Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs

Teng Ma, George Bosilca, Aurélien Bouteiller, Brice Goglin, Jeffrey Squyres, Jack Dongarra

Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs

Teng Ma*, George Bosilca*, Aurelien Bouteiller*, Brice Goglin†, Jeffrey M. Squyres‡, Jack J. Dongarra*§¶
* *EECS Department, University of Tennessee - Knoxville TN, USA*
† *INRIA, LaBRI, University of Bordeaux - Talence cedex, France*
‡ *Cisco Systems, Inc. - San Jose CA, USA*
§ *Oak Ridge National Laboratory - Oak Ridge TN, USA*
¶ *University of Manchester - Manchester, UK*
{*tma, bosilca, bouteill, dongarra*}*@eecs.utk.edu, Brice.Goglin@inria.fr, jsquyres@cisco.com*

***Abstract*—Shared memory is among the most common approaches to implementing message passing within multi-core nodes. However, current shared memory techniques do not scale with increasing numbers of cores and expanding memory hierarchies – most notably when handling large data transfers and collective communication. Neglecting the underlying hardware topology, using copy-in/copy-out memory transfer operations, and overloading the memory subsystem using one-to-many types of operations are some of the most common mistakes in today's shared memory implementations. Unfortunately, they all negatively impact the performance and scalability of MPI libraries – and therefore applications.**

In this paper, we present several kernel-assisted intra-node collective communication techniques that address these three issues on many-core systems. We also present a new Open MPI collective communication component that uses the KNEM Linux module for direct inter-process memory copying. Our Open MPI component implements several novel strategies to decrease the number of intermediate memory copies and improve data locality in order to diminish both cache pollution and memory pressure. Experimental results show that our KNEM-enabled Open MPI collective component can outperform state-of-art MPI libraries (Open MPI and MPICH2) on synthetic benchmarks, resulting in a significant improvement for a typical graph application.

***Keywords*-MPI, multi-core, many-core, shared memory, NUMA, kernel, collective communication**

I. Introduction

In recent years, thermic issues have prevented the straightforward performance improvement of increasing processor operating frequency. Maintaining an increase in performance therefore had to be achieved a different way. Parallelism on a chip – in the form of multi-core processors – has been widely adopted as the solution. This trend is even more pronounced when considering the systems of the TOP500 list of supercomputers [1]. These systems are expected to feature, in the near future, *fat* many-core nodes composed of one hundred (or more) cores. While increasing the number of cores raises the theoretical peak performance, keeping that many processing units busy requires a significant amount of data to be transferred to and from main memory. The flat memory bus, as featured in many legacy Symmetric Multi Processors (SMP) *north-bridge* chipsets, is not able to sustain such a bandwidth and request throughput, practically limiting the achievable performance to a tiny fraction of the computing peak – a problem known as *the memory wall*. To avoid this problem, most recent multi-core designs embrace Non Uniform Memory Access (NUMA) and hierarchical memory interconnects to enable core count scalability and adequate bandwidth between the cores and the memory banks, but at the expense of an excruciating programming complexity.

Message passing has been the dominant programming model in High Performance Computing (HPC) applications for over a decade. However, MPI applications that are oblivious of the NUMA topologies and the associated performance traps, are bound to suffer from unacceptable performance because they generate a load pattern on the memory subsystem that crashes into the memory wall. While the MPI programming model is expressive enough to enable a mapping between the underlying shared memory topology and the application communication pattern, such an approach requires the modification of every code on every platform, defeating one key feature that has empowered the prevalence of MPI: performance portability. Indeed, application developers have come to expect performance portability not just in point-to-point MPI communications, but in MPI's *collective* communications. Although researchers have been advancing the state of the art in collective communication performance for years, the investigative focus has typically been on the complexity and performance difficulties posed by hierarchical network interconnects. We believe that the collective communication techniques used with hierarchical network interconnects should now also be applied to distributed machines featuring many cores and NUMA architectures.

The usual approach for transporting message payloads between MPI processes across shared memory is based on the copy-in/copy-out algorithm (as illustrated by the SM component in Open MPI [2] and the Nemesis [3] device in MPICH2 [4]). This algorithm uses an extra pre-allocated

shared memory buffer as an exchange zone between processes. Each message is copied to this intermediate zone by the sender process and then copied to the destination buffer by the receiver process. With the rise of multi-core processors, alternative methods of transferring data between processes have emerged in the form of kernel assisted memory copy (such as KNEM [5] and Limic [6]). These approaches utilize one-copy memory transfers, and have proved beneficial in the context of point-to-point communications [7].

As the premise of this work, we have identified three critical issues specific to collective communications between cores in NUMA architectures that prevent fully utilizing the benefits of kernel-assisted copy methods. The first problem arises from the contentions imposed on the root process in one-to-all or all-to-one collective operations. Every process has to wait for the progression of the core hosting the root process in the copy of data to or from the intermediate buffer. This effect actually prevents any potential acceleration from having multiple cores available to undertake multiple copies simultaneously. Second, many algorithms do not take into account temporal locality, which results in cache-ready data being discarded and then reloaded multiple times. More cache pollution, in turn, leads to a plummeting memory bandwidth as more data lines are reclaimed from the slow and contention-prone memory banks. Last, many implementations ignore topological characteristics such as NUMA and network-style processor interconnects. The blind application of the *one size fits all* collective algorithm can lead to unnecessary traffic between sockets, potentially overwhelming some memory links while under-utilizing others. These issues can be tackled by 1) extending the feature set of kernel-assisted memory copy mechanisms to allow the specification of copy direction and granularity, and 2) reworking the collective algorithms themselves to detect and exploit data locality in NUMA architectures.

