Benefits of Cross Memory Attach for MPI libraries on HPC Clusters.

Jerome Vienne
Texas Advanced Computing Center
The University of Texas at Austin
viennej@tacc.utexas.edu

ABSTRACT

With the number of cores per node increasing in modern clusters, an efficient implementation of intra-node communications is critical for application performance. MPI libraries generally use shared memory mechanisms for communication inside the node, unfortunately this approach has some limitations for large messages. The release of Linux kernel 3.2 introduced Cross Memory Attach (CMA) which is a mechanism to improve the communication between MPI processes inside the same node. But, as this feature is not enabled by default inside MPI libraries supporting it, it could be left disabled by HPC administrators which leads to a loss of performance benefits to users. In this paper, we explain how to use CMA and present an evaluation of CMA using micro-benchmarks and NAS parallel benchmarks (NPB) which are a set of applications commonly used to evaluate parallel systems.

Our performance evaluation reveals that CMA outperforms shared memory performance for large messages. Micro-benchmark level evaluations show that CMA can enhance the performance by as much as a factor of four. With NPB, we see up to 24.75% improvement in total execution time for FT and up to 24.08% for IS.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Performance

Keywords

Parallel programming, Multicore processing, Performance analysis, MPI

XSEDE '14, July 13 - 18 2014, Atlanta, GA, USA

http://dx.doi.org/10.1145/2616498.2616532 .

1. INTRODUCTION

During the last decade, in an effort to continue following Moore's law, chip manufacturers stopped focusing on speeding up their chips in favor of putting multiple cores on the same chip. This major change increased both the value of and need for parallel applications. Nevertheless, *Message Passing Interface* (MPI) [18] applications were able to exploit these multi-core processors without any change. This helped make MPI the de facto standard for writing parallel applications.

However, to provide high performance to MPI applications in a multi-core environment, an efficient implementation of intra-node communications is critical. Moreover, recent research on topology-aware mapping [19, 17, 11], which tries to reduce the number of hops between communications, increased the need to have fast communications between processes running on the same node.

For intra-node communications, MPI libraries generally use a double-copy implementation via shared memory, but this solution is not ideal for large messages because this mechanism can tie down the CPU and cause cache pollution leading to a waste of bandwidth. To improve the performance of large messages, LiMIC (**Li**nux Kernel Module for **M**PI **I**ntra-Node **C**ommunication) [13] and KNEM (**K**ernel **Nem**esis) [9] were developed. Both were able to greatly improve the bandwidth of MPI communications inside the node, but both are kernel modules which require root privileges for installation and maintenance.

In 2012, a new feature called *Cross Memory Attach* (CMA) was introduced with kernel 3.2. CMA is based on two system calls allowing read/write operations between the address spaces of two processes. Recent versions of MPI libraries like OpenMPI [8] and MVAPICH2 [12] already support CMA and its popularity will certainly grow. Moreover, new HPC clusters, like Stampede or Maverick at TACC, support this kernel feature. Unfortunately, CMA is not always used on clusters that support it due to the lack of evaluation papers and the lack of knowledge of system administrators.

This leads us to the following questions:

- What kind of performance improvement can we expect with CMA compare to shared memory for simple MPI communication?
- What are the potential benefits of using CMA on the performance of parallel scientific applications?
- How does the performance of CMA compare to an existing single copy mechanism such as LiMIC ?

The rest of the paper is organized as follows: Section 2 describes existing mechanisms for intra-node communications and CMA; Section 3 evaluates the benefit of CMA for point-to-point, collectives and applications; and Section 4 summarizes our findings and presents conclusions.

2. INTRA-NODE MPI MECHANISMS

Different mechanisms for intra-node communication exist inside MPI libraries. This section provides an overview of the mostly common ones; they are illustrated in Figure 1.

2.1 Loopback

Some Network Interface Cards (NIC), like Myrinet or Infiniband, have the ability to use Direct Memory Access (DMA) to transfer data between two processes inside the node. However, as shown in Figure 1a, two DMA operations across the I/O buses are performed by the NIC: one to transfer data from the card to the NIC and the other to transfer the data from the NIC to the destination process. These operations can lead to contention and affect performance, but this approach can still provide better performance than shared memory for applications using Infiniband networks [14].

