective operations by reusing KNEM memory regions multiple times, using vectorial buffers, adapting copy direction to the communication scheme, *etc.*, as described in Section 3.3.

5. Performance Evaluation

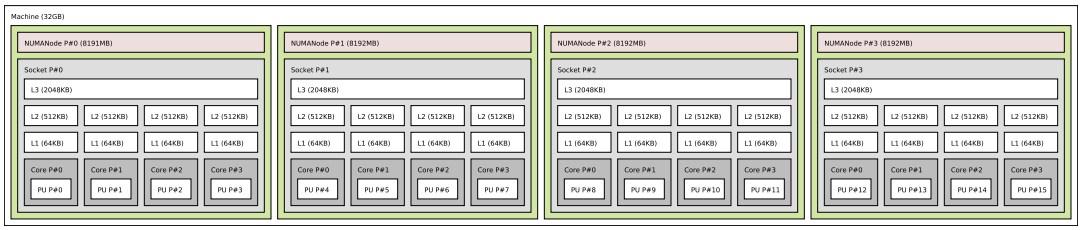
We present in this section the performance of KNEM, from dedicated microbenchmarks to MPI benchmarks and real applications.

5.1. Methodology

One has to keep in mind when benchmarking intra-node communication that the performance is strongly dependent on the architecture and task placement. Indeed shared caches and NUMA distance may dramatically modify the data transfer throughput [3]. Figure 8 presents some common server architectures among our experimentation platforms. It shows that shared caches are indeed widely used in modern servers. Each processor usually has one higher level cache shared by all its cores. However, it should be noted that some processor models (such as AMD *Magny-Cours*) actually contain two identical dies, making this large cache shared by only half of the processor cores.



(a) A hyperthreaded dual-socket hexa-core INTEL XEON *Westmere* X5650 NUMA server.



(b) A quad-socket quad-core AMD OPTERON *Barcelona* 8437HE NUMA server.

Figure 8: Topology of some of our experimentation platforms as reported by HWLOC's `lstopo` program.

Such cache characteristics cause the data transfer performance to vary significantly depending on the process placement because the existence of a shared cache and its size are critical to the copy performance. It is therefore important to check whether a transfer has its source data available in any cache before trying to understand intra-node communication performance.

We first focus on KNEM point-to-point performance and explain how it behaves depending on caching and depending on the data transfer strategy. We then further focus on topology to emphasize that the MPI implementation should carefully select the best strategy according to the current message(s) and process placement(s). We finally look at collective operations and application-level performance improvements.

OPEN MPI version 1.5 and MPICH2 1.3.1 over KNEM 0.9.7 were used together with the *Intel MPI Benchmarks* (IMB) version 3.2 [27]. It offers the `-off_cache` option to control buffer reuse so as to better understand the impact of caches on communication performance.

5.2. KNEM Basic Point-to-point Performance

We present in this section a basic performance comparison of MPI point-to-point communication performance with KNEM versus the legacy double-copy shared-memory implementation. Figure 9 shows the throughput of the IMB *Pingpong* benchmark on a INTEL platform. Cases entitled *Shared Cache - Oncache* consist in the most cache-friendly case: a large cache is shared between the communicating processes and the same buffers are reused in each iteration. Contrariwise *Different Sockets - Offcache* is the most cache unfriendly case because no cache is shared and different buffers are used in each iteration.

It shows that KNEM benefits the MPI intra-node point-to-point throughput without depending on the application behavior with respect to caches. In all cases, including other platforms, KNEM offers a throughput that is either very close or higher than the existing shared-memory implementation. When no shared-cache and buffer-reuse is involved, KNEM achieves between 18% and 30% higher throughput because its single-copy model is far less dependent on caches. The double-copy model is competitive only when the data buffer fits in a shared cache. However KNEM is obviously not competitive for small message size, usually

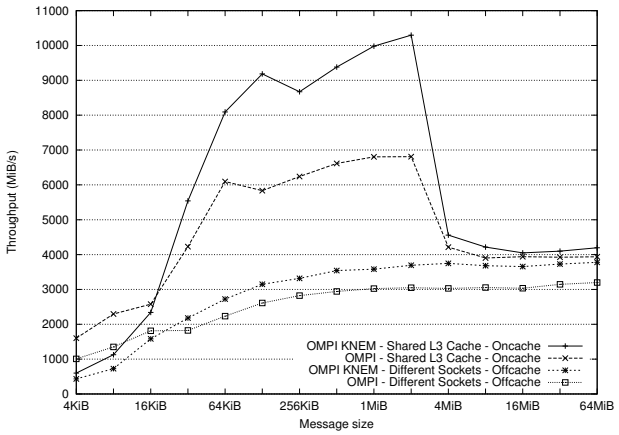


Figure 9: IMB Pingpong performance with or without KNEM, with or without caching, on a dual-socket INTEL *Westmere* platform (Figure 8(a)).

less than 16 kB, because of its system call overhead. It means that applications that do not use medium or large messages (at least tens of kilobytes) will not benefit from KNEM significantly.

