MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption

Marc Perache, Patrick Carribault, Hervé Jourdren

MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption

Marc Pérache, Patrick Carribault, and Hervé Jourdren

CEA,DAM,DIF F-91297 Arpajon, France
{marc.perache,patrick.carribault,herve.jourdren}@cea.fr

Abstract. Message-Passing Interface (MPI) has become a standard for parallel applications in high-performance computing. Within a shared address space, MPI implementations benefit from the global memory to speed-up intra-node communications while the underlying network protocol is exploited to communicate between nodes. But, it requires the allocation of additional buffers leading to a memory-consumption overhead. This may become an issue on future clusters with reduced memory amount per core. In this article, we propose an MPI implementation upon the MPC framework called MPC-MPI reducing the overall memory footprint. We obtained up to 47% of memory gain on benchmarks and a real-world application.

Key words: Message passing, Memory consumption, High-performance computing, Multithreading

1 Introduction

Message-Passing Interface [1] (MPI) has become a major standard API for SPMD[1] programming model in the high-performance computing (HPC) area. It provides a large set of functions abstracting the communications between several *tasks*, would these tasks be on the same address space or on different nodes of a cluster. Within a shared-memory environment, MPI runtimes benefit from the global memory available by using buffers stored inside a memory-segment shared among the tasks (or processes). The size of these buffers is mainly implementation-dependent but it may have a huge impact on the final performance of the running application.

Moreover, the amount of memory per core tends to decrease in recent and future high-performance computers. Currently, the Tera10 machine[2] gives access to 3GB of memory per core with a total of about 10,000 cores. With its 1.6 million cores, the future Sequoia machine will only have on average 1GB of memory available per core. One explanation of this trend could be that the

[1] Single Program Multiple Data

[2] As of November 2008, Tera10 is ranked 48th in the TOP500 list `http://www.top500.org/`

number of cores is still increasing, thanks to Moore's law, while the memory space cannot grow the same speed. The memory per core is therefore becoming a bottleneck for efficient scalability of future high-performance applications. MPI libraries have to adapt their memory consumption to guarantee a correct overall scalability on future supercomputers.

In this paper, we introduce MPC-MPI: an extension to the MPC framework [2] (MultiProcessor Communications). MPC-MPI provides a full MPI-compatible API with a lower memory footprint compared to state-of-the-art MPI runtimes. The MPC framework implements an MxN thread library including a NUMA-aware thread-aware memory allocator and a scheduler optimized for communications. MPC-MPI is developed upon the MPC framework by providing a compatibility with the MPI API version 1.3 [1] where each task is a single thread handled by the user-level scheduler. Thanks to this implementation, we reduced the whole memory consumption on benchmarks and real-world applications by up to 47%.

This article is organized as follows: Section 2 details the related work and Section 3 summarizes the MPC framework, highlighting the features that are already implemented. Then, Section 4 explains how we extended MPC with MPC-MPI before describing its memory gain in Section 5. Section 6 presents experiments comparing MPC-MPI to state-of-the-art MPI implementations before concluding in Section 7.

2 Related Work

A significant part of MPI libraries memory consumption is due to additional buffers required to perform efficient communications. This section describes different approaches and their corresponding memory consumption/efficiency.

2.1 Standard MPI Approach

The standard approach implements MPI tasks as processes. In such runtimes, the message-passing operations rely on copy methods from the source buffer to the destination one. There are four main approaches to accomplish this copy.

Two-copy method: The 2-copy method relies on the allocation of a shared-memory segment between the communicating tasks. First of all, this approach requires a copy from the original buffer to the shared-memory segment. Then, an additional copy is performed from this segment to the destination buffer. The size of these segments is a tradeoff between performance and memory consumption. The best-performing variant allocates a shared buffer for each pair of processes leading to a total of N^2 buffers where N is the number of tasks. Even though this approach is not scalable in memory, it is used in MPICH Nemesis fastbox [3] on nodes with low amount of cores. A more-suitable approach for scalability involves only one buffer per task, as MPICH Nemesis uses for large nodes.