2.2 Shared Memory

Another traditional double-copy implementation involves a shared buffer space used by local processes to exchange messages. As presented in Figure 1b, the sending process copies the content of the message into the shared buffer before the receiver reads from it.
This mechanism is widely used by MPI libraries; many improvements have also been done for collectives [10, 3]. Optimized versions like MPICH2 Nemesis [7] provides an efficient shared memory implementations with low latency and high bandwidth, but this solution is not ideal for large messages because this mechanism can tie down the CPU and cause cache pollution leading to a waste of bandwidth [6]. Nevertheless, shared memory is often the best approach to use for small messages even on large core count nodes [15] as we will see later in Figure 4.

2.3 Kernel assistance

CMA and kernel modules like LiMIC or KNEM enable single copy mechanisms for intra-node communication in MPI libraries. These approaches are only interesting for medium and large messages because the overhead caused by the kernel level calls is higher than the benefit of shared memory for small messages.

2.3.1 LiMIC

LiMIC [13] is a kernel module that allows a process to map and access contiguous portions of a remote process's virtual address space. It uses a simple interface for memory mapping / data copy between processes and is supported by MVAPICH2. Nevertheless, LiMIC has some weaknesses. Passing an invalid LiMIC buffer identifier that will try to access a non-existing address space will cause LiMIC to crash the Linux kernel. Furthermore, there is no security mechanism that prevents users logged on the node to access the content of the memory of all processes.

2.3.2 KNEM

KNEM [9] is a kernel module supported by many MPI implementations, including OpenMPI, MPICH2, BULLX MPI and MVAPICH2. Unlike LiMIC, KNEM uses opaque identifiers called cookies which hide the remote buffer layout. It also includes a checking mechanism to prevent Linux kernel crashes when a non-existing address space is provided. Collective algorithms were also optimized to improve the performance of collective operations inside nodes with KNEM [16]

2.3.3 Cross Memory Attach

CMA (**C**ross **M**emory **A**ttach) was introduced with Linux kernel 3.2 and has been back-ported to some Linux distribution, including RHEL 6.3 and CentOS 6.4. It introduces two system calls, ***process_vm_readv*** and ***process_vm_writev***, which directly read/write to another process's memory given its PID and the remote virtual address.

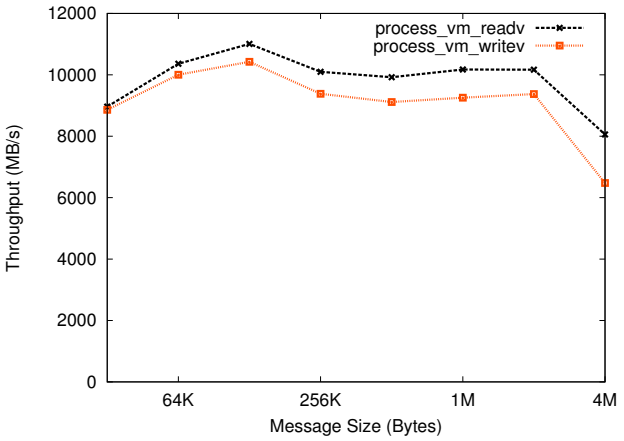


Figure 2: Performance comparison of IMB Pingpong using process_vm_readv and process_vm_writev between two MPI processes on the same socket.

CMA is currently available with OpenMPI 1.7.2 and MVAPICH2 1.9. Figure 2 shows the throughput obtained with IMB Pingpong using MVAPICH2 1.9 that uses the system call *process_vm_readv* and a modified version calling *process_vm_writev*. This experiment is done on a E5-2680 server node, from Stampede, that will be introduced in 3.1. Using *process_vm_readv*, performance is slightly better, this is why both MPI implementations use this system call.

To exploit CMA with their MPI libraries, users need only a MPI library configured with CMA support ("–with-cma" for MVAPICH2 and "–with-cma=yes" for OpenMPI). Depending on your MPI library, once the installation is done, CMA will be enable automatically for large messages (MVAPICH2); or a runtime flag ("–mca btl_sm_use_cma 1" for OpenMPI) may be necessary.

