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 single cluster node, MPI implementations benefit from the shared memory to speed-up intra-node communications while the underlying network protocol is exploited to communicate between nodes. However, 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 built 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.

Keywords: 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.

The amount of memory per core tends to decrease in recent high-performance computers. Currently, the Tera-10 machine[2] provides 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 number of cores is still increasing, thanks to Moore's law, while the memory space cannot grow the same speed. The

[1] Single Program Multiple Data.

[2] As of November 2008, Tera-10 is ranked 48th in the TOP500 list `http://www.top500.org/` and was 5th when it was installed, in June 2006.

M. Ropo et al. (Eds.): EuroPVM/MPI 2009, LNCS 5759, pp. 94–103, 2009.

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 and 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 current features of the MPC framework. Then, Section 4 explains how we extended MPC with MPC-MPI. Memory consumption optimizations are detailed 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 the memory consumption of MPI libraries is due to additional buffers required to perform efficient communications. This section describes standard approaches to handle the issue.

2.1 Standard MPI Approaches

According to the MPI standard, MPI tasks are processes. In MPI 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. If a message is larger than the memory buffer, it has to be split. The size and the number 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.

Two-copy with Rendez-vous method: The rendez-vous method is a variant of the 2-copy method using a few larger additional buffers to realize the intermediate copy. This method improves performance for large messages, avoiding message splitting; but it requires a synchronization of involved tasks to choose a free buffer. This method increases the memory consumption.

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: Network Interface Controller (NIC) libraries require per process data structure allocations. The size of these data is often related to the number of neighbors of each process. Thus, the memory consumption is N^2 where N is the number of processes. Whereas some processes may share a node, NIC libraries are not able to share these data 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 related to the NIC and the NIC library capabilities [4,5].

2.2 Process Virtualization

Unlike standard MPI approaches that map tasks to processes, *process virtualization* [6,7,8] benefits from shared-memory available on each cluster node by dissociating tasks from processes: tasks are mapped to threads. It allows efficient 1-copy method or straight-forward optimizations of inter-node communications. Nevertheless, process virtualization requires to restrict the use of global variables because many processes now share a unique address space.

3 The MPC Framework

MPC-MPI is an extension of the MultiProcessor Communications (MPC) framework [2]. This environment provides a unified parallel runtime built to improve the scalability and performance of applications running on clusters of large multiprocessor/multicore NUMA nodes. MPC uses process virtualization as its execution model and provides dedicated message-passing functions to enable task communications. The MPC framework is divided into 3 parts: the thread library, the memory allocator and the communication layer.

3.1 Thread Library

MPC comes with its own MxN thread library [9,10]. MxN thread libraries provide lightweight user-level threads that are mapped to kernel threads thanks to a user-level thread scheduler. One key advantage of the MxN approach is the ability to

optimize the user-level thread scheduler and thus to create and schedule a *very* large number of threads with a reduced overhead. The MPC thread 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 much larger than the number of available cores. Furthermore, collective communications are integrated into the thread scheduler to enable efficient *barrier*, *reduction* and *broadcast* operations.

3.2 Memory Allocation

The MPC thread library is linked to a dedicated *NUMA-aware* and *thread-aware* memory allocator. It implements a *per-thread heap* [11,12] 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. A *large page* is a parametrized number of system pages. Memory deallocation is locally performed in each private heap. When a large page is totally free, it is returned to the *second-level global heap* with a lock-free method. Pages in *second-level global heap* are virtual and are not necessarily backed by physical pages.

3.3 Communications

Finally, MPC provides basic mechanisms for intra-node and inter-node communications with a specific API. Intra-node communications involve two tasks in a 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 used up to now 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 that provides a whole MPI interface for both intra-node (tasks sharing a same address space) and inter-node communications (tasks located on different nodes). This section deals with the implementation choices of MPC-MPI.

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. Point-to-point communications have been optimized to reduce the memory footprint (see Section 5) whereas collective communications

already reached 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 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 may be associated to 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 reduces the overhead of adding the MPC-MPI message headers required for communications. So far, MPC-MPI supports three network protocols: TCP, Elan(Quadrics) and InfiniBand but there are no restrictions to extend this part to other networks.

