Improving MPI Communication Overlap with Collaborative Polling

Sylvain Didelot[1,3], Patrick Carribault[2,1],
Marc Pérache[2,1], and William Jalby[1,3]

[1] Exascale Computing Research Center, Versailles, France
{sylvain.didelot,william.jalby}@exascale-computing.eu
[2] CEA, DAM, DIF F-91297, Arpajon, France
{patrick.carribault,marc.perache}@cea.fr
[3] Université de Versailles Saint-Quentin-en-Yvelines (UVSQ), Versailles, France

Abstract. With the rise of parallel applications complexity, the needs in term of computational power are continually growing. Recent trends in High-Performance Computing (HPC) have shown that improvements in single-core performance will not be sufficient to face the challenges of an Exascale machine: we expect an enormous growth of the number of cores as well as a multiplication of the data volume exchanged across compute nodes. To scale applications up to Exascale, the communication layer has to minimize the time while waiting for network messages. This paper presents a message progression based on Collaborative Polling which allows an efficient auto-adaptive overlapping of communication phases by performing computing. This approach is new as it increases the application overlap potential without introducing overheads of a threaded message progression.

Keywords: HPC, Overlap, MPI, High-Speed Network, Polling.

1 Introduction

The scalability of a parallel application is mainly driven by the amount of time wasted in the communication library. One solution to decrease the communication cost is to hide communication latencies by performing computation during communications. From the application developer's point of view, parallel programming models offer the ability to express this mechanism through non-blocking communication primitives. One of the most popular communication libraries, Message Passing Interface (MPI), allows the programmer to use non-blocking send and receive primitives (i.e., `MPI_Isend` and `MPI_Irecv`) to enable overlapping of communication with computation. For example, Figure 1-a exposes one MPI task performing a non-blocking communication without overlapping capabilities. In such a situation, the message is actually received from the network during the `MPI_Wait` call. On the other hand, the same example with overlapping shows a significant improvement reducing the overall time consumed (see Fig. 1-b).

Achieving overlap usually requires a lot of code restructuring and transformations. Users are often disappointed after spending a lot of time to enforce overlap because the runtime does not provide an efficient support for asynchronous

J.L. Träff, S. Benkner, and J. Dongarra (Eds.): EuroMPI 2012, LNCS 7490, pp. 37–46, 2012.

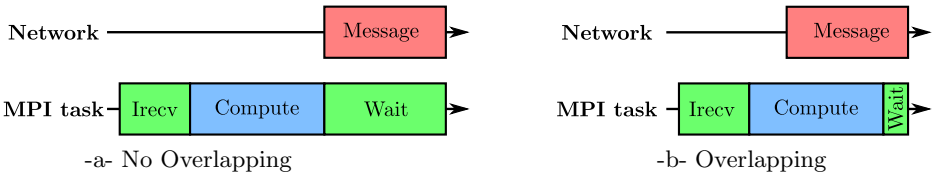

Fig. 1. Influence of Communication/Computation Overlapping in MPI

progress [1,2]. The MPI standard does not define a clear implementation rule for asynchronous communications but only gives recommendations. Most of the current MPI libraries does not support true asynchronous progression and performs message progression within MPI calls (i.e., inside `MPI_Wait` or `MPI_Test` functions). The main difficulty with these implementations occurs when an MPI task performs a time consuming function with no call to MPI routines for progressing messages (i.e., calls to BLAS).

In this paper, we propose a collaborative polling approach for improving the communication overlap without disturbing compute phases. This runtime optimization has been implemented inside a thread-based MPI runtime called MPC (Multi-Processor Computing [3]). Collaborative polling allows message progression when a task is blocked waiting for a message, enabling overlapping with any other task within the same compute node. This method expresses a significant message-waiting reduction on scientific codes. In this paper, we focus on the MPI standard and Infiniband network but the collaborative polling could be adapted to any network interconnect and could be extended to other distributed-memory programming models.