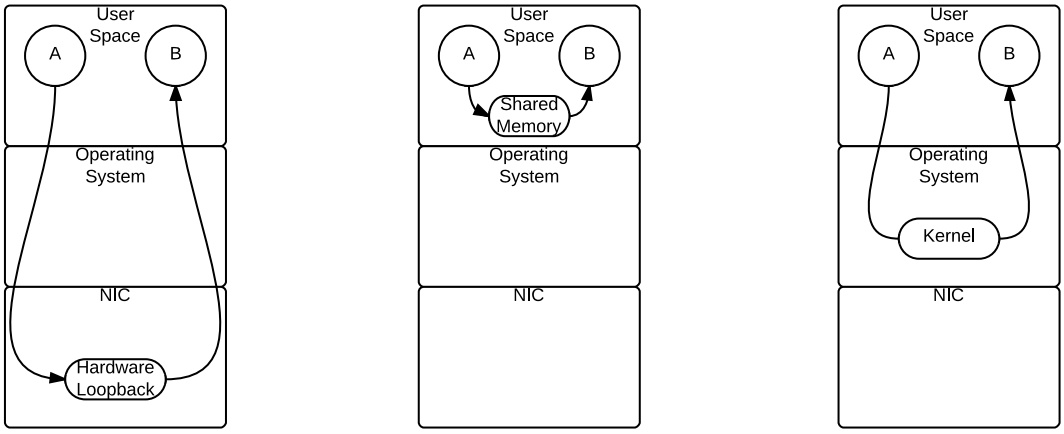
Like KNEM, CMA has security check processes that prevent node crashes. Moreover, there is no need for the cluster administrator to install or maintain CMA. All these small things make CMA attractive.

3. PERFORMANCE EVALUATION

3.1 Experimental Setup

Our experiments have been conducted on TACC's Stampede supercomputer. Stampede contains 6400 dual-socket eight-core Sandy-Bridge E5-2680 server nodes with 32 GB of memory at 1600 MHz, called "compute nodes", and 16 quad-socket eight-core Sandy-Bridge E5-4650 server nodes at 2.7 GHz with 1 TB of memory at 1333 MHz, called "large memory nodes". The nodes are interconnected by InfiniBand HCAs in FDR mode [20] and the operating system used is CentOS 6.4 with kernel 2.6.32-358.el6.

To do this evaluation, we use MVAPICH2 1.9 which is available on Stampede with CMA and LiMIC support. For the point-to-point and collectives evaluation, we use Intel MPI Benchmarks (IMB) [1] version 3.2.4. To evaluate the performance of CMA with applications, we use the well-documented NAS Parallel Benchmarks (NPB) [2, 21, 4]. The NPB is a suite of parallel workloads designed to evaluate performance of various hardware and software



(a) Loopback (b) Shared Memory (c) Kernel assisted

Figure 1: Overview of different existing mechanisms for intra-node communications.

components of a parallel computing system. All results presented here are based on the average of 10 iterations.

3.2 Point-to-point operations

To study the performance of point-to-point communication, it is important to distinguish between inter-socket and intra-socket communication. In fact, performance is greatly improved when MPI processes are sharing the same cache [5].

3.2.1 Intra-socket level

Figures 3, 5 and 6 present the throughput obtained with IMB Pingpong using shared memory, loopback, CMA and LiMIC for intra-socket communication on compute nodes and for different message sizes.

As shown in Figure 3, loopback performs poorly compared to the other mechanisms but we can also observe that there is no difference between shared memory, CMA and LiMIC. Due to the overhead caused by the kernel level calls, MPI libraries don't use kernel assisted mechanisms for these message ranges. Figure 4 shows the execution time of IMB Ping-Pong for messages between 0 and 8 KBytes when CMA and LiMIC are enabled and illustrates this overhead. We can see that CMA performs a little better than LiMIC and that LiMIC is always 0.11 microseconds slower than CMA for this range of messages. This difference could be explained by the fact that LiMIC is a kernel module and that there is a cost for the module call.

For medium sized messages, Figure 5 shows that the switch point between shared memory and CMA is identical to that of LiMIC. After 8 KBytes, the performance of both kernel approaches steeply increase up to the point where they provide a throughput two times larger than shared memory and four times larger than loopback. This point corresponds to the switch point between the eager and rendez-vous protocol. Here, CMA performs slightly better than LiMIC. It is an interesting result, because like KNEM, CMA has a security mechanism to avoid kernel crashes when accessing a non-existing address space. But contrary to KNEM, as described in [9], there is no overhead for medium messages.

Figure 6 displays the throughput for large messages. For this size of messages, we can clearly see the benefit of kernel assisted mechanisms compared to shared memory. LiMIC performs a little better than CMA, but overall, their performance is similar. Here again, shared memory performs better than loopback. Probably due to cache effects, the throughput provided by CMA and LiMIC decreases significantly at message sizes of 4 MBytes.