In this paper, we propose new MPI collective communication algorithms that take advantage of the NUMA memory subsystem to avoid memory contention and to maximize the overall sustained bandwidth. Our approach is based on the KNEM Linux module – a software mechanism that enables direct memory copying between processes. We investigate several different optimizations to collective algorithms that maximize both parallelism and pipelining, all of which are NUMA topology-aware. A key point in the design is that multiple processes can access the same buffer – or different parts of the same buffer – simultaneously, without the need for more than a single memory copy between processes. Moreover, stream direction control enables the collective algorithm to select sender-writing or receiver-reading according to the communication pattern (all-to-one or one-to-all) to avoid the root process bottleneck. Last, our collective algorithms can detect distance between hardware units to build an optimized communication topology that minimizes inter-socket traffic. Each of these approaches are evaluated experimentally on a variety of different hardware setups, exhibiting a better scalability when increasing the number of cores, leading to substantial performance gains on many-core hardware.

The rest of this paper is organized as follows: Section II introduces related work on intra-node collective operations and kernel-assisted memory copy. Section III introduces some key concepts of the KNEM kernel copy framework. Next, Section IV discusses the simultaneous use of KNEM-based collective algorithms with a NUMA topology aware hierarchical layout. Then, Section V describes the linear KNEM collective algorithms: one-to-all (Broadcast and Scatter), all-to-one (Gather), all-to-all (Alltoall and Allgather), and their corresponding implementations in a new collective component for Open MPI. All those algorithms are compared experimentally with state-of-the-art MPI implementations: Open MPI and MPICH2 in Section VI. Finally, Section VII concludes the paper with a discussion of the results.

II. Related Work

The work presented in this paper is the meeting point between two different trends focusing at optimizing MPI communications within shared memory machines. It therefore relates both the optimization of MPI collective operations on shared memory machines and the advances in kernel-assisted process-to-process copies.

Several optimizations have been used to maximize throughput of collective communications on shared memory nodes. Most of them have been based on adopting different communication topologies (linear, chain, split binary tree, binomial tree, etc.), [8] and by enabling parallel treatment of the message through pipelining. Both MPICH2 and Open MPI feature many of those algorithms and select among them with highly tuned and optimized switch-points based on the message size. Consequently, they deliver good performance on SMP nodes, even though all those algorithms rely on the double memory copy shared memory transport device. However, this last aspect is greatly challenged by the multiplication of the number of cores in currently deployed *fat* supercomputer nodes.

One of the most recent of those efforts is due to Richard Graham et al. [9], who proposed a shared memory-based fan-in/fan-out implementation for multi-core MPI collectives, implemented in the Open MPI SM collective component. Their optimization focuses on lightweight synchronization, reducing memory copy times, increasing parallelism by copying messages in a pipeline way, and controlling working set size to fit into caches by building a logical fixed degree tree. This shared memory based method simply takes multi-core/many-core as a SMP system and ignores other architecture characteristics, such as NUMA, memory hierarchy, and core distance. The fixed degree tree is built following the

logical ranks layout, which cannot always reflect architecture characteristics. With more heterogeneity in modern NUMA multi-core designs, it is hard to optimally tune a shared memory based implementation for different platforms.

Open MPI also features another interesting intra-node collective component named *tuned*. The fundamental idea of the tuned collectives is to make available several different algorithms, and use a runtime decision to select the best algorithm according to message size, communicator size, and other parameters [10]. As an example, for a Broadcast in the *tuned* collective component, a binomial algorithm is used to deliver small messages, a split binary tree algorithm is selected for intermediate messages, and large messages are transferred by a pipeline algorithm in which the pipeline size varies with the message and communicator size. However, it is still hard to tune for an unknown platform, even for expert developers. Indeed, there are too many parameters such as pipeline size, thresholds, etc, and any wrong selection might ruin the overall performance of the *tuned* collectives.

These previous approaches are orthogonal works to the proposed ideas of this paper. They try to maximize the throughput of the collective operation by developing new collective topologies, or selecting, among the available algorithms, the most suitable one. For those approaches to reach their full potential, there is a need to cooperate with another approach to alleviate the penalty due to heterogeneous NUMA architectures, a feature that kernel assisted approaches are able to deliver, if used properly. Unlike those collective algorithms, our component is fully aware of the existence of kernel assisted collective, and uses strategies specifically tailored to maximize the resulting benefits.

Several platform-specific efforts offered single-copy large message communication. For instance BIP-SMP implemented such an optimization for Myrinet based clusters [11]. This idea has spread into most vendor specific HPC stacks, such as Myricom MX, Qlogic IPath, and Bull MPI. Some lightweight kernels enabled an even bigger rework of the model on Cray platforms, thanks to the ability of the operating system to make all processes address spaces accessible to any of them. This unusual feature enabled single-copy RMA-based communication (SMARTMAP [12]) which greatly reduces memory copies needed by intra-node message passing, especially for collectives. Recent Linux kernels support the remapping of others' processes memory thanks to the XPMEM module which enables similar optimizations but is restricted to SGI machines.