2 Related Work

2.1 Message Progression Strategies

Researches provide significant speedups using overlap of communication on large scale scientific applications [4,5]. For common MPI runtimes, message progression is accomplished when the main thread calls a function from the MPI library. To achieve overlap at user level, MPI applications may be instrumented with repeated calls to the `MPI_Test` function to test all outstanding requests for completion. This solution is not convenient for the developer and irrelevant for not MPI-aware functions. For implementations supporting the `MPI_THREAD_MULTIPLE` level of thread safety, Thakur et al. [6] present an alternative overlapping technique. Hager et al. [7] investigate a Hybrid MPI/OpenMP implementation with explicit overlap optimizations. However, both techniques rely on source-code modifications and involve multiple programming models.

Recent Host Channel Adapters (HCAs) provide hardware support for total or partial independent progress but rely on specific network hardware capabilities [8]. To enable software overlapping without user source code modifications, MPI libraries investigate a threaded messages progression. Additional threads

(also known as progression threads) are created to retrieve and complete outstanding messages even if large computation loops prevent the main thread to call the runtime library. For accessing the network hardware, progression threads may be set to use the polling or the interrupted-driven methods.

The polling access approach increases performance on a spare-core thread subscription where the progression thread is bound on a dedicated core. It was for example adopted by IBM in the Bluegene systems [9]. Because only a part of the cores participates to computation, the spare-core mode is barely used on regular HPC clusters. MPI is often used in a fully subscribed mode sharing the progression thread and the user thread on the same core. However the decision when and how often the polling function should be called is non-trivial. Too many calls may cause overhead and not enough calls may waste the overlap potential.

The interrupted-driven message detection is different from the polling approach since it allows the sender or the receiver to have an immediate notification of completed messages [10]. If no work has to be done, the progression thread enters into the wait queue and goes to sleep. When a specific event is generated from the network card (i.e., an incoming message), an interruption is emitted and the progression thread goes back to the run queue. Because generating an interruption for each message may be costly, MPI runtimes often implement a selective interrupt-based solution [11, 12]. Only messages which are critical for overlapping performance may generate an interruption.

For the fairness of the CPU resource sharing, each process has a maximum time to run on a CPU: the time-slice. For example on a Linux kernel, it varies from 1 to 10 milliseconds. Once the time-slice is elapsed, the scheduler interrupts the current running thread, places it at the end of the run queue for its static priority and schedules a new runnable thread. When an interruption occurs, the progression thread has to be immediately scheduled, raising two main concerns. First, it is unclear how much time is required to switch from the active thread to the progression thread: the scheduler may wait for the running thread to finish its time-slice and it is uncertain that the progression thread is the next to be scheduled. Second, one time-slice may be insufficient to receive the entire message. One solution to increase the reactivity would be to use real-time threads. However, this might increase the context switching overheads since the progression thread is scheduled every time an interrupt occurs [13].

The approach most closely related to ours is described in the I/O Manager PIOMan [14] where the preemptive scheduler is able to run tasks in order to make the communication library progress. This previous work is able to efficiently overlap messages in a multi-threaded context but does not allow a MPI rank to steal tasks from another MPI rank.

2.2 Thread-Based MPI

In a thread-based MPI library, each MPI rank is a thread. All threads (MPI ranks) share the same memory address space within a unique UNIX process on a compute node. AMPI [15], AzequiaMPI [16], MPC [3], TOMPI [17], TMPI [18], USFMPI are some thread-based MPI implementations.

Because of the implicit shared-memory context among tasks, thread-based runtimes are well suited for implementing global policies, such as message progression, within a compute node. We implemented our contribution in the MPC framework, an hybrid parallelism framework exposing a thread-based MPI 1.3 runtime. According to our needs, MPC brings two main features:

- Customizable two-level thread scheduler (help for tuning the message progression strategies).
- Support for a high-speed and scalable network (access to the Infiniband network using the OFA IBverbs library with an OS-bypass technology).