Rendez-vous: The rendez-vous method is a variant of the 2-copy method using large buffers to avoid the message splitting required to fit into the shared-memory segments. This method adds task synchronization to choose a free shared buffer and to perform the 2-copy operation, but one advantage is that it only requires few large buffers for large-message copy.

TODO: rewrite the rendez-vous explanations

One-copy method: The 1-copy method is designed for large-message communications when the MPI library is able to perform a direct copy from the source buffer to the destination. This can be done through a kernel module performing physical-address memory copy. This method is efficient in terms of bandwidth but it requires a system call that increases the communication latency. The main advantage of this method is that it does not require additional buffers. This method is used in MPIBull.

Inter-node communications: The main issue related to memory consumption with inter-node communications is that the number of buffers is often in N^2 where N is the number of process. Whereas processes may share a node, it is hard to share buffers used by the Network Interface Controller (NIC) library. This is due to the process model that requires the separation of the memory address spaces. Nevertheless, some solutions have been proposed to reduce the memory footprint of communications related to the NIC and/or the NIC library capabilities [4, 5].

TODO: Rewrite first 2 sentences: what is *N*?

2.2 Process virtualization

Unlike other approaches [6, 7] that map MPI tasks to processes, *process virtualization* [8–11] benefits from shared-memory available on each cluster node by dissociating tasks from processes: tasks are mapped to threads. This allows efficient 1-copy method and it enables straight-forward optimizations of inter-node communications. Process virtualization is the approach used in the MPC framework.

TODO: more details? (not sure) + talk about issues w/ global variables? (not sure either)

3 The MPC Framework

Our work takes place inside the MultiProcessor Communications (MPC) framework. This environment provides a unified parallel runtime aiming at improving scalability and performance of applications running on large multiprocessor/multicore clusters of NUMA nodes. MPC uses process virtualization as execution model and provides dedicated message-passing functions to enable task communications. The MPC framework is divided into 3 parts: the thread library, the communication layer and the NUMA-aware allocator.

3.1 Thread Library

MPC provides an MxN thread library [12, 13] that maps tasks on lightweight user-level threads. One key advantages of such MxN approach is the ability to optimize the user-level thread scheduler and thus to create and schedule a large number of threads with a reduced overhead. This scheduler provides a polling method that avoids busy-waiting and keeps a high level of reactivity for communications even when the number of tasks is larger than the number of available cores. Furthermore, collective communications have been integrated into the thread scheduler to enable efficient *barrier*, *reduction* and *broadcast* operations.

Explain MxN (maybe 1 small sentence?)

3.2 Memory Allocator

The MPC thread library is linked to a dedicated NUMA-aware and thread-aware memory allocator. It implements a *per-thread heap* [14, 15] approach to avoid contention during allocation and to maintain data locality on NUMA nodes. Each new data allocation is first performed by a lock-free algorithm on the thread private heap. If this local private heap is unable to provide a new memory block, the requesting thread queries a *large page* to the *second-level global heap* with a synchronization scheme. This large page increases its private-heap size leading to the satisfaction of the memory allocation. Memory deallocation is locally performed in each private heap. When a large page is totally free, it is returned to the system with a lock-free method.

3.3 Communications

Finally, MPC provides simple mechanisms for intra-node and inter-node communications. Intra-node communications involve two tasks in the same process (MPC uses one process per node). These tasks use the optimized thread-scheduler polling method and thread-scheduler integrated collectives to communicate with each other. As far as inter-node communications are concerned, MPC uses an underlying MPI library. This method allows portability and efficiency but hampers aggressive communication optimizations.

4 MPC-MPI Extension

This paper introduces MPC-MPI: an extension to the MPC framework providing a whole MPI interface both for intra-node (tasks sharing the same address space) and inter-node communications (tasks located on different nodes). This section deals with the various aspects of creating this extension of the MPC framework.