Lei Chai et al. [6] introduced a kernel module interface called LiMIC. This kernel-based approach can reduce the number of necessary memory copies to one. KNEM [13] is another similar kernel module used in MPICH2 and Open MPI. KNEM offers additional features such as an asynchronous copy model, vectorial buffer support, and copy offload on dedicated hardware. This approach has already proved to be valuable to increase point-to-point bandwidth between processes communicating over shared memory [5]. However, beyond the free performance upgrade offered by using more efficient point-to-point operations [14], hierarchy aware MPI collective components need more control over the underlying memory copy mechanism to reach their full potential. In this paper, we present new collective algorithms that take into account the specificities of kernel assisted memory copies, and require a new feature compared to state of the art kernel assisted copies: directional control of transfers. The details of those novel KNEM features are introduced in the following section.

III. kernel Assisted Memory Copies

Copy-in/copy-out in a shared memory segment is still the most common approach for transferring data between processes on a shared memory platform. The most prominent drawback is the necessity to copy every data twice. Kernel-assisted memory copy alleviates this issue by using system calls to offload the copy to the kernel. Because the kernel has complete access to the memory space of both processes, it can perform the copy from the source buffer in the sender's address space directly to the target buffer in the receiver's address space without the need for an intermediate buffer. KNEM is an example of such a Linux kernel module that enables high-performance, inter-process, one-copy memory copies. It offers support for asynchronous and vector data transfers. KNEM can also offload memory copies to a hardware DMA engine (such as the Intel I/O AT hardware), if available.

It was not long before the above-cited features raised interest for improving intra-node MPI communications. Open MPI v1.5 includes KNEM support in its shared memory point-to-point communications component. MPICH2 v1.1.1 uses KNEM in the DMA LMT to improve large message performance within a single node. The general operating principles of the integration between KNEM and MPI point-to-point messages are the following (more details can be found in [5]): (1) The sender process declares a send buffer to KNEM. The kernel module saves the list of virtual segments contained in the buffer and associates them with a unique cookie. (2) The sender passes the cookie to the process which is interested in this buffer by an out-of-band transfer. (3) The receiver gives this incoming cookie to KNEM along with a receive buffer. (4) The KNEM module copies the data from the send buffer to the receive buffer within the kernel.

The security model of this strategy is similar to System V IPC shared memory segments. Declaring a memory buffer to the KNEM driver makes it available to any other process. However, other memory regions cannot be accessed unless explicitly declared to KNEM as well. A malicious user modifying a cookie value would either get an invalid parameter error or get a valid access to a previously properly declared buffer.

A. Issues with MPI Collective Operations

While the beneficial effect of KNEM on point-to-point performance also translates into collective improvements [14], we have identified a series of additional optimizations that further boost collective communication performance. They require that the collective component has more control over the movement of data (vs. simply using MPI point-to-point primitives): (1 Because control of the kernel module is delegated to the point-to-point MPI message passing engine, using inter-process kernel-assist memory copies results in the same data region being registered multiple times when sent to different destination processes. The overhead of synchronizing and exchanging cookies therefore cannot be amortized. The collective component knows when the same buffer is used with multiple recipients, and can therefore eliminate redundant registrations. (2 Many collective algorithms exhibit a one-to-all communication pattern. Such a communication pattern stresses the root process resources. Although in kernel execution space when using KNEM, only the core hosting the root process is performing all the data movements. This actually serializes all memory copies and limits the progression of the collective to the speed of only a single core (i.e., where the root process is executing), even though multiple cores are available (and probably waiting for the collective operation to progress). Attempting to alleviate this issue by inverting the point-to-point algorithm – i.e., having the receiver(s) make the copies – simply results in degrading all-to-one communication patterns, instead. Thus, it is desirable for the collective algorithm to be able to express the direction of data transfer, independently of concurrent point-to-point strategies.

B. KNEM Direction and Granularity Control

Implementing collective operations directly on top of the original KNEM interface wastes system resources. Collective patterns such as one-to-all or all-to-one would declare the same multiply-accessed memory buffer multiple times and pass multiple KNEM cookies between processes for synchronization. To solve this issue, we introduced, (available since KNEM 0.7), an extended programming interface designed to address the needs of collective operations. Instead of only offering a point-to-point send-receive interface, KNEM now offers the ability to register persistent memory regions and access them multiple times from different processes. Such accesses may either touch all or only part of the region, enabling the actual copy a single message at once or as multiple chunks simultaneously. This model avoids wasting system resources and reduces the overhead of synchronizing processes when creating and passing region identifiers.

Another extension added to KNEM is the ability to read (receiver-reading) or write (sender-writing) to each region, enabling effective direction control of the data transfer. Direction control is a very important feature to unleash the performance of collective operations. The effective direction control of the copies can be decided by the collective component to match the communication pattern (one-to-all or all-to-one), with the goal of maximizing the number of cores participating to the progression of the data transfers, and parallelize the progression of the collective algorithm as much as possible.

Those two novel features introduced into the KNEM kernel module are used by our KNEM collective component in Open MPI. Unlike previous components that only used kernel-assisted copies simply through the point-to-point interface, this component takes advantage of directional control and persistent registrations to further increase the performance achieved on collective communications.