3 Our Contribution: Collaborative Polling

During the execution of a parallel MPI application, the time spent while waiting for messages or collective communications is wasted. This idle time is often responsible for the poor scalability of the application on a large number of cores. Even on a well balanced application at user level, some imbalance between tasks may appear from several factors such as:

- The distance between communicating MPI peers.
- The number of neighbors.
- Micro-imbalance of communication (network links contentions, topology).
- Micro-imbalance of computation (non-deterministic events such as preemption) [5].

The main idea of the collaborative polling is to take advantage of idle cycles due to imbalance for progressing messages at the compute node level. During its unused waiting cycles, an MPI task is able to collaborate on the message progression of any other MPI task located on the same compute node. Fig. 2 compares the processing of messages arriving from a Network Interface Controller (NIC) with a regular message progression and with the collaborative polling method.

The algorithm depicted in Fig. 2 at application level is the following: each MPI task executes a non MPI-aware function (Compute) with an unbalanced workload between tasks before waiting for a message and calling a synchronization barrier. On the left part, a regular message progression is presented. On the right part, the collaborative polling method is used. Collaborative polling allows task 1 to benefit from the unused cycles while waiting its message: it can poll, receive and match messages for task 0 which is blocked into a non-interruptible computation loop. Once the computation loop is done on task 0, the expected message has already been retrieved by task 1 and the `MPI_Wait` primitive immediately returns.

As described in section 2.1 most message progression methods require to suspend the computing phase (with an interruption, an explicit call to MPI or a context switch to the progression thread) to perform progression. Collaborative polling does not require these interruptions as it only uses idle time to perform progression. Thus, the impact of collaborative polling on compute time

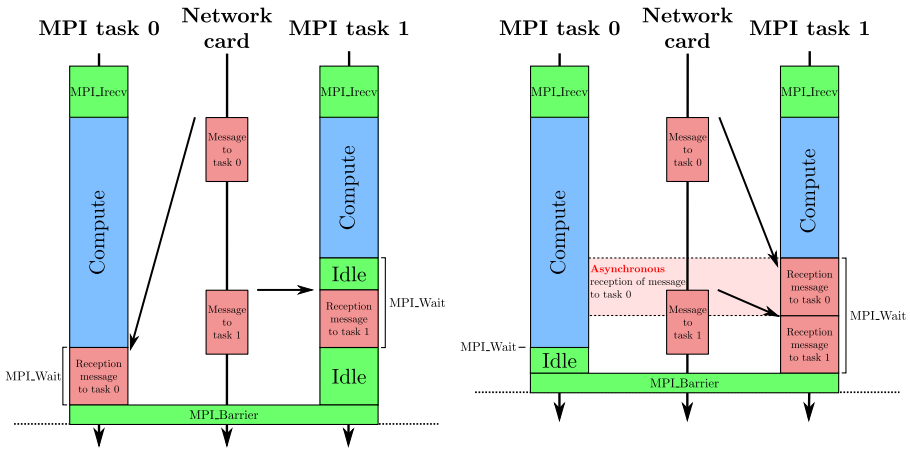

Fig. 2. MPI runtime without collaborative polling (left) and MPI with collaborative polling (right)

is reduced compared to other methods. Collaborative polling also provides an auto-adaptive polling frequency. Indeed, the frequency of calls to polling method is directly linked to the amount of tasks waiting for a communication. For example, when the number of tasks waiting on a barrier increases, the frequency of calls to the message progression method increases as well.

4 Implementation

We designed and implemented our collaborative polling approach into MPC. Since the Infiniband implementation of MPC uses the Reliable Connection (RC) service, the message order is guaranteed and messages are reliably delivered to the receiver. Three message transfer protocols are available: eager, buffered eager (split a message into several eager messages) and Rendezvous based on RDMA write. To guarantee the order across these three protocols, the high level reordering interface of MPC is in charge of sorting incoming messages.