3.2.2 Inter-socket level

Figure 7 provides the result of the same experiment with MPI processes on two different sockets. For inter-socket communications, CMA and LiMIC still perform better than shared memory and does not seem to suffer too much from a different process placement compared to shared memory. The bad performance of shared-memory is related to double-copy mechanism that can not use the benefit of shared cache. Surprisingly, we see that loopback performs sometimes better than shared memory. This may be due to the performance improvement of PCI Express 3.0 and FDR.

To validate this hypothesis, we retried the same experiment on a large memory node which uses the same infiniband card but with a different CPU with lower memory speed and the result is presented by Figure 8. As LiMIC was showing a similar behavior than CMA, we only show here CMA, shared memory and loopback results. On this system, the performance difference between loopback and shared memory is much higher. The low performance of the shared memory mechanism could be explain by the lower memory speed. For loopback, we can see that the performance on this node is very similar to the one obtained on a compute node, showing that the performance of loopback are mainly lead by the HCA and the PCI bus speed.

3.3 Collective operations

Collective operations involve more than 2 MPI processes and use different kinds of algorithms depending on message sizes and system sizes. We study here the three different classical patterns for collectives: one-to-many, many-to-one and many-to-many. As loopback provides bad performance with collectives and as CMA has a similar throughput than LiMIC, we shows only results obtain for shared memory and CMA. All the experiments show here are done on large memory node.

3.3.1 One-to-many pattern with IMB Scatter

With MPI_Scatter, each process receives a segment from the root. Figure 9 provides the time reported by IMB Scatter for different message sizes on 32 cores. As performance of CMA and shared memory are identical up to 32 KB (cf. 3.2), the results are only reported from 16KB to 4MB. For this range of message sizes, MPI_Scatter inside MVAPICH2 uses an algorithm called *Scatter_Direct*. All the MPI processes reads a segment from the

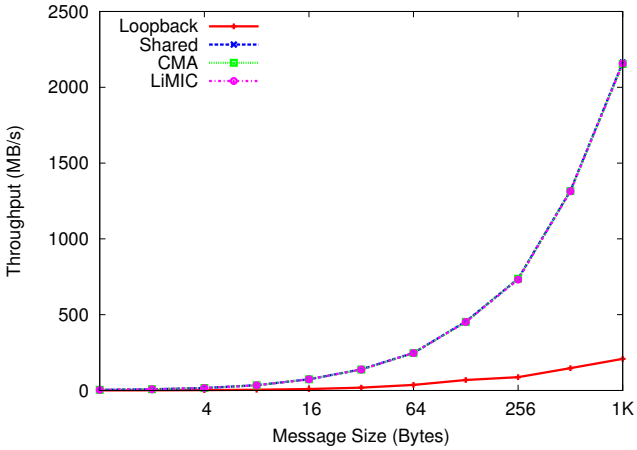


Figure 3: Performance comparison of intra-node mechanisms for small messages using IMB Pingpong between two MPI processes on the same socket.

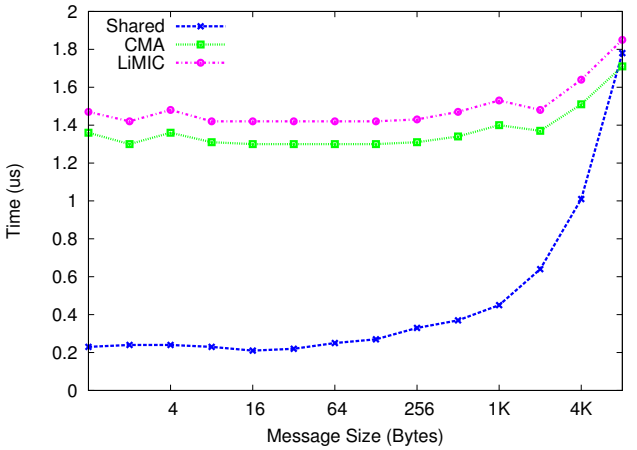


Figure 4: Evaluation of the overhead of LiMIC and CMA compare to Shared Memory for small messages