IV. Topology and NUMA aware collectives

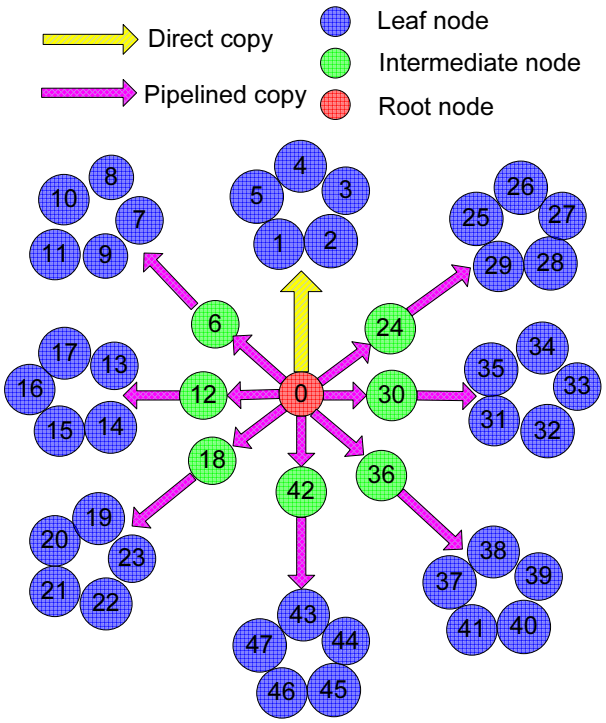


Figure 1. Progression of the hierarchical pipeline KNEM Broadcast

A significant amount of work has been done over the past decade toward improving the collective communication performance by taking into account the network features. Some algorithms take advantage of the low latency of some specific high-performance, while others capitalize on the bandwidth capabilities. Additionally, the network topology (butterfly, torus) has been another important factor in redesigning collective communications algorithms. While this work is significant, and has clearly influenced our approach, one has to keep in mind the tight conditions required for collective algorithms in shared memory. The memory access

latency and bandwidth are important, but not more important than the topology of the links to and from the memory banks. Therefore, the algorithms designed in this context are significantly more complex than the usual two-level hierarchical collective algorithms [15], and take into account the topology as well as the memory accesses performance, that usually exhibit several order of magnitude performance differences for different levels in the hierarchy.

In order to explain how the collective algorithm will be affected by the hardware architectural features, let us take our large NUMA node (IG) as an example. This machine consists of 8 NUMA nodes, each of them containing a six-core AMD Opteron processor and 16GB of local memory. 8 NUMA nodes are interconnected by AMD HyperTransport (HT). The Figure 1 shows how a 48 processes broadcast will unfold on this large NUMA node. Processes are split into 8 sets according to their NUMA locality information, which means processes within the same socket and NUMA node are in the same set. A two levels' tree is then built accordingly, one process per NUMA node will belong to the first tree level (the green background circles), while all the remaining processes per NUMA node will belong to the second level (the blue background circles), and behave as leafs to the tree. This tree structure reflects the architecture topology such as core distances and relationships which can be retrieved thanks to the *Hardware Locality* [16] software.

Dividing the processes based on the topology and NUMA information has the advantage of reducing inter-socket traffic since a single data transfer is performed towards each NUMA node. Moreover, since a cache is shared between all processes inside the same NUMA node, multiple copies between processes in the same set benefit from cache hits. However, one of the disadvantages of such an algorithm is the reduction in the degree of parallelism between the memory copies toward NUMA nodes, and between the leaf nodes and their corresponding intermediary node (a leaf process cannot start a memory copy until the intermediary process received the entire data). To alleviate this strong synchronization requirement, and therefore minimize the unnecessary waiting on the leaf nodes, we divide the data in several segments and pipeline the operations corresponding to each one of these segments (pink arrows).

V. Implementation

A. Framework

We implemented the KNEM based collectives as a new component named *KNEM coll* in Open MPI. Open MPI is based on a flexible component architecture, collective algorithms are placed under the *COLL* framework (as depicted in Figure 2). Multiple collective components are available (*Tuned*, *Basic*, *SM*, and our proposed *KNEM coll*), and can be selected at runtime. In Open MPI's component architecture, different collective components can use different point-to-point communication components underneath (the

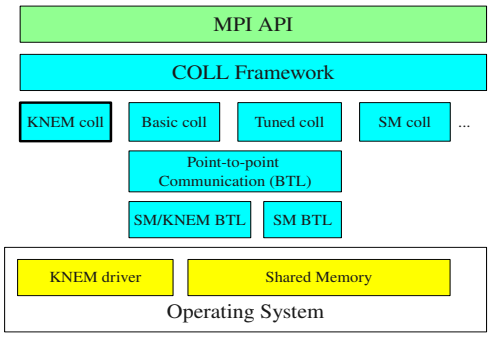


Figure 2. Open MPI collective communication framework

byte transfer layer, BTL) [17]. By default, the Tuned collective component and the SM BTL component are teamed to provide for MPI collective operations, resulting in the default setup to use the best collective algorithms with the copy-in/copy-out point-to-point transport. In previous works the *SM/KNEM BTL* have been introduced. This BTL enhances shared-memory point-to-point operations by using the KNEM driver for large messages. While this is not a default setup, the Tuned collective component can be teamed with this BTL to benefit from KNEM speedup to some extent for large messages.