Figure 10 presents the same evaluation on our AMD platform. Surprisingly, KNEM only improves the Pingpong throughput when the message fits in the shared cache. Overall the default shared-memory implementation is slightly faster for large messages, especially when there is no shared cache. This behavior is different from the one of our INTEL platform above. This is caused by the architectural differences in memory coherency protocols and implementations in the AMD and INTEL platforms. It appears that the AMD platform successfully handles the double copy overheads on memory bandwidth, cache pollution and CPU utilization while the INTEL platforms did not.

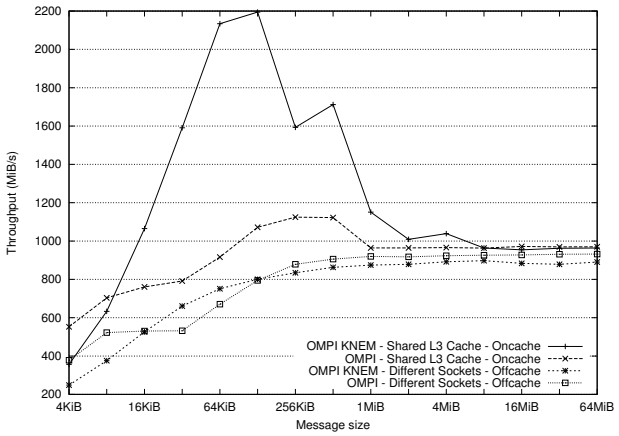
The second subfigure shows that adding contention to the problem makes KNEM benefits appear as expected. A Pingping consists of having both processes send pings and pongs simultaneously. When two processes communicate at the same time, the KNEM per-process throughput remains unchanged while the shared-memory throughput is almost divided by two. This shows that KNEM's obvious advantages (reducing the CPU utilization, cache pollution, and memory bandwidth by using a single copy) benefit much more on real communication patterns than on point-to-point benchmarking patterns on idle machines.

5.3. Understanding the Performance of KNEM Copy Strategies

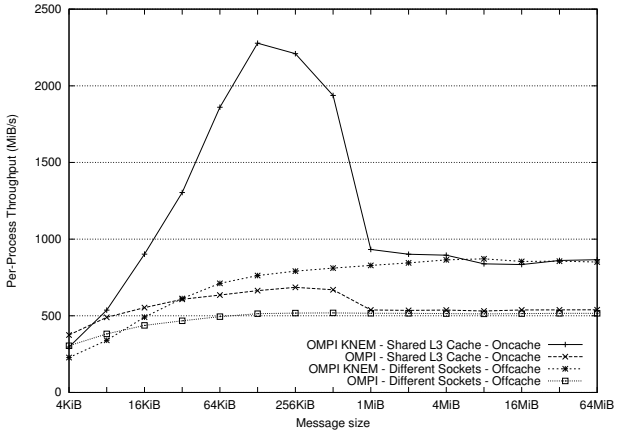
We now take a deeper look at how to benefit from kernel-assisted data-transfer by comparing the existing KNEM copy strategies so as to understand which one should preferably be used. This study relies on KNEM custom microbenchmarks because the MPI implementations have not been designed to test all strategies enabled by the extended KNEM interface.

KNEM offers a first data transfer alternative by letting the application decide whether the sender or the receiver process performs the actual copy. *Sender-writing* is similar to RDMA *Put* while *Receiver-reading* looks like an RDMA *Get* (and is very close to the old KNEM send-receive interface). Their implementations in KNEM are entirely symmetrical. We therefore measured very similar raw performances on benchmarks (see Figure 11). The application behavior should thus be the key when deciding between these schemes:

- If the data to be transferred was stored in the source buffer recently and will not be read soon by the receiver, it is better to perform the copy in the sender to benefit from its hot cache.



(a) IMB Pingpong throughput



(b) IMB Pingping per-process throughput

Figure 10: IMB Pingpong and Pingping performance with or without KNEM, with or without caching, on a quad-socket AMD *Barcelona* host (Figure 8(b)).

- If the data was prepared by the sender a long time ago and is immediately needed by the receiver, performing the copy in the receiver will improve cache reuse.

Another decision criterion lies in collective algorithms. As explained in Section 3.3, rooted operations should be implemented by moving the copy processing away from the root process bottleneck.