5 Memory Consumption Optimization

MPC-MPI reduces the memory consumption of MPI applications thanks to process virtualization, intra-node/inter-node communications and memory-management optimizations. This section depicts these optimizations.

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

5.2 Inter-node Optimizations

The second category of memory optimizations are related to the inter-node message exchange. With MPC-MPI, a process handles several tasks and only one

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

process is usually spawned 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.

5.3 Memory Allocation Optimizations

Finally, the MPC memory allocator has been optimized to recycle memory chunks between tasks. Many high-performance applications allocate temporary objects that have very short lifetimes. With a standard memory allocator, these allocations are buffered to avoid a system call each time such event occurs. MPC-MPI uses smaller buffers in 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 the system calls required in the multiprocess approach.

6 Experiments

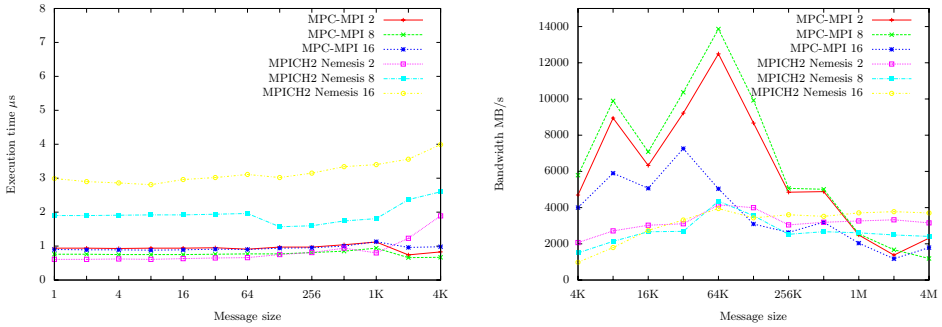
This section describes some experimental results of MPC-MPI, conducted on two architectures: a dual-socket Intel Nehalem-EP (2×4 cores) with 12GB of main memory running Red Hat Enterprise Linux Client release 5.2 and an octo-socket Intel Itanium Montecito (8×2 cores) with 48GB of memory running Bull Linux AS4 V5.1 (one node of the CEA/DAM Tera-10 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 Itanium Tera-10 machine.[5] These experiments are divided into 3 categories: (i) point-to-point communication benchmarks, (ii) collective communication benchmarks and (iii) results on our real-world application.

6.1 Point-to-Point Benchmarks

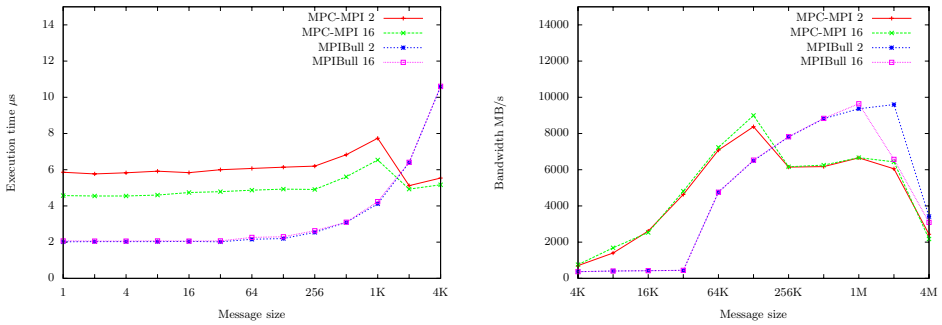
Figure 1 provides the latency and the bandwidth of the MPC-MPI library on the standard pingpong benchmark, with two tasks communicating among a total of 2, 8 or 16 tasks. MPC-MPI is a bit slower than MPICH Nemesis on the Nehalem architecture when the number of MPI tasks is low. But with a total of 8 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 Itanium Tera-10 machine is concerned, MPC-MPI is a bit slower than MPIBull. This gap is mainly due to a lack of Itanium-specific optimizations in MPC-MPI.