Unlike other collective components, the KNEM collective component uses the shared memory BTL only as an out of band channel for synchronization or delivering "cookie". All data movements are directly handled inside the collective component by resorting to direct calls to the KNEM kernel driver. While the KNEM kernel module is used directly, the collective algorithms themselves remain implemented in user-space. Direct calls to KNEM within the collective component are required to have enough flexibility to express the new algorithms intended to avoid unnecessary buffer registrations, handle directional copies, and fragment the messages to establish a pipeline suitable for the most complex memory hierarchies. The KNEM RMA API that we introduced in KNEM (see Section III) is used to create memory regions dynamically and read or write all or part of them multiple times whenever needed. However, trapping into kernel mode has a non-negligible overhead (about 100 ns on modern processors) when delivering small messages. So we only consider KNEM for optimizing collectives for intermediate and large messages (larger than 16KBytes). We started by adapting the most useful algorithms first (Gather(v), Scatter(v), AlltoAll(v), AllGather(v) and Broadcast, v stands for vector variants of corresponding operations). For smaller messages, or unimplemented collective calls, the operation is delegated to the regular Open MPI component.

B. Rooted Operations: Broadcast, Scatter and Gather

The implementation of KNEM Broadcast is a straightforward adaptation of the KNEM point-to-point model: 1)

The root process declares a send buffer to KNEM and gets the corresponding cookie in return. 2) It passes the cookie to all non-root processes in the communicator through an out-of-band transfer (Open MPI SM BTL). 3) Each receiver process passes the incoming cookie to KNEM along with a receive buffer. 4) KNEM triggers a memory copy from the send buffer to receive buffer within the kernel. The copy is performed by each receiver core in parallel. 5) Each receiver process sends back a synchronization message to root process after the completetion of KNEM copy. 6) After the root process receives all synchronization messages from non-root processes, it deregisters the send buffer from the KNEM driver. The KNEM Broadcast also features a hierarchical pipelining algorithm (as sketched in Section IV), that can be turned on or off depending on the properties of the hardware. The topology mapping is static for now, but will be dynamic in future works.

The KNEM Scatter implementation is overall similar to the broadcast, except that each non-root process reads only parts of the root buffer. Their starting offset is calculated from their ranks and the data type.

The KNEM Gather consists of the opposite communication pattern of Scatter. Therefore it can benefit from the direction control of the new KNEM version to declare the root process memory as a write buffer. Unlike the regular Gather, all non-root processes can write simultaneously to that buffer.

C. AllGather and Alltoall

The KNEM AllGather is an assembly of a gather followed by a broadcast. During the first step, all processes do a KNEM Gather operation to rank 0. In the second stage, the root process performs a KNEM Broadcast. This method is far from being optimal, because it puts the memory controller of the the core hosting the root process under a high pressure. Although it could be perfected, it is a simple and straightforward way to capitalize on the improvements achieved in Broadcast and Gather.

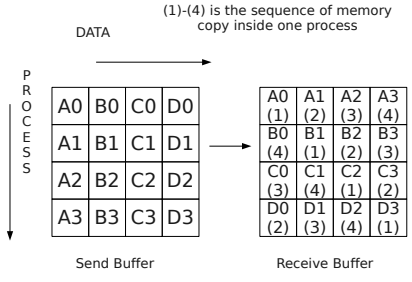


Figure 3. An example of copy sequence for KNEM AlltoAll on four processors.

Unlike one-to-all operations, in the AlltoAll communication pattern, cache reuse plays little to no role. Therefore, our AlltoAll algorithm focuses on avoiding bandwidth sharing, by rotating the access pattern so that at any instant, a core is sending and receiving exactly one fragment of data. First, each process declares its send buffer to KNEM and gets back the corresponding cookie. Those cookies are exchanged by doing an out of band AllGather operation based on shared memory(not KNEM AllGather); it is necessary to pre-allocate an integer array the size of the communicator to store cookies from all other processes. A loop of KNEM copies is performed to fetch the corresponding messages from other nodes (receiver-reading). Each process offsets the starting point of this loop in a round-robin manner. Finally, each process deregisters memory buffer from the KNEM driver after a barrier operation. Figure 3 gives an example of this communication scheduling; as one can see, for every step of the algorithm (marked between parenthesis on the right buffer), the memory belonging to a particular sender is accessed only once, and the workload is evenly spread on the entire duration of the collective.

VI. Experimental Evaluation

A. Experimental Conditions

Our experimental platform is composed of four multi-core/many-core machines that cover the spectrum of typical current commodity high-performance computing nodes, but also expected designs for the upcoming years. They feature Intel and AMD processors, and represent a wide variety of setups, from SMP to massively NUMA machines. As our approach does not impact the performance of inter-node communications, our experiment focuses on intra-node communications performance.

Zoot is a 16 core machine with 32GB of memory. The system has four sockets with a quad-core 2.40 GHz Intel Xeon Tigerton E7340 featuring 4 MB L2 caches shared between pairs of cores. A single SMP memory controller in the north-bridge chipset connects all the sockets with the global shared memory.

Dancer is an 8 core machine with 4GB of memory. The system is composed of two sockets populated with a quad-core 2.27 GHz Intel Xeon Nehalem-EP E5520 with 8 MB L3 caches and 2 GB of memory on each NUMA socket. Hyper-threading is disabled in the configuration.

IG is a 48 cores machine with 128GB of memory. The system is composed of 8 sockets with a six-core 2.8 GHz AMD Opteron 8439 SE, 5 MB L3 caches and 16 GB memory per NUMA node. The sockets are further divided as two sets of 4 sockets on two separate boards connected by a low performance interlink.

Saturn is a 16 core machine with 64GB of memory. It is composed of two sockets with an octo-core 2.00 GHz Intel Xeon Nehalem-EX X7550, 18 MB L3 caches and 32 GB memory on each NUMA socket. Hyper-threading is enabled but not used.