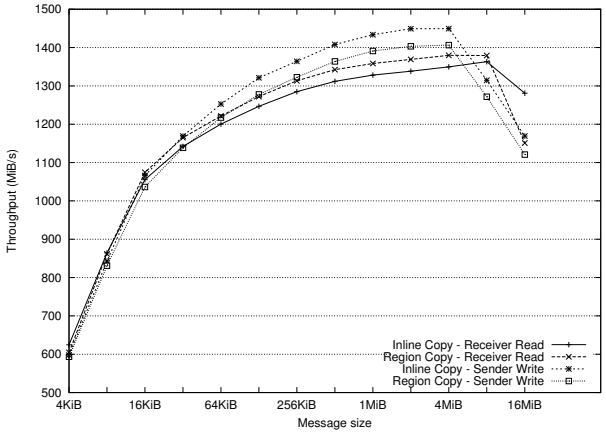


Figure 11: Performance of KNEM copy strategies without buffer reuse between different sockets of a dual INTEL *Westmere* host (Figure 8(a).

The other copy strategy choice in KNEM relates to how the buffers are identified. The usual KNEM *inline copy* strategy can only transfer data between a declared memory region and a local buffer. The *region copy* alternative can copy between regions, even if none of them belongs to the caller process. This model is more flexible but requires both source and destination buffers to be declared as KNEM regions. Declaring a region implies a system call and the pinning of all virtual pages into physical memory. We measured a base overhead of 0.5 to 1 μs on our platforms and a per-page overhead of 30 to 180 ns. Given the raw memory copy performance on these hosts, this additional pinning implies a theoretical throughput decrease of less than 10% when compared to the inline copy model. However, this overhead may be factored out if multiple data transfers reuse this region without redeclaring it each time.

The other difference between *inline* and *region* copy models lies in the way the memory is accessed. Regions require a temporary remapping of the application memory in the kernel while the inline copy accesses the application mapping directly. Such a remapping is not very expensive (only the local processor MMU is updated), but large copies imply more TLB misses because the application and the kernel copy do not use the same virtual addresses. Also, the data may not actually be shared between the kernel and application mappings if the cache uses virtual addresses. Fortunately this case is not widely used by modern processors.[16]

Figure 11 compares the performance of these strategies in a KNEM-specific benchmark. Due to the very different behaviors of the copy strategies with respect to caches it is important to make sure that the source and destination buffers are actually used during this test. This pingpong therefore initializes the source buffer before sending and reads it after receiving.[17] Surprisingly the *region copy* shows very similar performance to the *inline copy* even if it involves page pinning and remapping. The reason may be that the *inline copy*

[16]Only the first level data cache (L1d) is virtually indexed (and physically tagged) in our platforms. Such caches (32 or 64 KiB) are often entirely flushed when copying a KNEM message anyway (usually tens or hundreds of kilobytes).

[17]It explains why the overall throughput is 2-3 times lower than in other benchmarks.

model has other overheads that are hidden inside direct accesses to user-space application memory because it has to handle pageable memory.

Overall, this section shows that KNEM offers four different copy strategies whose throughputs are similar despite their very different implementations and interfaces. This allows the developer to use any of these strategies depending on regions being already declared, data being cache-hot, and collective operations being involved.

5.4. Architecture and Topology-awareness

KNEM was first designed to avoid double copies because they consume more CPU cycles and memory bandwidth, and cause more cache pollution. This is obviously mostly relevant for large messages. KNEM was not designed for small message latency where system calls should generally be avoided in favor of user-space shared-memory. The question that arises is which MPI message sizes should actually be transferred using KNEM. We demonstrated in the previous section that KNEM copy performance does not vary significantly with the copy strategy. However, one still has to consider the additional ability to offload this copy to dedicated DMA hardware such as INTEL I/O AT. The MPI implementation therefore actually has to select among at least three strategies when sending a message: shared-memory double-copy, KNEM, and KNEM with I/O AT offload.

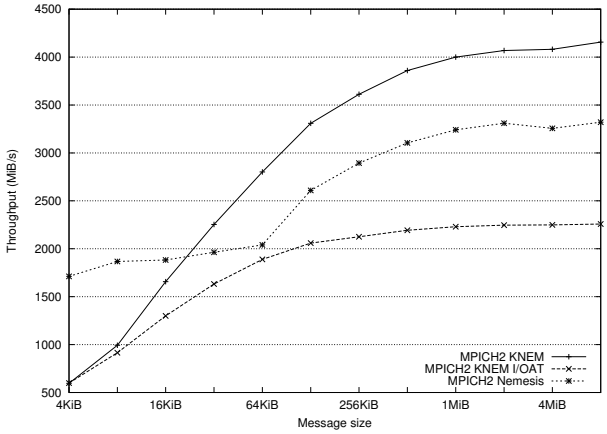


Figure 12: Comparison between IMB Pingpong throughputs using the native MPICH2/NEMESIS implementation, MPICH2 over the KNEM LMT, and MPICH2 over KNEM with I/O AT copy offload, on a dual-socket INTEL *Westmere* platform (Figure 8(a)), with no cache reuse and different sockets.

