Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis

Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, Stéphanie Moreaud

Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis

Darius Buntinas,[1] Brice Goglin,[2] David Goodell,[1] Guillaume Mercier,[2] Stéphanie Moreaud[2]

[1] Mathematics and Computer Science Division – Argonne National Laboratory – Argonne, IL 60439, USA
[2] INRIA, LaBRI, Université of Bordeaux – Talence, France
{buntinas,goodell}@mcs.anl.gov,{goglin,mercier,moreaud}@labri.fr

Abstract

The emergence of multicore processors raises the need to efficiently transfer large amounts of data between local processes. MPICH2 is a highly portable MPI implementation whose large-message communication schemes suffer from high CPU utilization and cache pollution because of the use of a double-buffering strategy, common to many MPI implementations. We introduce two strategies offering a kernel-assisted, single-copy model with support for noncontiguous and asynchronous transfers. The first one uses the now widely available vmsplice Linux system call; the second one further improves performance thanks to a custom kernel module called KNEM. The latter also offers I/OAT copy offload, which is dynamically enabled depending on both hardware cache characteristics and message size. These new solutions outperform the standard transfer method in the MPICH2 implementation when no cache is shared between the processing cores or when very large messages are being transferred. Collective communication operations show a dramatic improvement, and the IS NAS parallel benchmark shows a 25% speedup and better cache efficiency.

1   Introduction

Multicore architectures are ubiquitous today in high-performance computing, with new architectures promising increasing core counts and a more complex organization of memory resources. Despite the lack of consensus on what the most adequate programing model for massively multi-core architectures should be, MPI will continue to play a major role in parallel application development. As a consequence, improving MPI implementations performance in the intranode communication case using shared memory is more than ever relevant. This issue has fueled research in many existing projects, such as [6] or [9]. Communication libraries (such as MPI) must be able to take full advantage of the features offered by new, complex multicore archi-

tectures. For instance, processor cores located on the same node will share not only physical main memory but also part of the cache hierarchy.

MPI implementations will typically use different transfer mechanisms for communicating small messages, where low latency is important, versus large messages, where high bandwidth is important. We previously examined optimizing small-message communication in [5]. In this paper we focus on intranode large-message communication. In [4] we evaluated several methods for large-message communication over shared memory, including double-buffered memory copies and kernel module support. Here we expand on that work by presenting KNEM, a new LINUX kernel data transfer module, and analyzing its various operating modes such as asynchronous vs. synchronous or hardware-supported vs. kernel-thread memory copies. We also examine the recent LINUX system call vmsplice, which is designed to allow for efficient data transfer between processes. We evaluate all these mechanisms with respect to throughput and cache pollution.

The rest of this paper is organized as follows. In Section 2 we provide background information about large message transfers in MPICH2-NEMESIS. Section 3 explains two approaches to implementing single-copy intranode transfers (vmsplice and KNEM). Experiments shown in Section 4 emphasize the performance advantages of the new methods we designed. We discuss related work in this area and propose future research directions in Section 5 and Section 6, respectively.

2   Transferring Large Messages with MPICH2-NEMESIS

MPICH2 is a widely portable, high-performance implementation of version 2.1 of the Message Passing Interface (MPI) standard [13]. The next release (1.1) of MPICH2 will use a communication subsystem called NEMESIS [6], which utilizes shared memory for intranode communication and networks for internode communication. Shared-memory communication in NEMESIS is highly optimized

1

to minimize the latency of small messages and maximize the bandwidth of large messages. One of the optimizations for large-message communication is the introduction of an internal API called the *Large Message Transfer* (LMT) interface, which is designed to be general enough to support various mechanisms for transferring large messages. In particular, the LMT interface supports both one-sided and two-sided data transfer mechanisms, such as using RDMA put or get operations (one-sided), copying data to and from a UNIX pipe (two-sided), and copying data through a memory-mapped shared-memory buffer (two-sided). This flexibility allows NEMESIS to use the best transfer mechanism available on each platform for each specific transfer.

Prior to the work described in this paper NEMESIS used a double-buffering scheme to transfer large messages, as described in [4]. This method always results in two copies, one from the source buffer into the copy buffer and another out of the copy buffer into the destination buffer. This number of copies cannot be reduced by using the double-buffering approach. But if two processors are participating in the transfer, the copies might overlap to some degree, one thereby partially hiding the cost of the other. However, this method requires both processors to actively take part in the transfer, which prevents them from performing useful application computation. This method also pollutes the cache by evicting application data from it as the copy operation is being performed [4].