Modern interconnects such as Infiniband usually exploit Event Queues. When a message is completed by the NIC, a new completion descriptor is posted to the corresponding completion queue (CQ). Then, the CQ is polled to read incoming descriptors and process messages. MPC implements two CQ: one for send, another for receive. Both of them are shared among tasks meaning that all notifications are received and multiplexed into the same CQ.

As depicted on Fig. 3, each MPI task implements two pending lists: one private for point-to-point messages and one global for collective operations. To ensure the message progression, the MPC scheduler calls the polling function every time a context switch occurs. The polling function is divided into three successive operations. *First* the task tries to access the CQ and returns if another task is already polling the same CQ. We limit to one the number of tasks authorized

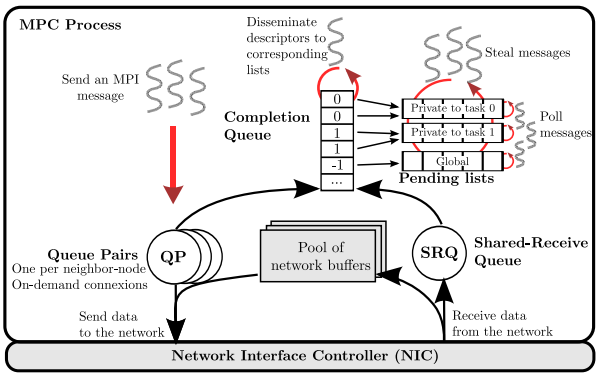

Fig. 3. Collaborative-Polling Implementation inside MPC Infiniband Module

to simultaneously poll the NIC because we observed a performance-loss with a concurrent access to the same CQ. Then, each completed Work Request (WR) found from the CQ is disseminated and enqueued to the corresponding pending list. At this time, the message is not processed. *Secondly*, the global and the private pending lists are both polled. *Thirdly*, with collaborative polling, if a task does not find any message to process, it tries to steal a WR for a task located on the same NUMA node before lastly trying another NUMA node.

4.1 Extension to Process-Based MPI

Collaborative polling requires the underlying MPI runtime to share some internal structures among tasks located on the same node. Within a regular process-based MPI runtime, collaborative polling could be implemented by mapping the same shared-memory segment in each process. The cumbersome job here is to extract the polling-related structures from the existing runtime and place them into the shared memory. Another approach would to use the Linux XPMEM Linux kernel that enables a process to expose its virutal address space to other MPI processes [19].

5 Experiments

This section presents the impact of collaborative polling on three MPI applications: EulerMHD [20], BT from the NAS Parallel Benchmark suite [21], and Gadget-2 [22] from the PRACE benchmarks. These codes run on the Curie supercomputer owned by GENCI and operated into the TGCC by CEA. This is a QDR Infiniband cluster with up to 360 nodes equipped with 4 Intel Nehalem EX processors for a total of 32 cores per node. We compare our approach (MPC w/ CP) against the regular version of MPC (MPC w/o CP), MVAPICH 1.7, Open MPI 1.5.4 and Intel MPI 4.0.3.088.

5.1 Block Tridiagonal Solver (NAS-BT)

The Block Tridiagonal Solver solves three sets of uncoupled systems of equations. It uses a balanced three-dimension domain partition in MPI and performs coarse-grained communications.

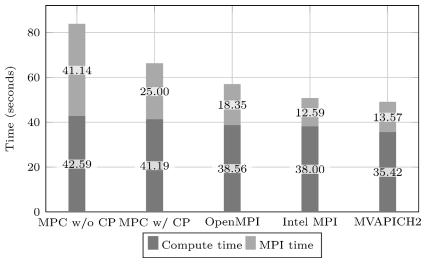

Fig. 4. BT Evaluation (class D)