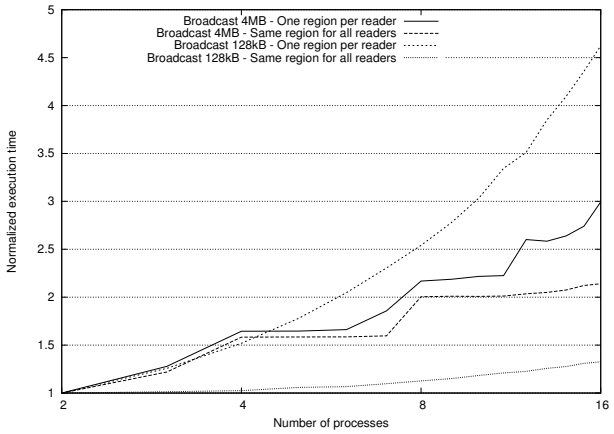
Several papers presented numerous graphs comparing the performance of these strategies under MPICH2 [3, 4] and OPEN MPI [5]. The ideal thresholds for switching between strategies depends on the architecture and on the process placement. Figure 12 shows that, in this case, MPICH2 should start using KNEM once the message size reaches 32 kB. However I/O AT copy offload is never useful on such a recent platform which exhibits much higher CPU copy throughput. This behavior is very different from older non-NUMA platforms where I/O AT was always useful for large messages [3]. The reason is that I/O AT DMA performance has not improved significantly since 2007. It is therefore only useful for overlap on modern servers.

Some heuristics for predicting these thresholds have been proposed [3, 5] together with topology detection tools such as HWLOC [26]. Looking at cache sizes and checking whether they are shared between the communicating cores gives useful hints. However the application behavior, especially when it comes to collective communication patterns, has a huge impact on the actual performance of each transfer strategy [4].

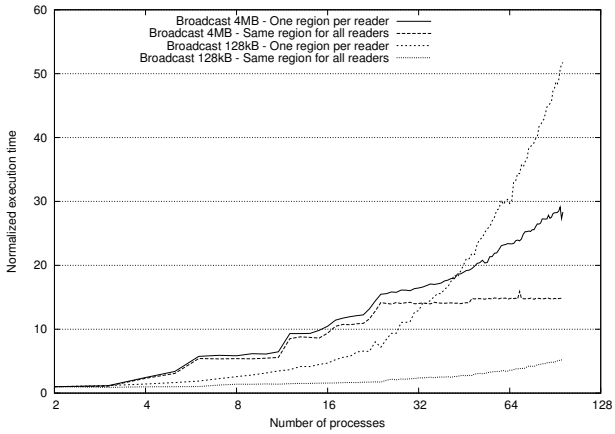
The optimal thresholds may therefore hardly be decided on a point-to-point basis and they should rather be decided dynamically, for instance with auto-tuning techniques.

5.5. Collectives Operations

As explained in the previous section, KNEM improves large message performance by reducing memory copies. This is especially meaningful in cases of heavy memory contention under complex communication patterns such as MPI collective operations. The extended KNEM programming interface was designed to target collectives by enabling the reuse of memory regions in multiple data transfers.



(a) Quad-socket quad-core AMD Opteron (Figure 8(b)).



(b) Sixteen-socket hexa-core Intel Xeon *Dunnington* (96 cores, NUMA)

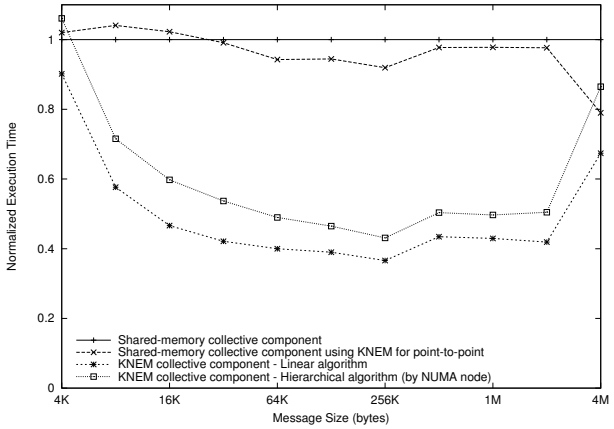
Figure 13: Comparison of the scalabilities of broadcast operations over KNEM with a single multiply-used source region and with one dedicated region per reader. Execution times are normalized to the operation between 2 processes.