Optimally, the data would be transferred by using a single copy. In LINUX and other UNIX environments, however, a process cannot directly access the address space of another process. In order to achieve a single-copy transfer, the kernel must be involved. The two methods described in this paper are to use either a LINUX kernel module called KNEM or the `vmsplice` system call. KNEM offers very good performance with the option of hardware-supported data transfer and asynchronous operation. However, deploying such a nonstandard kernel module on a system requires administrative privileges, which most users are unlikely to possess. Using `vmsplice` has the benefit of being standard on newer LINUX operating systems; however, it does not perform as well as KNEM and is unavailable on non-LINUX systems. The double-buffered copying mechanism is more portable than KNEM and `vmsplice` (it is available on any system that supports the `mmap` system call or System V shared memory), but it does not perform as well as the others in all circumstances.

Not all of the methods will be available on all platforms, and some methods will perform better than others depending on several factors such as message size or sender/receiver processes locations on the various cores. By using the LMT interface, however, NEMESIS is able to choose the most adequate data transfer mechanism for each situation.

3 Design and Implementation of Single-Copy Communication Mechanisms

In this section we describe the implementation of two new single-memory-copy LMT backends using `vmsplice` and KNEM, respectively.

3.1 The vmsplice Large Message Transfer

LINUX kernel 2.6.17 introduced a set of new system calls to enable the movement of large amounts of data through sockets, pipes, and files with a minimum of user-space-to-kernel-space and kernel-space-to-user-space copies. The `vmsplice` system call is one of these [2]. To use the `vmsplice` system call, the sending and receiving processes open the same UNIX pipe. Then, instead of calling the `writev` system call, the sending process calls `vmsplice`, which takes similar arguments. The `vmsplice` system call attaches virtual memory pages directly into a kernel pipe buffer instead of copying the corresponding data from user space to kernel space. When the receiver calls the `readv` system call, the data is then directly copied from these pages to the destination buffer. In this way the data is transferred using a single memory copy.

We implemented a new LMT backend in NEMESIS using `vmsplice` in this manner. By default the LINUX kernel has a compile-time limitation of 16^1 pages per pipe (4 KiB/page), for a total limit of 64 KiB transfered per call to `vmsplice` or `readv`. At first glance this limitation seems cumbersome; however, in practice it actually improves NEMESIS responsiveness by allowing NEMESIS to periodically poll for new messages between chunks. There is a small overhead due to calling the `vmsplice` system call several times to transfer a single large message. However the cost of the system call on a modern processor (about 100 ns on an INTEL XEON) is much lower than copying a single 64 KiB chunk (approximately $8 \mu s$ with 8 GiB/s memory bandwidth), making this an acceptable trade-off in order to maintain responsiveness.

3.2 KNEM: Dedicated LMT in the Linux Kernel

The `vmsplice` LMT has the advantage of working on all recent LINUX systems. However, it supports only a blocking and synchronous interface on the receiver side and cannot benefit from hardware-supported copy offload. To further study the idea of replacing the original NEMESIS two-copy model with a kernel-based, single-copy strategy, we developed a dedicated kernel driver and LMT backend. We previously had experimented with single-copy transfers within OPEN-MX, an ETHERNET-specific message-passing stack [8]. We reworked this idea to fit in the context of a general-purpose MPI implementation.

[1] controlled by the `PIPE_BUFFERS` definition in `pipe_fs_i.h`
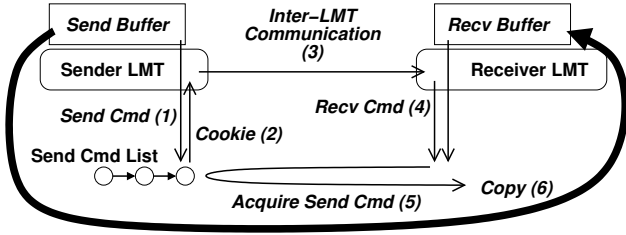
Figure 1. Large-message transfer with the KNEM custom kernel module.