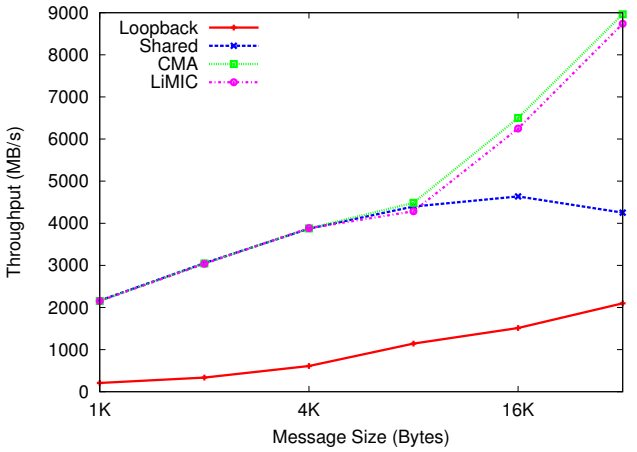


Figure 5: Performance comparison of intra-node mechanisms for medium messages using IMB Pingpong between two MPI processes on the same socket.

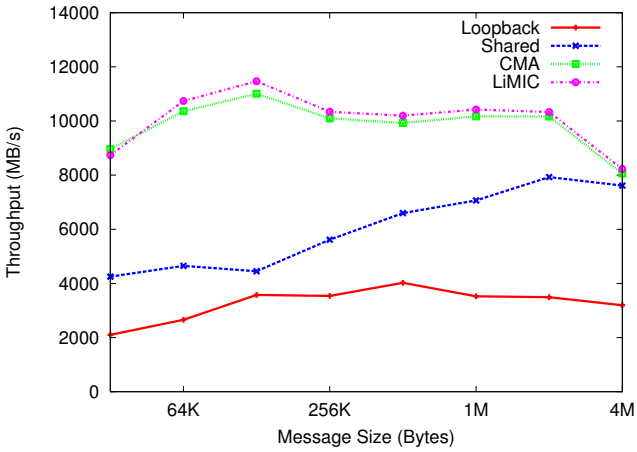


Figure 6: Performance comparison of intra-node mechanisms for small messages using IMB Pingpong between two MPI processes on the same socket.

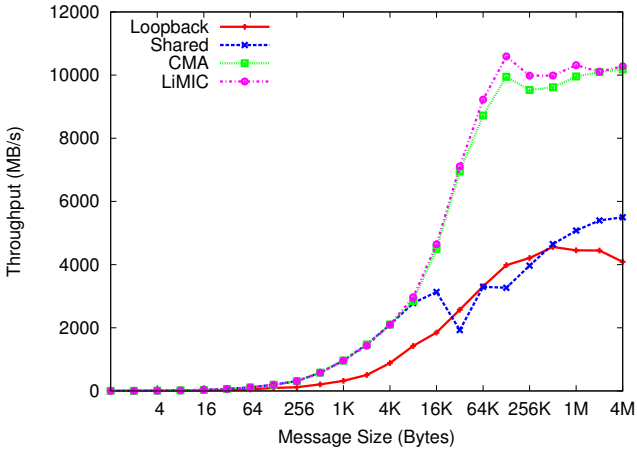


Figure 7: Performance comparison of intra-node mechanisms using IMB Pingpong between two MPI processes on different sockets on a Stampede compute node.

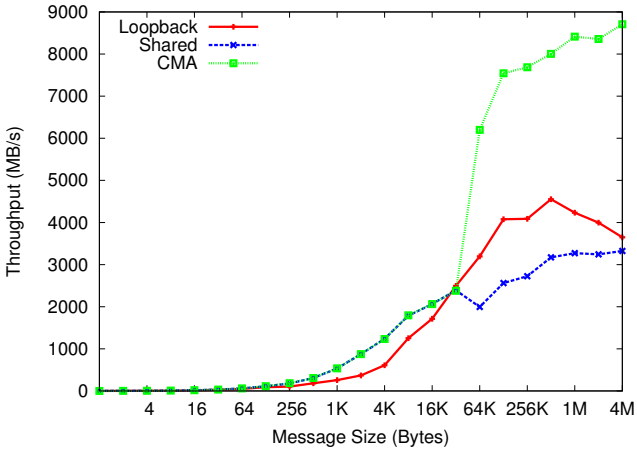


Figure 8: Performance comparison of intra-node mechanisms using IMB Pingpong between two MPI processes on different sockets on a Stampede large memory node.