Figure 13 presents the benefits of using a common source region for all destination peers during a broadcast implemented directly over KNEM. When the number of processes is small, using different regions for each message does not degrade performance significantly. However, using this strategy across the entire 16-core machine makes broadcast operations 3 to 4.6 times slower than between 2 processes. Reusing the same source region for all peers thanks to the extended KNEM interface improves scalability and decreases

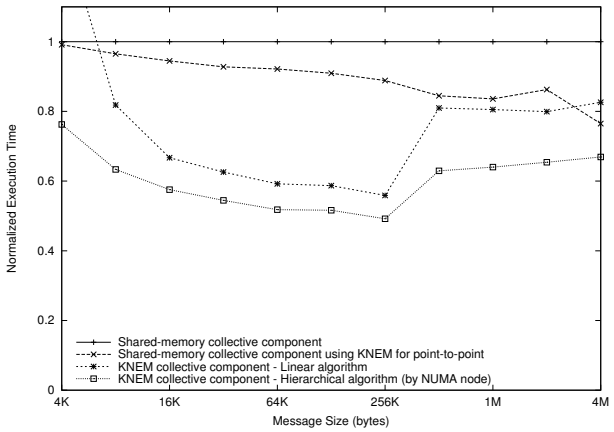
this factor down to 1.3-2.2×. On a 96-core machine, reusing the same region decreases the broadcast normalized time for 28-62× to 5-15×. Moreover we observe that the improvement is larger for small messages (128 KiB). It means that creating and destroying regions is actually not negligible for such message sizes. For larger messages (4 MiB) the improvement is smaller because reusing the same region does not reduce memory copies, it only reduces management overhead and improves cache and TLB reuse.

We ran similar tests on other collective operations such as Scatter and Gather and observed smaller improvements because the root process buffer is split between other processes during these operations. Not reusing the same region for all peers therefore means that we create multiple *smaller* regions. The overhead of creating these smaller regions thus degrades performance less significantly than during a broadcast.

The next step consists in leveraging these features in MPI implementations. This has been implemented as the OPEN MPI KNEM collective component as explained in Section 4.4. Figure 14 shows how it improves performance over the existing component. We observe 30-40% lower execution time for all message sizes between 4 KiB and 4 MiB on all our platforms while only using KNEM for point-to-point data transfers only significantly improves large messages.



(a) Dual-socket INTEL *Westmere* (Figure 8(a))



(b) Quad-socket quad-core AMD OPTERON (Figure 8(b)).

Figure 14: IMB Bcast performance comparison between the OPEN MPI shared-memory collective component with and without KNEM for underneath point-to-point transfers, and the native KNEM collective component. Execution times are normalized to the default component (shared-memory without KNEM).

Table 1: Execution time of some NAS Parallel Benchmarks on a 16-core AMD platform (Figure 8(b)).

NAS Kernel	Default Open MPI	KNEM kernel copy	Speedup
ft.B.16	69.38 s	62.63 s	**+ 10.8%**
is.C.16	82.86 s	70.07 s	**+ 18.2%**

Figure 14(b) also shows that the KNEM collective component can be further enhanced on large NUMA platforms by making it hierarchical. Using one process as an intermediate step on each NUMA node reduces the overall execution time by 10% to 20% because it benefits from the shared L3 cache inside each NUMA node. It should also be noted that this case can benefit from the *region* copy request because intermediate buffers that are read from the root process have to be declared as a KNEM region before the leaves read from them.

This work led to the native implementation of other collective operations over KNEM, showing benefits on many multicore and manycore platforms [6].

5.6. Applications

We now present the impact of KNEM on real applications performance by looking at point-to-point data transfers and collective operations.

We first focus only on point-to-point operations, or collective operations implemented over point-to-point. Table 1 presents the execution times of some NAS Parallel Benchmarks [28]. Most of these benchmarks do not send many large messages and therefore show small or insignificant changes in performance. However, IS, which is known to use very large messages, shows an 18% performance improvement when using KNEM. FT is also improved by 10%. On older non-NUMA platforms, we observed up to a 25.8% speedup thanks to I/O AT copy offload, and the improvement seemed linear with the total number of cache misses [3]. KNEM using a single memory copy implies less cache pollution, therefore less cache misses and better performance.

We next focus on collective operations and therefore look at their native implementation over KNEM. ASP [29] is a parallel implementation of the Floyd-Warshall algorithm that solves the all-pairs-shortest-path problem. We tested this application on one large NUMA manycore host: a 48-core machine made of eight six-core Opteron processors. The main MPI collective operation used in this application is the Broadcast. We compare the native hierarchical pipelined KNEM collective implementations (see Section 5.5) with the existing Open MPI and MPICH2 implementations that only use KNEM for point-to-point transfers. The problem size is scaled to match the available memory. Matrices are distributed by rows across all the available cores.