Software setup includes KNEM version 0.9.2 [13]. The Intel MPI benchmark suite IMB-3.2 was used to assert the difference between the collective components with the *off-cache* option in enabled to avoid cache reuse. Open MPI version 1.7a1 and MPICH2-1.3.1, both properly tuned, are compared to our approach. Tuning parameters between all components based on Open MPI are identical, unless stated. On a particular machine, the mapping between physical cores and MPI processes is identical, regardless of the MPI implementation used. Due to the large number of combinations, we restrict the discussion only to the most meaningful algorithms, but still covering the entire spectrum of collective patterns.

Because our own component (referred to as KNEM-Coll from now on) is inside Open MPI, we undergo a deeper comparison with its default collective component: the *Tuned* component. It does not use kernel assisted copies by default, but uses the SM device that relies on copy-in/copy-out; results obtained with this setup are called *Tuned-SM*. To assert clearly what are the extra benefits of our approach, we also present the results obtained when simply using the Tuned component on top of KNEM point-to-point messages (*Tuned-KNEM*). *MPICH2-SM* and *MPICH2-KNEM* are similar to *Tuned-SM* and *Tuned-KNEM*, with respectively shared-memory or KNEM as the underneath point-to-point communication transport. The MPICH2 KNEM implementation is broken for messages of size beyond 64 KB on the Saturn platform; we could not obtain a revised version from the author at the date of publication of this paper.

To ease the comparison, our KNEM collective component is taken as the reference point for performance, and the execution time of other implementations are normalized against it. The smaller these normalized values, the better the performance of the corresponding collective component.

B. Hierarchical effect and tuning the pipeline size

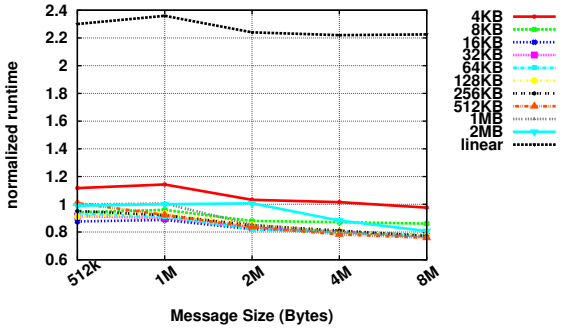


Figure 4. Performance comparison between linear KNEM Broadcast, hierarchical KNEM without pipeline, and different pipeline sizes in the hierarchical pipelined KNEM Broadcast on the IG platform. Results are normalized to the runtime of hierarchical KNEM Broadcast without pipeline (lower is better).

Figure 4 presents the effect of the pipeline size on the hierarchical pipelined KNEM Broadcast on the large NUMA machine IG. The pipeline sizes range from 4KB to 2MB. The execution time of different pipeline sizes is normalized to the execution time of hierarchical KNEM Broadcast without pipeline. Compared with the linear KNEM broadcast, the hierarchical approach itself contributes a 2.2× to 2.4× speedup on this large NUMA node. And the pipelining provides hierarchical KNEM Broadcast extra up to 1.25× speedup. One can see that except using too small pipeline size (4KB), hierarchical KNEM Broadcast always can get significant benefits from pipelining.

Selecting a suitable pipeline size can be a challenging problem for the hierarchical pipelined strategies. A too small pipeline size induces more synchronization between each segment and makes the transfer efficiency suffer due to KNEM copy startup overhead when delivering small messages (4KB and 8KB in Fig 4), while a too large pipeline size (2MB in Fig 4) leads to a long initialization time for the pipeline algorithm to take effect. One can see that the best pipeline size is 16KB for intermediate message size (smaller than 2MB), and 512KB for large message sizes. In the rest of this paper, we settled the pipeline size according to this tuning on IG: 16KB for intermediate message size and 512KB for large message size.

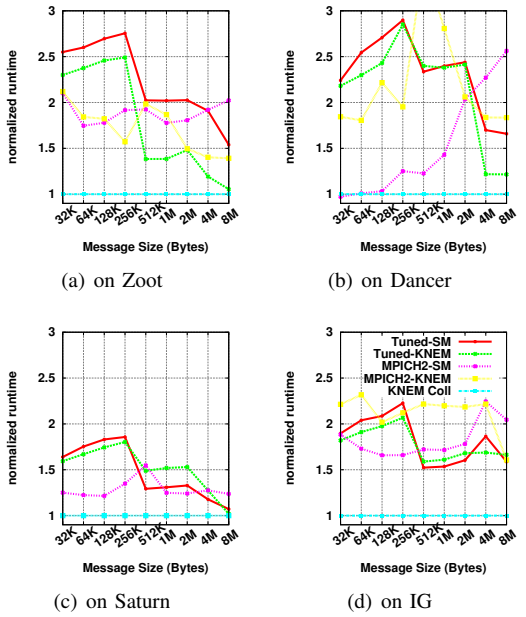


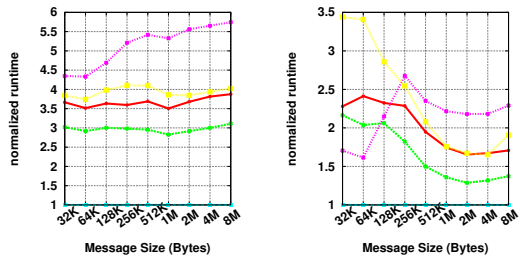
Figure 5. Performance comparison of Broadcast operations between Open MPI Tuned components, MPICH2, and the KNEM collective component. Results are normalized to the runtime of the KNEM Bcast (lower is better).