KNEM stands for *Kernel-Nemesis*. It relies on a pseudo-character device that implements two main communication commands as depicted in Figure 1. A sender process (1) declares a send buffer to KNEM (*send* command). The driver saves the list of virtual segments contained in this buffer and associates them with a unique *cookie* so that other processes may directly read from them in the future. A cookie id is given back to the sender process (2) and then sent (3) to the receiver. The receiver passes (4) a receive buffer along with the send *cookie* id to KNEM (*receive* command). The KNEM driver takes care of moving data (5) from one buffer to another within the kernel thanks to the sender's virtual segments having been stored in the cookie earlier. The new KNEM LMT backend in NEMESIS uses these commands and passes the cookie from sender to receiver through the usual NEMESIS user-space *rendezvous* handshake.

3.3 DMA Engine Backend in KNEM

I/OAT is a set of hardware features implemented in modern INTEL memory controllers [10]. It specifically contains a dedicated device, called a *DMA engine*, that can be used to perform efficient memory copies in the background, allowing the processor to perform other useful work. Also, because the processor does not perform the memory copies itself, the processor's caches are not polluted. We added the ability for KNEM to use I/OAT to offload the copying and synchronously poll for completion before returning to userspace. This feature can be enabled through a parameter to the KNEM receive command.

Normally, there is no guarantee that any virtual address (which is used by applications) will always map to the same underlying physical address (which is used by any hardware such as I/OAT). Indeed, the virtual memory management of an operating system may, for instance, swap pages to the disk in case of memory shortage. This is a problem if a page is swapped out while the I/OAT hardware is performing a copy. In order to ensure that the mapping will not change, the KNEM kernel module pins application buffers down in physical memory before passing them to the I/OAT hardware. In fact, because the receiver process cannot easily access the sender address space in LINUX, the send buffer is pinned even when I/OAT is not used. So, the send KNEM
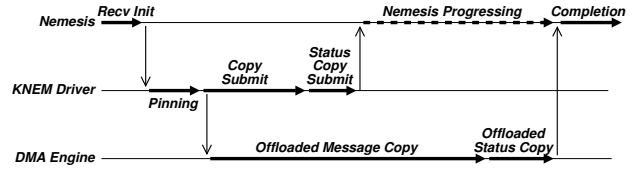
Figure 2. Asynchronous large-message transfer with KNEM and I/OAT copy offload.

command will always pin the sender buffer, while the receive command pins the receiver buffer only when I/OAT is used.

3.4 Asynchronous KNEM Model

The I/OAT *DMA Engine* hardware frees the host processors while the copy is performed in the background, thereby opening an opportunity to overlap the copy with useful computation. Moreover, the user space NEMESIS implementation expects to be able to poll for incoming messages periodically. We thus designed an asynchronous KNEM implementation that lets the receiver process return to userspace while the copy is performed in the background.

To do so, we removed the blocking completion waiting in KNEM. Now, in order to notify the library that the copy has completed, the library passes an address to a status variable to KNEM. When the copy operation has completed, KNEM writes *Success* in the status variable. The library then has to poll the variable to check for completion.

This model is difficult to implement because the I/OAT hardware interface is polling based. Since no interrupt is raised on copy completion, having the kernel driver change the status requires that it first polls for I/OAT completion. Instead of using this CPU consuming solution, we decided to benefit from the fact that I/OAT processes copy requests in order. After submitting a large message copy, we submit a single-byte copy that writes *Success* in the status variable as depicted in Figure 2. In this way, not only the copy but also its completion notification are performed in the background, enabling full overlap.

We also designed an asynchronous model for non-I/OAT transfers by offloading the memory copy to a kernel thread that runs on the receive process core. This model enables overlap of the memory copies but it implies that the user-space NEMESIS process and the kernel thread will compete for the same core. While we expect this to slightly reduce the performance of the copy, it does allow the copying to proceed asynchronously with user-space progression or computation.

3.5 Deciding Which LMT to Use

NEMESIS can now benefit from two new LMT methods, with and without I/OAT. Of the two methods, KNEM

is expected to provide the best performance because it has been designed for MPI data transfer whereas `vmsplice` is a general-purpose solution. Users thus should use KNEM if loading a custom kernel module is acceptable. In situations where this is not acceptable, the `vmsplice` LMT should be used, if possible, because it works on all modern LINUX kernels and will improve performance by reducing the number of memory copies.

Using the KNEM LMT backend raises the question of when to switch from a regular copy to an I/OAT offloaded copy. While looking at early performance numbers on 2.33 GHz XEON processors with a 4 MiB L2 cache shared between 2 cores, we observed that KNEM should offload copies to I/OAT hardware when the size passes 1 MiB. We ran the same test between 2 cores not sharing a cache and observed that the threshold jumps to 2 MiB. Running the experiment on another host with 6 MiB L2 caches increased the threshold by 50%.