4.1 MPI Tasks and Intra-Node Communications

The MPC framework already provides basic blocks to handle message passing between tasks. Therefore, MPC-MPI maps MPI tasks to user-level threads instead of processes. Moreover, we optimized point-to-point communications to reduce their memory footprint (see Section 5) whereas collective communications already reach good performance and low memory consumption. MPC-MPI currently exposes an almost-full compatibility with MPI 1.3 [1]: it passes 82% of the MPICH test suite.[3] The missing features are the inter-communicators and the ability to cancel a message.

Nevertheless, compatibility issues remain: MPC-MPI is not 100% MPI compliant due to the process virtualization. Indeed, the user application has to be *thread-safe* because several MPI tasks will be associated with the same process and will share global variables. To ease the migration from standard MPI source code to MPC-MPI application, we modified the GCC compiler to warn the programmer for each global-scope variable (privatizing or removing such variables is one way to guarantee the thread safety of a program).

4.2 Inter-Node Communications

The MPC framework relies on an external MPI library to deal with inter-node message exchange. MPC-MPI now implements its own *inter-node communication layer* based on direct access to the NICs as far as high performance networks are concerned. This new layer enables optimizations according to communication patterns. For example, it allows to add, with a small overhead, the MPC-MPI messages headers required for communications. So far, MPC-MPI provides three network-protocol implementations: TCP, Elan(Quadrics) and InfiniBand- but there are no restrictions to extend this part to other networks. .

5 Memory Consumption Optimization

MPC-MPI implements an almost-full MPI runtime. Thanks to process virtualization, intra-node/inter-node communications and memory-management optimizations, MPC-MPI reduces the memory-consumption of MPI applications. This section depicts the various aspects of memory gain in MPC-MPI.

5.1 Intra-Node Optimizations

The optimizations of memory requirements for intra-node communications mainly rely on the removal of temporary buffers. The unified address space among tasks within a node allows MPC-MPI to use the 1-copy buffer-free method without requiring any system call. That is why the latency of MPC-MPI communications remains low even for small messages. Nevertheless, for optimization purposes, tiny messages (messages smaller than 128B) may use a 2-copy method on a 2KB

[3] `http://www.mcs.anl.goc/research/projects/mpi/mpi_tests/tsuite.html`

per-task buffer. This optimization reduces the latency for tiny messages. Finally, the 1-copy method has been extended to handle non-contiguous messages using derived types without additional packing and unpacking operations.

TODO: more details on 1-copy optimization?

5.2 Inter-Node Optimizations

The second category of memory optimizations are related to the inter-node message exchange. In MPC-MPI, a process handles several tasks and only one process is usually spawn on each node. Thus, network buffers and structures are shared among all the tasks of the process. The underlying NIC library only sees one process and, therefore one identifier for communication between nodes. This enables further optimizations at the NIC level e.g., packet aggregation.

TODO: I changed the last sentence (the old one is still as comment in the TEX file

5.3 Memory Allocation Optimizations

Finally, the MPC memory allocator has been optimized to recycle memory between tasks. Many high-performance applications allocate temporary arrays of objects having a very short lifetime. With a standard memory allocator, these allocations are buffered to avoid a system call each time such event occurs. MPC-MPI uses smaller buffers for each thread private heap but it also maintains a shared pool of memory pages in the global heap to avoid system calls. It enables intra-node page sharing between tasks without system calls unlike in the multiprocess approach.