Figure 2 depicts the memory consumption of MPC-MPI compared to other implementations. The optimizations described in this paper lead to memory gain

[4] `http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm`.
[5] MPIBull is the vendor MPI available on the Tera-10 machine.

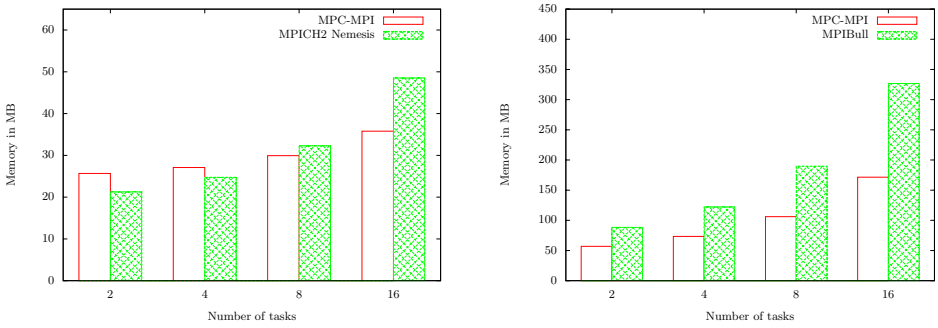


(a) Latency and bandwidth results for the Nehalem platform.



(b) Latency and bandwidth results for the Itanium Tera-10 platform.

Fig. 1. Performance of the point-to-point communication benchmark

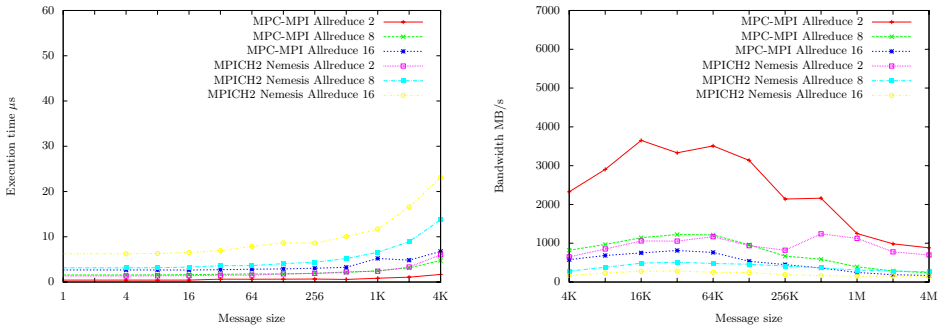


(a) Nehalem platform. (b) Itanium Tera-10 platform.

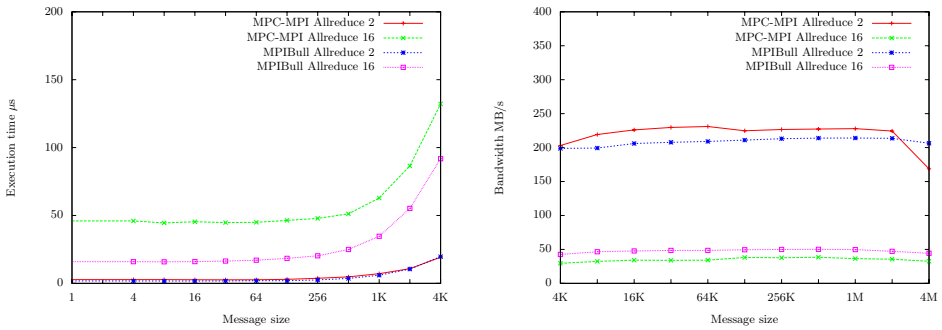
Fig. 2. Memory consumption of the point-to-point communication benchmark

when the node is fully used, even on a simple test such as a pingpong. MPC-MPI allows to reduce the overall memory consumption by up to 26% on the Nehalem compared to MPICH and up to 47% on the Itanium compared to MPIBull.

This pingpong benchmark illustrates the rather good performance of MPC-MPI compared to other MPI libraries in terms of latency and bandwidth. It also



(a) Latency and bandwidth results for the Nehalem platform.



(b) Latency and bandwidth results for the Itanium Tera-10 platform.