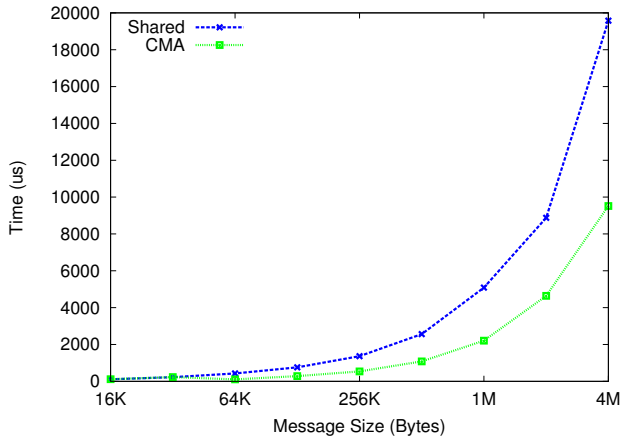


Figure 9: Performance comparison of IMB Scatter using Shared Memory and CMA.

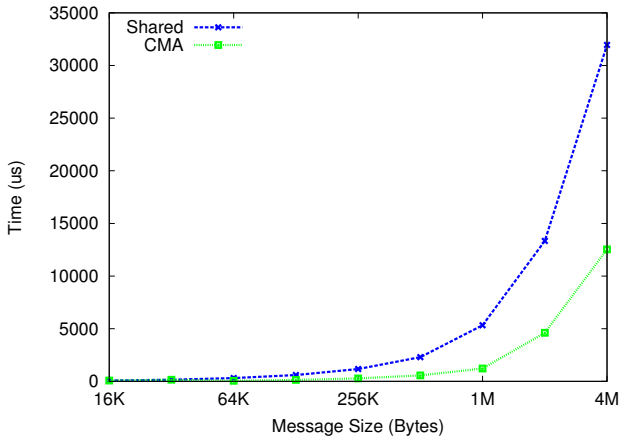


Figure 10: Performance comparison of IMB Gather using Shared Memory and CMA.

root buffer at the same time and CMA handles this perfectly. It is also up to 4 times faster than shared memory for this operation.

3.3.2 Many-to-one pattern with IMB Gather

With MPI_Gather, each process sends contents to the root. This operation is the opposite of the previous one. For this range of message sizes, MPI_Gather inside MVAPICH2 uses an algorithm called *Gather_Direct*. The root process reads a segment from the buffer of each of the other processes; again, CMA handles this well. Figure 10 provides the time reported by IMB Gather for different message sizes on 32 cores. CMA reduce the time from 60-84% compared to shared memory.

3.3.3 Many-to-many pattern with IMB AlltoAll

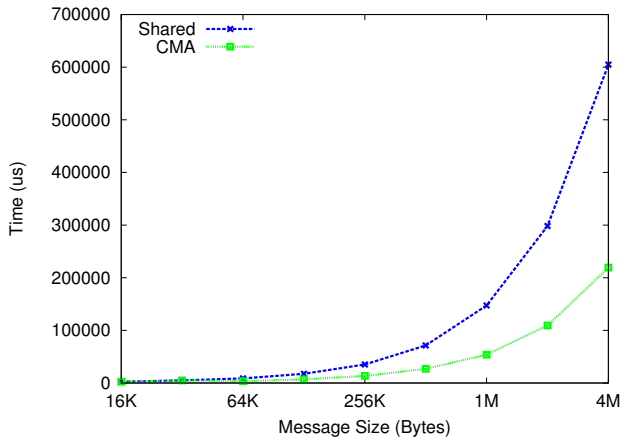


Figure 11: Performance comparison of IMB Alltoall using Shared Memory and CMA.

MPI_Alltoall sends data from all to all processes. For large messages, MVAPICH2 uses pair-wise exchanges. Here each node receives and sends at the same time. These data transfers between processes cause cache pollution and memory contention due to concurrent accesses. Nevertheless, CMA is still able to handle the load. Figure 11 shows the time reported by IMB Alltoall for different message sizes on 32 cores. Here again, CMA outperforms shared memory for large messages.

3.4 Performance of NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) suite contains a set of kernels and pseudo applications designing to mimic the computation and data movement in Computational Fluid Dynamics (CFD) applications. These benchmarks span different problem sizes, called classes in NPB terminology, and in this paper we use classes C and D which are standard for the analysis of large memory node systems.