6 Experiments

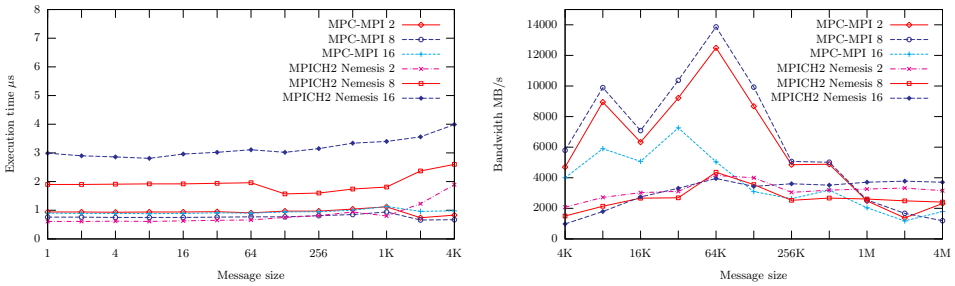
This section describes the experimental results of MPC-MPI, conducted on two architectures: a dual Intel Nehalem-EP (8 cores) with 12GB of main memory running Red Hat Enterprise Linux Client release 5.2 and an octo-dual core Intel Montecito with 48GB of memory running Bull Linux AS4 V5.1 (one node of the CEA/DAM Tera10 machine). To evaluate the MPI performance of MPC-MPI, the Intel MPI Benchmarks[4] have been used with the option *-off-cache* to avoid cache re-usage and, therefore, to provide realistic throughput. The tests compare MPC-MPI to MPICH 2 1.1a2 Nemesis on the Nehalem architecture and MPIBull 1.6.5 on the Tera10 machine.[5] These experiments are divided into 3 categories: (i) the point-to-point benchmarks, (ii) the collective benchmarks and (iii) the results on our real-world application.

[4] `http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm`

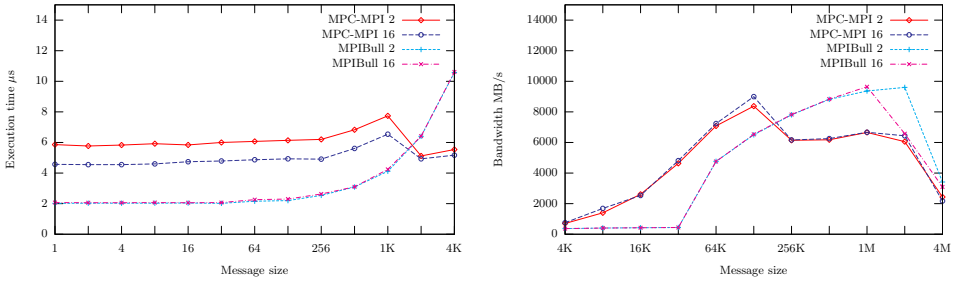
[5] MPIBull is the vendor MPI available on the Tera10 machine.

6.1 Point-to-Point Benchmarks

Figure 1 illustrates the latency and the bandwidth of the MPC-MPI library on the standard pingpong benchmark with two tasks communicating among a total of 2, 8 or 16 tasks. MPC-MPI is a bit slower than MPICH Nemesis on the Nehalem architecture when the number of MPI tasks is low. But with a total of 8 tasks or with 16 hyperthreaded tasks per node, MPC-MPI keeps a constant latency and bandwidth whereas MPICH Nemesis slows down. As far as the Tera10 machine is concerned, MPC-MPI is a bit slower than the vendor MPI library: MPIBull.



(a) Latency and bandwidth evaluation of the Nehalem platform.

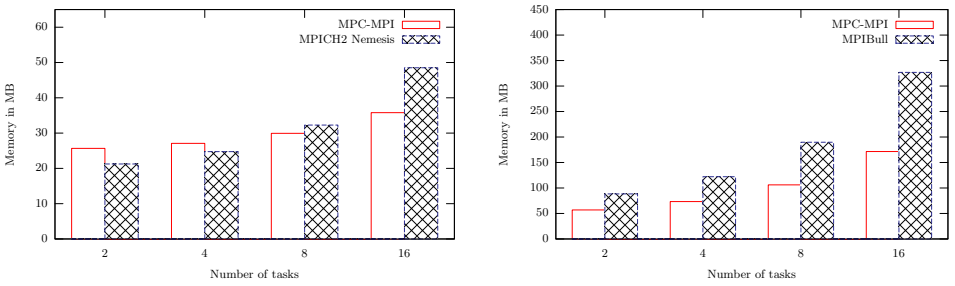


(b) Latency and bandwidth evaluation of Tera10 platform.

Fig. 1. Performances of the point to point communication benchmark

Figure 2 depicts the memory consumption of MPC-MPI compared to other implementations. The different optimizations described in this paper leads to memory gain when the node is fully used even on a simple test such as a ping-pong. MPC-MPI allows to reduce the overall memory consumption by up to 26% on the Nehalem compared to MPICH and up to 47% compared to MPIBull.