These results led us to correlate the largest cache size (L2 here) and number of processes using it with the observed threshold. Indeed, a process receiving data with I/OAT copy offload does not use any cache line. When not using I/OAT, however, the process first fetches the data in its cache when reading from the sender pages into its processor registers. Then it stores the data back in its receive buffer and thus fills its own cache again. In order to prevent communication from flushing the entire local cache, the cache must be at least two times larger than messages being received. Larger messages should be preferably transfered with I/OAT copy offload. From this idea, we derive the following message size threshold.

$$\text{DMA}_{\text{min}} = \frac{\text{Cache Size}}{2 \times \text{Processes Using The Cache}}$$

When a 4 MiB L2 cache is shared between 2 processes, the formula leads to our 1 MiB threshold. When no cache is shared, each process uses its own cache; the threshold thus jumps to 2 MiB as expected, and the threshold is proportional to the cache size as expected. Then, since we expect users to run one MPI process per core, the threshold may actually be computed from architecture characteristics only, instead of depending on running processes.

$$\text{DMA}_{\text{min}} = \frac{\text{Cache Size}}{2 \times \text{Cores Sharing The Cache}}$$

Another threshold that should be discussed is when NEMESIS should switch from its usual two-copy strategy to KNEM or `vmsplice`. While the `vmsplice` LMT does seem to help performance before 64 KiB (the hardwired threshold), KNEM starts being interesting near 16 KiB messages. Since we do not know how to predict this threshold dynamically, however, we will not change it yet.
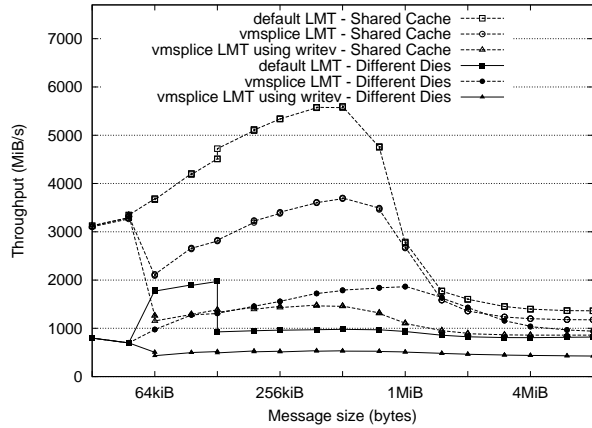
Figure 3. IMB Pingpong with the vmsplice LMT using vmsplice (single-copy) or writev (two copies). The LMT is enabled when the message size passes 64 KiB.

4 Performance Evaluation

We now present performance evaluations of our new LMT backends within MPICH2/NEMESIS. Most experiments were run on a dual-socket quad-core INTEL XEON E5345 (2.33 GHz). Each processor has two 4 MiB L2 caches shared between a pair of cores. We also ran experiments on other hosts, such as a single-socket quad-core XEON X5460 (3.16 GHz) with two 6 MiB L2 caches, and observed similar behavior.

4.1 vmsplice LMT

Figure 3 presents the throughput of a ping-pong measured with the INTEL MPI benchmark (IMB) over MPICH2. As expected, removing the copy on the send side by using `vmsplice` instead of `writev` dramatically increases performance, up to a factor of 2 here. Compared to NEMESIS' usual two-copies LMT implementation, the new `vmsplice` LMT appears to be interesting when no cache is shared between the cores running the processes. If a cache is shared between these cores, however, NEMESIS remains faster because the large overhead of copies is compensated by the reduced data access latency thanks to this cache. Therefore, we feel that NEMESIS should dynamically enable the `vmsplice` LMT when no cache is shared between the processing cores. As pointed out later in this section, this result is specific to very simple communication patterns such as ping-pong. Collective operations will exhibit a larger improvement thanks to `vmsplice` reducing the overall cache pollution and CPU utilization.