Matrix size (32bits integers)	32768^2	
Bcast operations	32768×128 KiB	
Execution times	Bcast	Total
MPICH2 with KNEM point-to-point	293.9s	6413.8s
Open MPI with KNEM point-to-point	550.2s	6650.9s
Open MPI with KNEM collectives	198s	6288.1s
Speedup	+48%	+2%

Table 2: Execution time of the ASP application depending on the MPI collective implementation. The speedup compares our KNEM collective implementation with the best performing existing MPI library (Open MPI or MPICH2).

Table 2 presents the execution time of the ASP application. Thanks to the native KNEM collective implementation, the application gets a significant improvement in the time spent doing broadcast operations with a speedup of +48%. Then the shorter time spent in the collective operations leads to reducing the overall application runtime by 2%. Some other tests on a 4-socket non-NUMA machine, whose outdated memory interconnect hardly scales to its 16 cores, even shows 5× faster Bcast operations thanks to KNEM reduced memory bandwidth usage.

Our NAS results showed that KNEM may significantly help real application performance by improving point-to-point operations while ASP experiments show that the native KNEM collective operations bring further significant efficiency. Other applications such as CPMD or FFTW have been recently studied and showed similar improvements thanks to KNEM [30].

6. Related Works

We described in Section 2.3 and Figure 1 different models for offering high-performance intra-node communication. Aside from network-dependent solutions, several operating system extensions have been proposed. For instance SMARTMAP enables direct access to other processes' memory in the Catamount lightweight kernel [31] but it is only available on Cray XT platforms. The XPMEM Linux kernel module which offers remapping of others processes' memory to enable similar optimization is also restricted to SGI machines.[18] Even if they are network agnostic, they are still not portable to commodity clusters.

The most important alternatives to KNEM are LiMIC and CMA. LiMIC [32, 33] is a Linux kernel module designed to offer kernel-assisted direct copy between MPI processes using the MVAPICH2 implementation [34]. The first revision offered matching inside its kernel driver [35] but it now only provides a send-receive interface for reasons detailed in Section 3.1. CMA (*Cross-Memory-Attach* [36]) brought very similar features in the Linux kernel 3.2 as two new system calls. However, no popular MPI implementation uses it yet. Some alternatives targeting MPI one-sided operations (remote memory accesses) specifically were also proposed [18], but the corresponding source code is not available for testing.

A major difference between these solutions lies in the awareness of the remote process addresses and layout. LiMIC, SMARTMAP and CMA let the copy initiator manage the remote address space layout. The target process must therefore first pass the description of its memory buffer before the initiator process can actually perform the copy. This may not be very convenient in cases of highly non-contiguous buffers such as MPI datatypes. Moreover, a rogue process could modify this information to read other pieces of the remote process memory. KNEM lets the copy initiator only manipulate opaque cookies. The exact list and layout of remotely accessible buffers is only managed by the target process when creating or destroying its own memory regions.

The main drawback of LiMIC lies in the severe security issue that it raises and that prevents it from being used on production machines. Indeed it does not check the remote region identifier given by the user-space process. Once the LiMIC kernel driver is loaded, it is possible to access the memory of all processes in the machine, even the privileged ones that do not use LiMIC. And passing an invalid LiMIC buffer identifier would crash the kernel by trying to access a non-existing address space or a non-existing memory region in an existing process. KNEM opaque cookies and security checks avoid these issues and return an error in such cases. It makes our work widely usable without facing security problems on production machines.

Section 4.1 describes the overhead of creating these regions. It might be seen as a KNEM drawback when compared to alternatives such as LiMIC or CMA that do not require such work because they have no

[18] A port of XPMEM to Cray machines is currently under early development.

notion of cookie. However, it is important to understand that LiMIC and CMA also need to pin pages down in physical memory.[19] They perform this step just before copying the data. Contrary to KNEM which can factor out this overhead by declaring a memory region once and reusing it multiple times, LiMIC and CMA require the very same pinning overhead for each copy request.

We compared the pingpong performance of MVAPICH2 1.8rc1 over LiMIC with Open MPI over KNEM and observed that the former is usually 3 to 6 μs faster on our platforms. We assume that about 2 μs come from KNEM security checks and the additional region declaration and lookup operations. The remaining difference may lie in the different MPI implementations considered here. Fortunately, these microseconds are only meaningful for medium-sized messages in cache-friendly cases. In other cases, we observed very similar throughput for MVAPICH2 over LiMIC and Open MPI over KNEM.

7. Conclusion and Future Work

As multicore processors are now widely used in high performance computing, the degree of parallelism inside nodes increases. High performance MPI communication is required between the computing tasks both across the networks and inside shared-memory nodes. In this paper, we presented in this paper an in-depth analysis of KNEM, a Linux module that offers an efficient interface for kernel-assisted direct copies between local processes. KNEM provides MPI implementations with a generic and scalable interface that enables high-performance flexible data transfers for point-to-point and collective operations.