This pinpong benchmark illustrates the rather good performance of MPC-MPI compared to other MPI libraries in terms of latency and bandwidth. It also illustrates the benefits of MPC-MPI memory optimizations that lead to save a significant amount of memory even on this simple test.

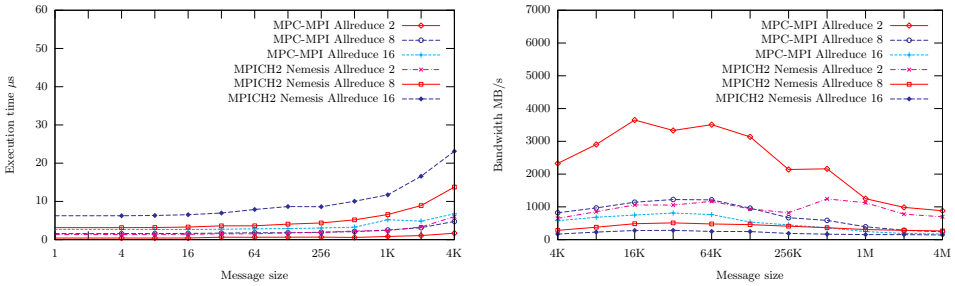


(a) Nehalem platform. (b) Tera10 platform.

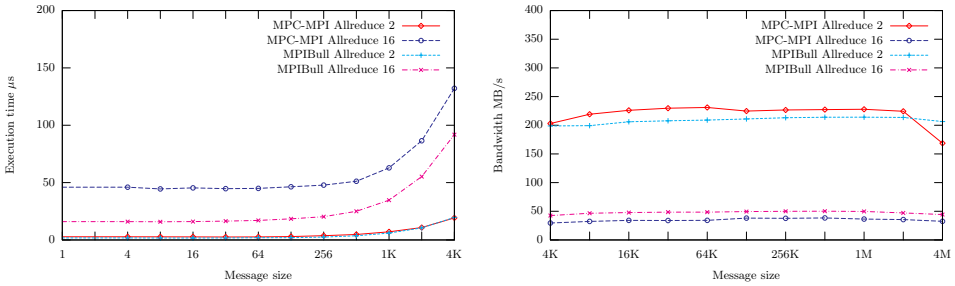
Fig. 2. Memory consumption of the point to point communication benchmark

6.2 Collective Benchmarks

The second benchmark category is related to collective communications. Figure 3 shows the latency and bandwidth of the MPC-MPI library on the Allreduce benchmark. MPC-MPI is a bit faster than MPICH Nemesis on the Nehalem architecture whatever the number of tasks is. On the Tera10 machine, MPC-MPI is still a bit slower than MPIBull.



(a) Latency and bandwidth evaluation of the Nehalem platform.

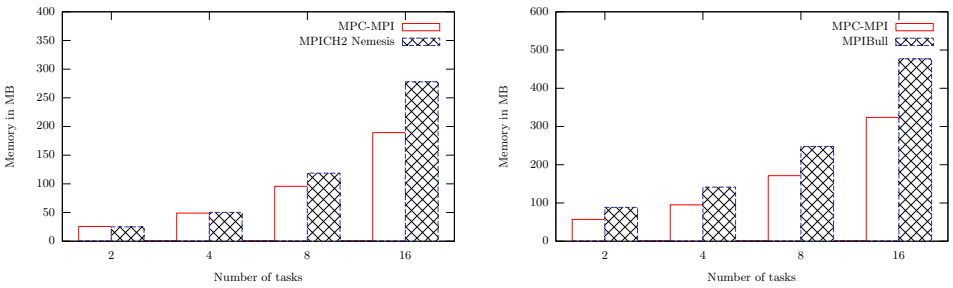


(b) Latency and bandwidth evaluation of the Tera10 platform.

Fig. 3. Performances of the allreduce communication benchmark

Figure 4 illustrates the memory consumption of MPC-MPI compared to other implementations. The results with the Allreduce benchmark is similar to the

pingpong one (see Section 6.1). MPC-MPI saves up to 31% of the overall memory on the Nehalem compared to MPICH and up to 32% compared to MPIBull.