4.2 KNEM Performance

Figures 4 and 5 present the throughput of the same IMB Pingpong program with the (synchronous) KNEM LMT. As
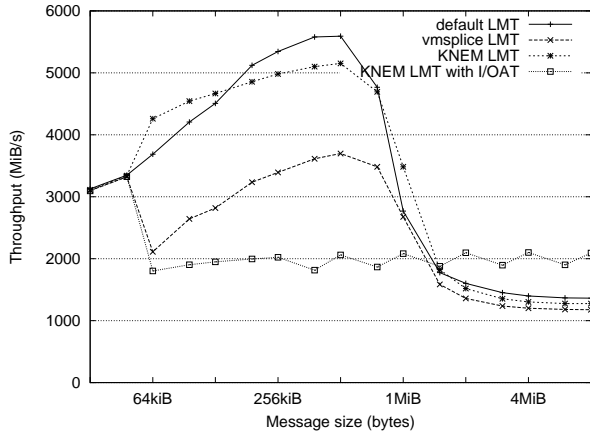
Figure 4. IMB Pingpong throughput between 2 processes sharing a 4 MiB L2 cache.
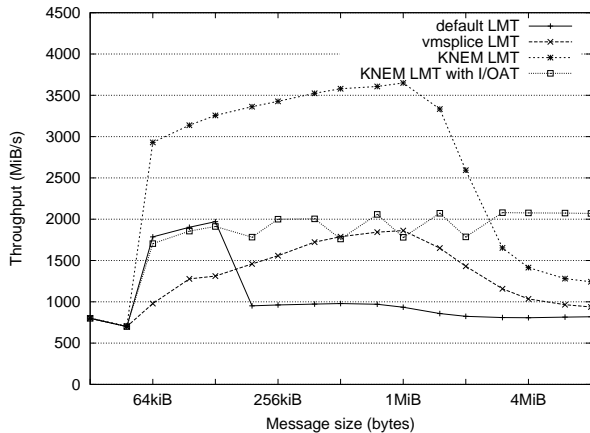
Figure 5. IMB Pingpong throughput between 2 processes not sharing any cache.

expected, KNEM goes far beyond `vmsplice` improvements because KNEM has been designed for MPI access patterns, whereas `vmsplice` is a generic purpose implementation which may for instance be used for multimedia applications. Indeed, `vmsplice` suffers from higher initialization costs due to *Virtual File System* requirements such as dealing with file descriptors and supporting miscellaneous source and destination file types.

If no cache is shared between the processing cores, KNEM is more than three times faster than NEMESIS and twice as fast as `vmsplice`, reaching up to 3.5 GB/s. Indeed, `vmsplice` involves much more synchronization between source and destination processes, causing a large overhead when no cache is shared. If a cache is shared between the processing cores, KNEM remains almost as fast as NEMESIS. We measured that KNEM becomes interesting when the message size passes 8 KiB, whereas NEMESIS usually enables LMT only after 64 KiB.

The KNEM I/OAT backend becomes interesting after 1 MiB, as predicted by the dynamic threshold that we presented in Section 3.5. Indeed, submitting copies to I/OAT requires an access to the physical device for every physically contiguous chunk. So the startup overhead of I/OAT remains higher than the non-I/OAT KNEM implementation. For very large messages, however, I/OAT reduces the CPU consumption and cache pollution and thus improves the overall performance, by a factor of 2.5 over NEMESIS. The I/OAT performance is not very stable because of page alignment problems on which we are still working.

Current INTEL XEON quad-core processors make it possible to bind two processes on the same die without sharing a cache. The behavior of our strategies in this case is similar to the non-shared-cache case (with a small overall throughput degradation due to a single memory link being involved). Moreover, this architecture appears to be a temporary workaround until *true* quad-core processors are available. According to vendor roadmaps, all upcoming x86 processors will share a large cache across all cores. We will thus not detail this case further.

4.3 Asynchronous Model

We explained in Section 3.4 that the KNEM driver can perform asynchronous data transfers, either thanks to I/OAT copy offload being processed in the hardware or by using a kernel thread performing memory copies. Figure 6 presents the corresponding performance evaluation. It shows that offloading regular copies into a kernel thread in the non-I/OAT case significantly reduces the overall throughput since the user-level process competes with the kernel thread for the CPU. However, the I/OAT model is improved by the asynchronous model because the work is performed by a dedicated hardware in the background. For this reason, KNEM enables the asynchronous mode by default only when I/OAT is used.

If greater reactivity is required in NEMESIS, however, the asynchronous model may be needed all the time anyway since the NEMESIS user-process needs to periodically progress while the kernel is working. The synchronous model indeed prevents NEMESIS from polling during several milliseconds when copying several megabytes of data.