Function	w/o CP	w/ CP	Speedup
Execution time	83.73	66.19	1.26
Compute time	42.59	41.19	1.03
MPI time	41.14	25	1.65
MPI_Waitall	8.06	6.6	1.22
MPI_Reduce	$1.29 \cdot 10^{-3}$	$1.28 \cdot 10^{-2}$	0.1
MPI_Allreduce	$2.39 \cdot 10^{-2}$	$6.59 \cdot 10^{-2}$	0.36
MPI_Wait	30.11	15.97	1.89
MPI_Isend	2.65	2.01	1.32
MPI_Irecv	0.29	0.33	0.88
MPI_Barrier	$4.37 \cdot 10^{-3}$	$1.53 \cdot 10^{-2}$	0.29
MPI_Bcast	$1.12 \cdot 10^{-2}$	$3.5 \cdot 10^{-3}$	3.2

Fig. 5. BT MPI Time Showdown (class D)

Figure 4 illustrates the results obtained running the BT benchmark with class D on 1024 cores on several MPI implementations. It decomposes the time spent inside the MPI runtime from the computational time. Collaborative polling allows a significant speed-up compared to regular MPC implementation. In comparison to other MPI implementations, we can however notice an overhead in MPC with collaborative polling. This is because the Message Passing layer of MPC is not well-optimized for the message sizes used by the NAS-BT benchmark in this configuration. We are currently investigating this issue.

Figure 5 exposes the details of the time spent in the MPI runtime. The gain comes from the time spent inside the *wait* functions (`MPI_Wait` and `MPI_Waitall`) because the messages have already been processed by another task when reaching such function. Indeed, Fig. 6 shows the amount of messages stolen per task (locally on the same NUMA node or remotely on another NUMA node located on the same computational node). It clearly states that the number of stolen messages is high, leading to the acceleration of the wait functions.

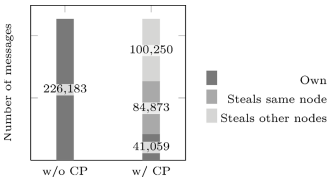

Fig. 6. Steal statistics (BT)

5.2 EulerMHD

EulerMHD is an MPI application solving both the Euler and the ideal magneto-hydrodynamics (MHD) equations at high order on a two dimensional Cartesian mesh. At each iteration, the ghost cells are packed into contiguous buffers and sent to neighbors through non-blocking calls with no-overlap capabilities. Furthermore, each timestep, a set of global reductions on one float number each is performed.

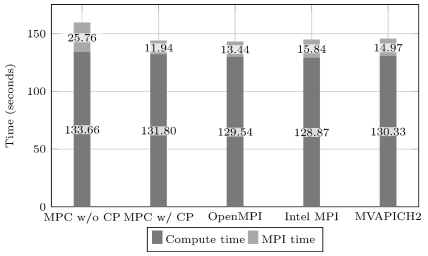

Fig. 7. EulerMHD Evaluation

Function	w/o CP	w/ CP	Speedup
Execution time	159.41	143.74	1.11
Compute time	133.66	131.8	1.01
MPI time	25.76	11.94	2.16
MPI_Allreduce	3.12	2.75	1.13
MPI_Wait	21.86	8.45	2.59
MPI_Isend	0.57	0.49	1.16
MPI_Irecv	0.21	0.24	0.87

Fig. 8. EulerMHD MPI Time Showdown

In these experiments, we use a mesh of size 4096×4096 for a total of 1024 MPI tasks and 193 timesteps. As depicted in Fig. 7, the collaborative polling decreases the time spent in MPI functions by a factor of 2. Details of time decomposition is illustrated in Table 8. The first time-consuming MPI call, the `MPI_Wait` function, shows a significant speedup by more than 2.5. The computation loop is also impacted and exhibits a minor improvement. This may be due to the polling function which is less aggressive while waiting messages with collaborative polling enabled, diminishing the memory traffic.

5.3 Gadget-2

Gadget-2 is an MPI application for cosmological N-body smoothed particle hydrodynamic simulations. At each timestep, the domain is decomposed and the work-load is balanced across MPI tasks using a combination of Allgather, Allgatherv and Ssend/Recv functions. During the force computation, each task exchanges the number of outgoing particles with a call to `MPI_Allgather` before sending a point-to-point message to each neighbor containing the new positions of the moving particles. From a task to another, the construction of the local tree differs causing an imbalanced work-load and a variation in the number of neighors. The configuration simulates $1e^7$ particles for 16 timesteps on 256 cores.