C. One-to-all and All-to-One Operations

Figure 5 shows the performance comparison of the Broadcast implementations on all platforms. The KNEM Broadcast outperforms other collective components in all cases. Compared with Open MPI's best collective component, the

KNEM Broadcast can provide a speedup of about 1-2.5× on Zoot, 1.2-2.8× on Dancer, 1-1.8× on Saturn and 1.5-2.1× on IG.

Figure 6 presents the performance comparison of the Gather operation. The linear KNEM Gather tremendously outperforms all other components in all cases. Compared with the best Gather in Open MPI and MPICH2, the maximum speedup, thanks to KNEM collective component, is 3.1× on Zoot, 2.2× on Dancer, 2.6× on Saturn, and 3.2× on IG.



(a) on Zoot (b) on Dancer

(c) on Saturn (d) on IG

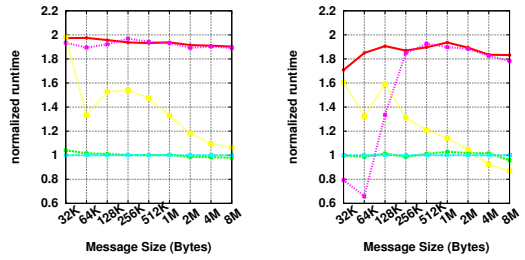
Figure 6. Performance comparison of Gather operations between Open MPI Tuned components, MPICH2 and the KNEM collective component. Results are normalized to the runtime of the KNEM Gather (lower is better).

KNEM Gather and Scatter operations are very similar, except from the different copy direction: sender-writing for Gather and receiver-reading for Scatter. Consequently, those two algorithms exhibit very similar performance profiles and the Scatter results are not presented here. Compared with Open MPI's best Tuned Scatter implementation, the maximum speedup of KNEM Scatter is about 3× on Zoot, 2× on Dancer, 4× on Saturn, and 4× on IG.

The KNEM collective component has a huge performance speedup in these "rooted" collective operations, thanks to unleashing parallel access to the buffer of the root process. Compared with the approach of indirectly using KNEM copy as underneath communication (Tuned-KNEM), the KNEM collective component can provide more reliable improvement in all cases, benefiting from KNEM drivers' new features we mentioned at section III.

D. All-to-all Operations

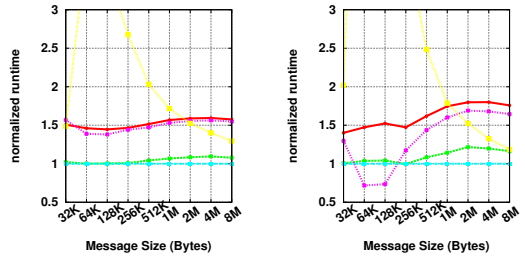
Figure 7 shows the performance comparison for the AlltoAllv collective operation. The performance of these



(a) on Zoot (b) on Dancer

(c) on Saturn (d) on IG

Figure 7. Performance comparison of AlltoAllv operations between Open MPI Tuned components, MPICH2 and the KNEM collective component. Results are normalized to the runtime of the KNEM AlltoAllv (lower is better).



(a) on Zoot (b) on Dancer

(c) on Saturn (d) on IG

Figure 8. Performance comparison of AllGather operations between Open MPI Tuned components, MPICH2 and the KNEM collective component. Results are normalized to the runtime of the KNEM AllGather (lower is better).

all-to-all collective operations is restricted by the press over memory buses due to delivering too many messages simultaneously. As a consequence, the relative performance benefits are smaller when compared to Tuned-KNEM than for the one-to-all or all-to-one rooted operations presented

in the previous paragraphs. However, compared with Tuned-SM based on the shared memory approach, the KNEM AlltoAllv can still show significant improvement, with a maximum speedup of 2× on Zoot, 1.9× on Dancer, 1.25× on Saturn, and 2.7× on IG.

Finally, Figure 8 presents the performance of the collective components on the AllGather operation. On the SMP machine (Zoot), and the medium size NUMA machines (Dancer, Saturn), the KNEM AllGather has the best performance among all collective components except some medium size messages on Dancer and Saturn. On the large NUMA node (IG), Tuned-KNEM performs better than KNEM AllGather by up to 25%. The loss of KNEM AllGather's performance on large NUMA nodes lies in the KNEM AllGather operation not being optimized as explained in Section V-C. The operation is split into two separated stages. Although the KNEM Gather and Broadcast are optimized in each stage, overlapping between these two stages is eliminated by this simple concatenation of the KNEM Gather and Broadcast algorithms. And the selected root process forms a single point in KNEM AllGather implementation, forcing the throughput of AllGather operation restricted by the limited memory bandwidth of the NUMA node owning the root process. This is also the reason why the KNEM AllGather's performance suffers more on the large NUMA nodes than on SMP or small NUMA nodes. However, even on large NUMA node IG, the KNEM AllGather still performs better than Tuned-SM and MPICH2-SM, which are based on shared memory approach. The implementation of these two all-to-all collective operations (e.g. AlltoAllv and AllGather) benefits greatly from the adoption of KNEM copy, thanks to reducing memory copies and cache pollution in these communication-intensive operations. In the next release of the KNEM collective component, we will borrow some ideas in Tuned collective components such as adopting a ring-style algorithm to distribute memory accesses evenly across memory controllers in the KNEM AllGather, especially on large NUMA nodes.