The kernel-thread based asynchronous model may also help when there are fewer processes than cores on the machine. Indeed, if half the cores can be dedicated to KNEM kernel threads while the actual processes use the other half, no competition for the CPU occurs. If the processes can be bound near their associated kernel thread, the latter may benefit from the process cache and thus copy as fast as the synchronous model.

5
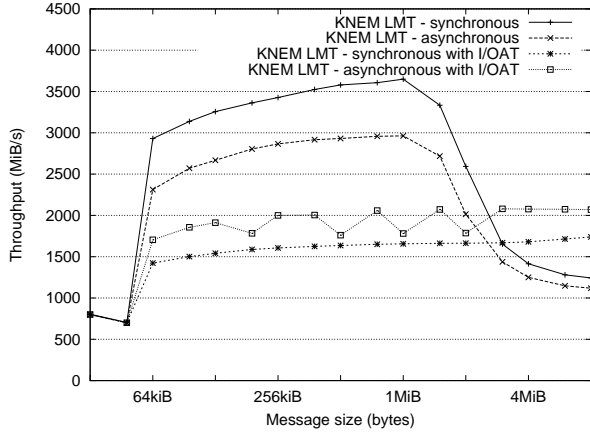
Figure 6. Performance comparison of KNEM synchronous and asynchronous models.
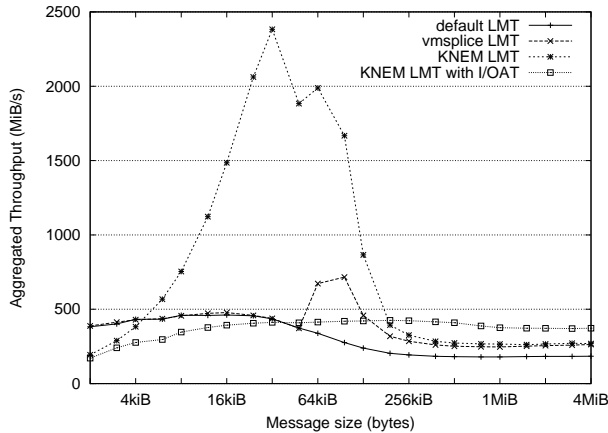
Figure 7. IMB Alltoall aggregated throughput between 8 local processes.

4.4 Collective Operations

Figure 7 illustrates the intranode performance implications of MPI collective communication with `vmsplice`, KNEM and I/OAT. We observed similar behavior for several operations but present only Alltoall results here. This figure confirms that KNEM provides a dramatic performance improvement thanks to reduced cache pollution and CPU usage. The throughput is up to five times higher for medium messages (near 32 KiB) and twice as high for very large messages thanks to I/OAT copy offload. This graph also shows that `vmsplice` provides a smaller but still worthwhile improvement for users who cannot afford loading KNEM's custom kernel module.

We note that I/OAT copy offload becomes interesting near 200 KiB, which is much lower than our 1 MiB predicted threshold in Section 3.5. This is related to the fact that 8 processes participate in this Alltoall benchmark,

Table 1. Execution time of some NAS Parallel Benchmarks.

NAS Kernel	default LMT	vmsplice LMT	KNEM kernel copy	KNEM I/OAT	Speedup
bt.B.4	454.3 s	452.1 s	453.6 s	452.3 s	+ 0.4%
cg.B.8	60.26 s	61.87 s	60.72 s	61.59 s	- 2.2%
ep.B.4	30.45 s	30.94 s	32.40 s	30.72 s	- 0.9%
ft.B.8	39.25 s	37.00 s	36.40 s	35.50 s	**+ 10.6%**
is.B.8	2.34 s	1.95 s	1.92 s	1.86 s	**+ 25.8%**
lu.B.8	85.83 s	87.45 s	86.09 s	88.32 s	- 2.9%
mg.B.8	7.81 s	hang[2]	7.89 s	7.98 s	- 2.1%
sp.B.8	302.0 s	311.4 s	298.9 s	299.4 s	+ 0.9%

whereas only two processes participate in the Pingpong benchmark. Since more messages are transferred simultaneously, the cache and memory bus utilization is higher and thus causes I/OAT to be interesting earlier. Moreover, mixing I/OAT and direct-copy inside a single Alltoall operation seems to improve performance further. We think that this is related to cache affinities between processes.