Benchmark	Class	Shared	CMA	Speedup
CG	C	10.29	9.66	+6.12%
EP	C	3.89	3.88	+0%
FT	C	16.04	12.07	+24.75%
IS	C	1.37	1.04	+24.08%
CG	D	381.95	382.03	-0.02%
EP	D	62.07	62.08	+0.8%
FT	D	365.84	289.32	+20.91%
IS	D	26.1	20.92	+19.8%

Table 1: Execution time of classes C and D NAS benchmarks on 32 processes.

Table 1 shows a performance comparison of classes C and D NAS parallel benchmarks for 32 processes with shared memory and CMA on a large memory node. Integer Sort (IS) and Fast Fourier Transform (FT) are the benchmarks most impacted by the increased performance offered by CMA. This is expected as Integer Sort (IS) and Fast Fourier Transform (FT) are known to be communication bound. The performance improvement provided by CMA allows IS and FT class C to perform 24% faster compared to shared memory. We get 19.8% and 20.91% improvement for IS and FT class D respectively when using CMA compared to shared memory.Please note that results obtained with CMA are similar that the one that we have with LiMIC.

This result shows that CMA benefits not only micro-benchmarks but also real application as long as these applications are communication bound and/or use mostly large message sizes. But unfortunately, CMA does not seem to provide any additional performance improvement compare to existing kernel assisted solutions.

4. CONCLUSION

With the emergence of HPC clusters using processors with an increasing number of cores, it was critical to find a solution to optimize the performance of large message sizes inside nodes. The recent introduction of CMA aimed to fix that and it does.

Our evaluation showed that CMA outperforms shared memory for messages higher than 32 KB which corresponds to the rendezvous protocol message range. We saw also that CMA performance does not depend on process placement like shared memory. Compared to LiMIC or KNEM, CMA does not provide any performance improvement, but it includes some security mechanisms without impacting performance as shown in 3.2.1. And as CMA is part of the Linux kernel, it does not depend on architecture and is portable.This new mechanism will be more and more common inside HPC clusters and will be beneficial for everybody. For administrators, it will not require maintenance like traditional kernel modules. For users, it will provide an easy way to improve the intra-node communications of their applications. For developers, it will allow them to improve the performance of models like the partitioned global address space (PGAS). Previous improvements done for collective algorithms with KNEM will be most likely portable for CMA, allowing all MPI libraries supporting CMA to benefit of them.

As part of future work, we expect to bring CMA into the kernel of the Intel Many Integrated Core (MIC) architecture and study its benefit. We plan also to study CMA on larger core-count nodes and see if new algorithms could be use to improve the performance of collectives with CMA.

5. ACKNOWLEDGMENTS

The author would like to thank Dr Doug James and Dr Todd Evans of the Texas Advanced Computing Center (TACC) for their useful discussions on this topic and feedback on the paper.
This research is supported by National Science Foundation grant #OCI-1134872.

6. REFERENCES

[1] Intel MPI Benchmark. http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[2] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] A. R. Mamidala, A. Vishnu and D. K. Panda. Efficient Shared Memory and RDMA Based Design for MPI-Allgather over InfiniBand. In *PVM/MPI User's Group Meeting*, 2006.

[4] 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, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The Intl. Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In *International Conference on Parallel Processing*, 2009.

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

[7] D. Buntinas, G. Mercier, and W. Gropp. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing*, 33(9):634–644, Sept. 2007.

[8] 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 *PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[9] B. Goglin and S. Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188, 2013.

[10] 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, Lecture Notes in Computer Science*, volume Volume 5205/2008, pages 130–140, 2008.

[11] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *International Conference on Supercomputing*, pages 75–85, 2011.

[12] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. x. D. K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *CCGRID*, pages 43–48, 2006.

[13] H. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *International Conference on Parallel Processing*, pages 184–191, 2005.

[14] M. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda. Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand. In *Hot Interconnects*, 2008.

[15] M. Luo, H. Wang, J. Vienne, and D. Panda. Redesigning MPI Shared Memory Communication for Large Multi-Core Architecture. *Computer Science - Research and Development*, 28(2-3), 2013.

[16] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra. Kernel Assisted Collective Intra-node MPI Communication among Multi-Core and Many-Core CPUs. In *International Conference on Parallel Processing*, pages 532–541, 2011.

[17] G. Mercier and E. Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *EuroMPI*, pages 39–49, 2011.

[18] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing*, 1993.

[19] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes. In *Supercomputing*, pages 70:1–70:12, 2012.

[20] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda. Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems. In *Hot Interconnects*, pages 48–55, 2012.

[21] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Supercomputing*, 1999.