E. Application Performance

To evaluate the impact of the improvement due to using KNEM collective operations on real application performance, we use the ASP [18] application, a parallel implementation of the Floyd-Warshall algorithm used to solve the all-pairs-shortest-path problem. The main MPI collective operation used in this application is MPI_Bcast. We tested this application on two machines: Zoot and IG, the two extreme platforms regarding the degree of complexity of the core hierarchy. The problem is scaled to match the available memory; the matrix size is 16384^2 on Zoot and 32768^2 on IG (32bits integers). Matrices are distributed by rows across all the available cores. The MPI_Bcast operation is called 16384 times (64 KB message) on Zoot and 32768 times (128 KB message) on IG. The KNEM collective component uses the linear KNEM algorithm on Zoot and the hierarchical pipelined algorithm on IG. All tests use the same mapping between cores and processes.

Table I
ASP APPLICATION EXECUTION TIME BREAKDOWN AND SPEEDUP FROM USING KNEM COLLECTIVES.

	Zoot		IG	
	Bcast	Total	Bcast	Total
Open MPI	405.7s	2891.2s	550.2s	6650.9s
MPICH2	152.3	2640.4s	293.9s	6413.8s
KNEM Coll	26.8s	2508.4s	198s	6288.1s
Improvement	82.4%	5.2%	33%	2%

Table I presents the application execution time of ASP when using different collective components. The improvement is the relative difference between the best performing MPI library (between Open MPI and MPICH2) and our KNEM collective components. By using the KNEM collective components, the application can see a significant improvement in the time it spends doing Broadcast operations, with the improvement of 82% on Zoot and 33% on IG. One can notice that the performance improvement of the Broadcast only on the SMP machine is even more pronounced than for the synthetic benchmark, because unlike the synthetic benchmark, the application does not systematically invalidate the cache before performing the operation. As a consequence of the shorter time spent in the collective operations, the overall application runtime is improved when compared to other optimized collective components, with an improvement of 5% on Zoot and 2% on IG.

VII. CONCLUSION

In this paper, we showed that an MPI implementation can successfully take advantage of new features in the kernel, namely kernel assisted memory copies between process spaces, to greatly improve the performance of shared memory collective communication while successfully hiding the complexity of the NUMA characteristics of modern many-core nodes. Three challenges specific to many-core systems have been addressed to meet that goal: NUMA complexity, cache pollution, and collective algorithm sequentialization.

The main contributions of this paper are: (1) extend kernel-assisted memory copy operations to include direction and granularity control, thereby giving more flexibility and data movement control to the collective component implementation, and (2) embed the use of kernel-assisted copies directly into collective algorithms rather than relying on kernel-assisted speedups from MPI point-to-point-based primitives. (3) Our collective component also maps the inherent parallelism of MPI collective communications on to the specific characteristics of NUMA multi-core systems. For example, the root process can offload memory copies to non-root processes in order to parallelize operations

with multiple receivers. Finally, the pipelining algorithm strategically overlaps the latency incurred by transferring first to the upper levels of the NUMA hierarchies with data transfers to the leaf nodes in the collective topology. Experimental results show that our approach outperforms all other types of optimizations on many-core systems, including the Tuned Open MPI component (even though Tuned benefits from KNEM-enabled point-to-point operations). These performance improvements translate into a substantial gain for our showcase application.

REFERENCES

[1] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon, "Top500 supercomputing sites (2010)," http://top500.org.

[2] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Sep. 2004, pp. 97–104.

[3] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem," *Cluster Computing and the Grid, 2006. Sixth IEEE International Symposium on*, vol. 1, pp. 10–20, May 2006.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[5] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," in *Proceedings of International Conference on Parallel Processing*, Sep. 2009, pp. 462–469.

[6] H.-W. Jin, S. Sur, L. Chai, and D. Panda, "LiMIC: support for high-performance MPI intra-node communication on linux cluster," *International Conference on Parallel Processing*, pp. 184–191, June 2005.

[7] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and topology aware intra-node communication among multicore CPUs," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10, 2010, pp. 265–274.

[8] L. Huse, "Collective communication on dedicated clusters of workstations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1999, vol. 1697, pp. 469–476.

[9] R. Graham and G. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5205, pp. 130–140.

[10] G. E. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, and J. Dongarra, "Tuned: A flexible high performance collective communication component developed for Open MPI," in *Proccedings of DAPSYS'06*. Innsbruck, Austria: Springer-Verlag, Sep. 2006, pp. 65–72.

[11] P. Geoffray, L. Prylli, and 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, Nov. 1999.

[12] R. Brightwell, K. Pedretti, and T. Hudson, "Smartmap: operating system support for efficient data sharing among processes on a multi-core processor," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 25:1–25:12.

[13] "KNEM: High-Performance Intra-Node MPI Communication," http://runtime.bordeaux.inria.fr/knem/.

[14] S. Moreaud, B. Goglin, D. Goodell, and 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*, Apr. 2010.

[15] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, vol. 34, no. 8, pp. 131–140, May 1999.

[16] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "HWLOC: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[17] J. M. Squyres and A. Lumsdaine, "The component architecture of Open MPI: Enabling Third-party Collective Algorithms," in *In Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, 2004, pp. 167–185.

[18] A. Plaat, H. E. Bal, R. F. H. Hofman, and T. Kielmann, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769–782, 2001.