Similarly, this memory-intensive collective operation suffers earlier from NEMESIS default two-copy strategy. KNEM is thus interesting starting at 4 KiB messages, whereas NEMESIS usually enables the LMT after 64 KiB. Therefore the threshold's current value should be reduced and dynamically set as well.

4.5 NAS Parallel Benchmarks

Table 1 presents the execution times of some NAS Parallel Benchmarks. Most of these benchmarks do not send many large messages and therefore show small or insignificant changes in performance (NAS results slightly vary between successive runs). However, IS, which is known to use very large messages, shows a 25% performance improvement when using KNEM and I/OAT copy offload. As expected, the `vmsplice` LMT brings an interesting gain as well, thanks to fewer memory copies being involved. FT also shows a 10% improvement thanks to this work.

To explain the dramatic performance improvement in IS further, we present in Table 2 the number of cache misses[3] that occur during the execution, which we measured with PAPI [3]. It shows that the execution time of IS is actually somehow linear with the total number of cache misses. KNEM and `vmsplice` using a single memory copy implies less cache pollution, therefore less cache misses and better performance.

Looking at point-to-point and collective behavior underneath also confirms that KNEM and `vmsplice` significantly avoid cache misses for large messages and collec-

---

[2]This hang is due to a known, but as of yet unresolved, bug in NEMESIS, not because of the `vmsplice` LMT backend.

[3]Cache miss rates are not presented because the compared strategies have noncomparable total numbers of cache requests due to their very different implementations.

Table 2. L2 Cache Misses. IS and Alltoall used all 8 cores. Pingpong processes were bound to different dies.

	default LMT	vmsplice LMT	KNEM kernel copy	KNEM I/OAT
64KiB Pingpong	91	166	52	92
4MiB Pingpong	45k	17k	14k	3.7k
64KiB Alltoall	2783	1266	582	833
4MiB Alltoall	624k	124k	262k	131k
is.B.8	11.25M	9.41M	9.50M	8.92M

tives, whereas the regular NEMESIS implementation competes only for small patterns.

5 Related Work

Intranode communication has been the subject of much research in the past decade. All major MPI implementations such as OPEN MPI [9] and MVAPICH [15] provide a similar two-copies strategy through a shared memory-mapped file. Some work has been carried out to improve the latency of small messages [6] and large messages [4]. It has been shown that using the software loopback of modern network interface [12] may significantly improve performance. However, this idea overloads the I/O bus while only the memory bus should be involved.

For more than two years now, `vmsplice` has been available in the LINUX kernel. It has been used mainly as a way to move large flows of data across multiple applications, for instance when processing multimedia streams. To the best of our knowledge, NEMESIS is the first general-purpose and high-performance MPI implementation that benefits from this standard way to communicate while reducing the amount of copies.

LiMIC2 [11] implements a single-copy model through the LINUX kernel in a similar way to KNEM. However, it does not support I/OAT copy offload, vectorial buffers, or asynchronous data transfer. It has been used within MVAPICH2 with configurable thresholds for switching from the usual two-copies to the kernel-based, single-copy model [7]. However, it does not provide any automatic threshold, whereas our KNEM LMT dynamically computes its thresholds depending on the hardware characteristics.

I/OAT copy offload [1] was originally designed as a way to improve datacenter networking performance by reducing the TCP stack overhead on the receive side [10]. User-level memory copy offload with the I/OAT DMA Engine has been studied in a single application [16]. Its comparison with offloading a regular `memcpy` in a thread revealed the same conclusion as ours: I/OAT becomes interesting for megabyte and larger messages [17]. Our work is, however, to the best of our knowledge, the first to experiment I/OAT copy offload between different user-level applications in a general purpose MPI implementation.

Direct-copy between user processes through the kernel has actually been used in some interconnect-specific implementations such as MX [14], which only works when MYRICOM hardware is used for internode communication. OPEN-MX is even able to offload this copy on I/OAT hardware [8], but it requires ETHERNET networking for internode communication. We used this implementation and reworked it to fulfill the requirements of the general-purpose MPI implementation in NEMESIS.

6 Conclusions and Future Work

We have presented several new techniques for intranode message passing in the context of MPICH2, a general-purpose MPI implementation. First we exploited the LINUX `vmsplice` facility to provide high bandwidth communication across a range of LINUX versions with no modification to the standard kernel. Second we have shown the implementation and benefits of using the custom KNEM kernel module [4], including even better performance than the `vmsplice` implementation.