(a) Nehalem platform. (b) Tera10 platform.

Fig. 4. Memory consumption of the allreduce communication benchmark

6.3 Application

MPC-MPI has been used on the application HERA on the Tera10 supercomputer. HERA is a large multi-physics 2D/3D AMR hydrocode platform [16]. On a typical ablation front problem, with 3-temperature multifluid hydrodynamics (explicit CFD solver) coupled to 3-temperature diffusion (iterative Newton-Krylov solver + preconditioned conjugate-gradient method), MPC-MPI allows a 47% reduction of memory consumption (1495MB for MPC-MPI instead of 2818MB for MPIBull) on a 16-core Tera10 NUMA node, for just 8% overhead compared to native MPIBull (about 400,000 cells, with aggressive Adaptive Mesh Refinement and dynamic load balancing).

This large C++ code uses number of derived types and collective communications during its execution, demonstrating the robustness of the MPC-MPI library.

7 Conclusion and Future Work

In this paper we proposed MPC-MPI: an extension of the MPC framework providing an MPI-compatible API. This extension reduces the overall memory consumption of MPI applications thanks to the process virtualization and a large number of optimizations concerning the communications (intra node and inter node) and the memory allocator. Experiments on standard benchmarks and a large application show significant memory gains (up to 47%) with only a small performance slowdown compared to vendor MPI (MPIBull on Tera10) and state-of-the-art implementations (MPICH Nemesis). Furthermore, optimizations of inter-node communications in MPC-MPI does not hamper further optimizations proposed by specialized high performance communications library. The benefit of these library are cumulative with MPC-MPI optimizations.

There are still some missing features to propose a full MPI API, these will be implemented in the near future. Furthermore, some work has to be done concerning the thread-safety needed by MPC-MPI. So far, only a modified GCC is able to extract the global variables of the application, but no tools are available to help the programmer to transform its code. Finally, the MPC-MPI extension on MPC is the first step to provide an optimized hybrid MPI/thread framework.

References

1. Message Passing Interface Forum: MPI: A message passing interface standard. (1994)
2. Pérache, M., Jourdren, H., Namyst, R.: MPC: A unified parallel runtime for clusters of NUMA machines. In: Proceedings of the 14th International Euro-Par Conference (Euro-Par 2008). (2008)
3. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid. (2006)
4. Sur, S., Koop, M.J., Panda, D.K.: High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. (2006)
5. Koop, M.J., Jones, T., Panda, D.K.: Reducing connection memory requirements of MPI for InfiniBand clusters: A message coalescing approach. In: CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid. (2007)
6. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: High-performance, portable implementation of the MPI Message Passing Interface Standard. Parallel Computing (1996)
7. Graham, R.L., Woodall, T.S., Squyres, J.M.: Open MPI: A flexible high performance mpi. In: Parallel Processing and Applied Mathematics. (2006)
8. Kalé, L.: The virtualization model of parallel programming: runtime optimizations and the state of art. In: LACSI. (2002)
9. Demaine, E.: A threads-only MPI implementation for the development of parallel programming. In: Proceedings of the 11th International Symposium on High Performance Computing Systems. (1997)
10. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Languages and Compilers for Parallel Compution (LCPC). (2004)
11. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: ICS'01: Proceedings of the 15th International Conference on Supercomputing. (2001)
12. Namyst, R., Mhaut, J.F.: PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In: Parallel Computing (ParCo '95). (1995)
13. Abt, B., Desai, S., Howell, D., Perez-Gonzalet, I., McCraken, D.: Next Generation POSIX Threading Project (2002)
14. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: a scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). (2000)

15. Berger, E., Zorn, B., McKinley, K.: Composing high-performance memory allocators. In: Proceedings of the ACM SIGPLAN conferance on Programming Language Design and Implementation. (2001)
16. Jourdren, H.: HERA: A hydrodynamic AMR platform for multi-physics simulations. In: Adaptive Mesh Refinement - Theory and Application, LNCSE. (2005)