KNEM brings significant throughput improvement to MPI communication on modern computing platforms because its single copy model reduces the CPU load, cache pollution, and memory bandwidth waste. It is already supported by most existing MPI implementations [3, 5] and the use of its native interface for collective operations in Open MPI shows promising results in real world applications [30, 37]. KNEM only benefits to intra-node communication, but it obviously adds a significant value to the general distributed case where both intra-node and inter-node communication are involved. For instance combining KNEM with advanced multilayer collective operation implementations has been proven to bring significant performance improvement [38].

We expect KNEM to bring additional improvements in the future when the third revision of the MPI standard will be released.[20] For instance, we did not detail the performance of KNEM asynchronous memory copies in this paper because they are not widely used by existing MPI implementations yet. We observed a performance decrease of 10% to 20% compared to the synchronous operations (when not offloading to the core that runs an MPI process). We will revisit this case once MPI non-blocking collective operations will be available in MPI implementations[21]. Indeed, non-blocking collective operations could combine the already demonstrated benefits of KNEM for collectives with its ability to offload data transfers on idle cores or dedicated DMA hardware. Moreover KNEM may also significantly help MPI remote memory accesses (RMA), whose interface is being revisited in MPI 3.0 as well.

Our work on KNEM led to a widespread use of portable and hardware-independent kernel-assisted intra-node communication. Its availability in most existing MPI implementations now raised the need for a standard solution. The integration of kernel-assisted point-to-point data-transfer with *Cross-Memory-Attach* [36] in the upcoming Linux kernel 3.2 is one step toward this goal. The next step will consist of

[19] KNEM, LiMIC and CMA all rely on the `get_user_pages()` kernel routine for pinning pages and translating virtual addresses into physical.

[20] MPI 3.0 is currently expected to be released during fall 2012.

[21] Non-blocking collective operations will be included in the upcoming 3.0 revision of the MPI standard.

including other KNEM features such as collective or asynchronous operations support in standard LINUX distributions.

Acknowledgements

The authors would like to thank David Goodell and Darius Buntinas from Argonne National Laboratory for porting MPICH2 over KNEM and contributing to the initial design of the model; George Bosilca and Teng Ma from the University of Tennessee, Knoxville, and Jeff Squyres from CISCO SYSTEMS for implementing OPEN MPI BTL and collective components over KNEM, and participating in the design of the extended interface; and Damien Guinier and Sylvain Jeaugey from BULL for helping with understanding the performance behavior of KNEM-enabled MPI layers.

References

[1] Top500 Supercomputing Sites, `http://top500.org`.

[2] M. P. I. Forum, MPI: A Message-Passing Interface Standard, Tech. Rep. UT-CS-94-230 (1994).

[3] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, S. Moreaud, Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis, in: Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009), IEEE Computer Society Press, Vienna, Austria, 2009, pp. 462–469.

[4] S. Moreaud, B. Goglin, D. Goodell, R. Namyst, Optimizing MPI Communication within large Multicore nodes with Kernel assistance, in: CAC 2010: The 10th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010, IEEE Computer Society Press, Atlanta, GA, 2010.

[5] T. Ma, G. Bosilca, A. Bouteiller, J. J. Dongarra, Locality and Topology aware Intra-node Communication Among Multicore CPUs, in: Proceedings of the 17th European MPI Users Group Conference, Lecture Notes in Computer Science, Springer, Stuttgart, Germany, 2010.

[6] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, J. J. Dongarra, Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs, in: Proceedings of the 40th International Conference on Parallel Processing (ICPP-2011), Taipei, Taiwan, 2011.

[7] OpenMP: The OpenMP API specification for parallel programming, `http://openmp.org`.

[8] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, R. Thakur, Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming, International Journal of High Performance Computing Applications 24 (2010) 49–57.

[9] E. Jeannot, G. Mercier, Near-optimal placement of MPI processes on hierarchical NUMA architecture, in: Proceedings of the 16th International Euro-Par Conference, Lecture Notes in Computer Science, Lecture Notes in Computer Science, Springer, Ischia, Italy, 2010.

[10] E. Jeannot, G. Mercier, Improving MPI Applications Performance on Multicore Clusters with Rank Reordering, in: Recent Advances in the Message Passing Interface. The 18th European MPI User's Group Meeting (EuroMPI 2011), Lecture Notes in Computer Science, Springer-Verlag, Santorini, Greece, 2011.

[11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W.-K. Su, Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro 15 (1) (1995) 29–36.

[12] M. Koop, W. Huang, K. Gopalakrishnan, D. K. Panda, Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand, in: 16th IEEE Int'l Symposium on Hot Interconnects (HotI16), Palo Alto, CA, 2008.