The KNEM module has several additional advantages over previous methods, such as asynchronous transfers and support for I/OAT offloading. Asynchronous transfers help to improve the liveness of the MPI implementation, while I/OAT improves both implementation liveness and reduces the amount of processor usage required to perform the transfer. In addition, I/OAT provides much greater performance at very large message sizes (> 1MiB). As I/OAT technology becomes more widely available in commodity clusters, we expect this technique to become increasingly important.

One of the major advantages to the `vmsplice` approach to interprocess data transfer is its ubiquity, in contrast to a custom kernel module that must be installed such as KNEM. However, the KNEM I/OAT offload support shows much higher performance in certain scenarios than the current implementation of `vmsplice`. Future work in this area will involve examining the feasibility of integrating I/OAT offloading into `vmsplice`-based transfers.

Performance evaluation shows that our work signicantly improves MPICH2, first by bringing high throughput even when no cache is shared between the processing cores, and second by maintaining this result, up to 2 GB/s, even for very large messages, thanks to I/OAT copy offload. Our study of collective operations reveals as expected that reducing the number of memory copies during large message transfers improves performance dramatically thanks to less CPU utilization and better cache efficiency. The large message intensive NAS IS indeed shows far fewer cache misses and a 25% speedup thanks to KNEM.

---

[4] Available for download at `http://runtime.bordeaux.inria.fr/knem`. Version 0.5 was used in this paper.

We have also explored and highlighted the importance of using dynamically computed thresholds for selecting the optimal transfer technique. No single method is optimal for all situations, and so a blended approach is essential for high performance for general benchmarks and applications. There is significant room for future work investigating more sophisticated techniques for computing thresholds than the simple heuristics presented here. For example, it may be possible to lower thresholds for collective communication with the assistance of the upper layers of the MPICH2 stack because the upper layer is aware when multiple large message communications will be occurring in parallel (see Section 4.4). Smaller hardware cache characteristics may also have to be involved when looking at smaller thresholds, such as where to switch from NEMESIS two-copy to the LMT.

The increasing number of cores and large, shared caches in the upcoming processors such as INTEL NEHALEM, and the democratization of nonuniform memory access architecture (NUMA), will keep raising the need to carefully tune intranode communication according to process affinities. We plan to improve our model to automatically exploit these new architecture using the right strategies and thresholds. We also plan to improve KNEM by enabling more communication overlap and easier reactivity thanks to a more flexible driver interface. The idea of moving the knowledge of collective operations down to the LMT also merits study because it may factorize expensive data manipulations currently performed on a per message basis in the LINUX kernel.

Acknowledgments

References

[1] Accelerating High-Speed Networking with Intel I/O Acceleration Technology, May 2005. http://www.intel.com/technology/ioacceleration/306517.pdf.

[2] The splice() weekly news, Apr. 2006. http://lwn.net/Articles/181169/.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

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

[5] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem. In S. J. Turner, B. S. Lee, and W. Cai, editors, *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, pages 521–530, Washington, DC, USA, May 2006. IEEE Computer Society.

[6] D. Buntinas, G. Mercier, and W. Gropp. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. 13th European PVM/MPI Users Group Meeting*, Bonn, Germany, Sept. 2006.

[7] L. Chai, P. Lai, H.-W. Jin, and 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-05)*, Portland, Oregon, Sept. 2008. IEEE Computer Society Press.

[8] 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)*, Weimar, Germany, Feb. 2009. IEEE Computer Society Press.

[9] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, Sept. 2005.

[10] A. Grover and C. Leech. Accelerating Network Receive Processing (Intel I/O Acceleration Technology). In *Proceedings of the Linux Symposium*, pages 281–288, Ottawa, Canada, July 2005.

[11] H.-W. Jin, S. Sur, L. Chai, and 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, Sept. 2007.

[12] M. Koop, W. Huang, K. Gopalakrishnan, and 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, Aug. 2008.

[13] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1, June 2008. http://www.mpi-forum.org/docs/mpi21-report.pdf.

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

[15] Network-Based Computing Lab, The Ohio State University. MVAPICH: MPI for InfiniBand over VAPI Layer. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/.

[16] K. Vaidyanathan, L. Chai, W. Huang, and 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, Sept. 2007.

[17] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using

Hardware Copy Engine: A Case Study with I/OAT. In *Proceedings of the International Workshop on Communication Architecture for Clusters (CAC), held in conjunction with IPDPS'07*, page 234, Long Beach, CA, Mar. 2007.