Fig. 3. Performance of the allreduce communication benchmark

illustrates the benefits of MPC-MPI memory optimizations that lead to save a significant amount of memory, even on this simple test.

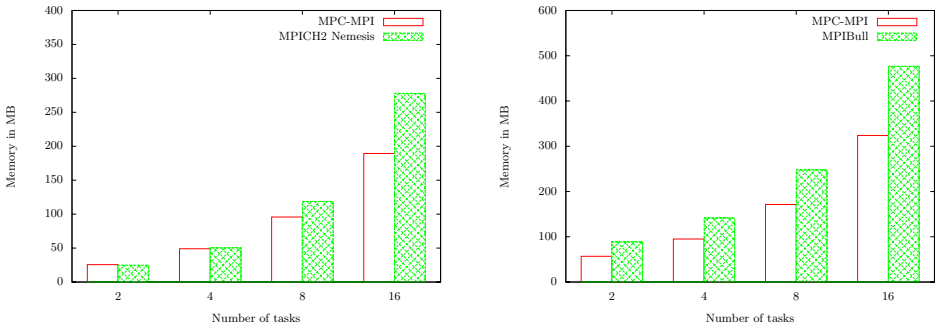
6.2 Collective Benchmarks

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 Itanium Tera-10 machine, MPC-MPI is still a bit slower than MPIBull.

Figure 4 illustrates the memory consumption of MPC-MPI compared to other implementations. The results with the Allreduce benchmark are similar to the pingpong ones. MPC-MPI saves up to 31% of the overall memory on the Nehalem compared to MPICH and up to 32% compared to MPIBull on Itanium.

6.3 Application

MPC-MPI has been used with the HERA application on the Tera-10 supercomputer. HERA is a large multi-physics 2D/3D AMR hydrocode platform [13].



(a) Nehalem platform. (b) Tera-10 platform.

Fig. 4. Memory consumption of the allreduce communication benchmark

On a typical ablation front fluid-flow problem, with 3-temperature multifluid hydrodynamics (explicit Godunov-type CFD solver) coupled to 3-temperature diffusion (iterative Newton-Krylov solver + preconditioned conjugate-gradient method), MPC-MPI allows a 47% reduction of memory consumption (1,495MB for MPC-MPI instead of 2,818MB for MPIBull) on a 16-core Tera-10 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 presented 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 process virtualization and a large number of optimizations concerning communications (intra-node and inter-node) and memory management. 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 Itanium Tera-10) and a state-of-the-art MPI implementation (MPICH Nemesis on Nehalem). Furthermore, optimizations of inter-node communications do not hamper further optimizations proposed by specialized high-performance communication NIC libraries on dedicated hardware.

There are still missing features for a full MPI 1.3 API, features to be implemented in the near future. Some work has also to be done concerning thread-safety required by MPC-MPI. So far, a modified GCC compiler is provided to highlight global variables of the application, but no tool is available to transform the application code.

The MPC-MPI extension of MPC[6] is the first step to provide an optimized hybrid MPI/thread runtime system on Petaflops machines, addressing more specifically the *stringent* memory constraints appearing on such computer architectures.

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: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)
3. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In: CCGRID 2006: 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 2006: 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 2007: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (2007)
6. Kalé, L.: The virtualization model of parallel programming: runtime optimizations and the state of art. In: LACSI (2002)
7. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Languages and Compilers for Parallel Compution, LCPC (2004)
8. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: ICS 2001: Proceedings of the 15th International Conference on Supercomputing (2001)
9. Namyst, R., Méhaut, J.F.: PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In: Parallel Computing, ParCo 1995 (1995)
10. Abt, B., Desai, S., Howell, D., Perez-Gonzalet, I., McCraken, D.: Next Generation POSIX Threading Project (2002)
11. 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)
12. 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)
13. Jourdren, H.: HERA: A hydrodynamic AMR platform for multi-physics simulations. In: Adaptive Mesh Refinement - Theory and Application, LNCSE (2005)

[6] MPC, including the MPC-MPI extension, is available at http://mpc.sourceforge.net.