[13] P. Geoffray, L. Prylli, B. Tourancheau, BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs, in: Proceedings of the 1999 ACM/IEEE conference on Supercomputing, Portland, OR, 1999.

[14] B. Goglin, High Throughput Intra-Node MPI Communication with Open-MX, in: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009), IEEE Computer Society Press, Weimar, Germany, 2009, pp. 173–180.

[15] Myricom, Inc, Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet, `http://www.myri.com/scs/MX/doc/mx.pdf` (2006).

[16] D. Buntinas, G. Mercier, W. Gropp, Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem, Parallel Computing, Selected Papers from EuroPVM/MPI 2006 33 (9) (2007) 634–644.

[17] D. Buntinas, G. Mercier, W. Gropp, Data Transfers between Processes in an SMP System: Performance Study and Application to MPI, Parallel Processing, 2006. ICPP 2006. International Conference on (2006) 487–496.

[18] P. Lai, S. Sur, D. K. Panda, Designing Truly One-Sided MPI-2 RMA Intra-node Communication on Multi-core Systems, in: Proceedings of the International Supercomputing Conference (ISC'10), Hamburg, Germany, 2010.

[19] R. Thakur, Improving the Performance of Collective Operations in MPICH, in: Proceedings of the 10th European PVM/MPI Users Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2003), Vol. 2840 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 257–267.

[20] J. M. Squyres, A. Lumsdaine, The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms, in: V. Getov, T. Kielmann (Eds.), Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications, Springer, St. Malo, France, 2004, pp. 167–185.

[21] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, R. Russell, Read Copy Update, in: Proceedings of the Linux Symposium (OLS2002), Ottawa, Canada, 2002, pp. 338–367.

[22] Accelerating High-Speed Networking with Intel I/O Acceleration Technology, `http://www.intel.com/content/www/us/en/io/i-o-acceleration-technology-paper.html` (May 2005).

[23] A. Grover, C. Leech, Accelerating Network Receive Processing (Intel I/O Acceleration Technology), in: Proceedings of the Linux Symposium (OLS2005), Ottawa, Canada, 2005, pp. 281–288.

[24] R. Huggahalli, R. Iyer, S. Tetrick, Direct Cache Access for High Bandwidth Network I/O, SIGARCH Computer Architecture News 33 (2) (2005) 50–59.

[25] K. Vaidyanathan, L. Chai, W. Huang, D. K. Panda, Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT, in: Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07), Austin, TX, 2007, pp. 159–168.

[26] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010), IEEE Computer Society Press, Pisa, Italia, 2010, pp. 180–186.

[27] Intel MPI Benchmarks, `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[28] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, The NAS Parallel Benchmarks, The International Journal of Supercomputer Applications 5 (3) (1991) 63–73.

[29] A. Plaat, H. E. Bal, R. F. H. Hofman, T. Kielmann, Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects, Future Generation Computer Systems 17 (6) (2001) 769–782.

[30] T. Ma, A. Bouteiller, G. Bosilca, J. J. Dongarra, Impact of Kernel-Assisted MPI Communication over Scientific Applications: CPMD and FFTW, in: Proceedings of the 18th European MPI Users Group Conference, Lecture Notes in Computer Science, Springer, Santorini, Greece, 2011.

[31] R. Brightwell, T. Hudson, K. Pedretti, SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor, in: Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2008, ACM Press, Austin, TX, 2008.

[32] H.-W. Jin, S. Sur, L. Chai, D. K. Panda, Lightweight Kernel-Level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems, in: Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07), Austin, TX, 2007.

[33] L. Chai, P. Lai, H.-W. Jin, D. K. Panda, Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems, in: Proceedings of the IEEE International Conference on Parallel Processing (ICPP-2008), IEEE Computer Society Press, Portland, Oregon, 2008.

[34] Network-Based Computing Lab, The Ohio State University, MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, `http://mvapich.cse.ohio-state.edu/`.

[35] H.-W. Jin, S. Sur, L. Chai, D. K. Panda, LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster, in: Proceedings of the IEEE International Conference on Parallel Processing (ICPP-2005), IEEE Computer Society Press, Oslo, Norway, 2005.

[36] C. Yeoh, Cross Memory Attach, `http://lwn.net/Articles/405284/` (2010).

[37] T. Ma, T. Herault, G. Bosilca, J. J. Dongarra, Process Distance-aware Adaptive MPI Collective Communications, in: International Conference on Cluster Computing (IEEE Cluster), IEEE Computer Society Press, Austin, Texas, 2011.

[38] T. Ma, G. Bosilca, A. Bouteiller, J. J. Dongarra, HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters, in: Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS'12), Shanghai, China, 2012.