Collaborative polling exhibits an improvement in message-waiting time (see Fig. 9). Table 10 details the time acceleration of MPI functions: collaborative polling allows speed-up on `MPI_Recv` and `MPI_Sendrecv` functions leading to a 7% improvement for the MPI time compared to regular MPC run.

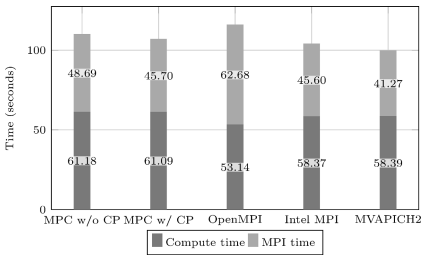

Fig. 9. Gadget Evaluation

Function	w/o CP	w/ CP	Speedup
Execution time	109.87	106.8	1.03
Compute time	61.18	61.09	1
MPI time	48.69	45.7	1.07
MPI_Reduce	1.03	0.83	1.25
MPI_Allreduce	3.81	4.24	0.9
MPI_Recv	2.62	1.31	2
MPI_Barrier	6.55	6.56	1
MPI_Bcast	0.32	0.25	1.26
MPI_Allgather	9.07	9.22	0.98
MPI_Sendrecv	6.25	5.06	1.24
MPI_Gather	$4.62 \cdot 10^{-3}$	$3.8 \cdot 10^{-3}$	1.21
MPI_Ssend	0.18	0.18	0.99
MPI_Allgatherv	18.85	18.05	1.04

Fig. 10. Gadget MPI Time Showdown

6 Conclusion and Future Work

In this paper, we proposed a transparent runtime optimization called Collaborative Polling. This solution does not require to modify the source code of the application nor the programming model. The experiments on scientific codes show a significant improvement of the MPI time with collaborative polling. Many kinds of MPI calls can benefit from this optimization: blocking/non-blocking point-to-point as well as global collectives such as barrier and allreduce. Additionally to this paper, collaborative polling was designed for MPI and Infiniband but may be extended to any programming model and any interconnect which does not implement a full independent message progression.

In the worst case of a perfectly well-balanced application, collaborative polling fails to progress message asynchronously. We plan to investigate a mixed-solution with an interrupt-based polling in a future work. We also plan to focus on hybrid MPI/OpenMP code where idle OpenMP would participate to collaborative polling and progress messages of any MPI task located on the same compute node.

Acknowledgements. This paper is a result of work performed in Exascale Computing Research Lab with support provided by CEA, GENCI, INTEL, and UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, INTEL or UVSQ. We acknowledge that the results in this paper have been achieved using the PRACE Research Infrastructure resource Curie based in France at Bruyres-le-Châtel.

References

1. Iii, J.B.W., Bova, S.W.: Where's the overlap? - an analysis of popular MPI implementations. Technical report (August 12, 1999)
2. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. IJHPCA (2005)
3. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009)

4. Bell, C., Bonachea, D., Nishtala, R., Yelick, K.A.: Optimizing bandwidth limited problems using one-sided communication and overlap. In: IPDPS (2006)
5. Subotic, V., Sancho, J.C., Labarta, J., Valero, M.: The impact of application's micro-imbalance on the communication-computation overlap. In: Parallel, Distributed and Network-based Processing (PDP) (2011)
6. Thakur, R., Gropp, W.: Open Issues in MPI Implementation. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697, pp. 327–338. Springer, Heidelberg (2007)
7. Hager, G., Jost, G., Rabenseifner, R.: Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of Cray User Group (2009)
8. Graham, R., Poole, S., Shamis, P., Bloch, G., Bloch, N., Chapman, H., Kagan, M., Shahar, A., Rabinovitz, I., Shainer, G.: Connectx-2 infiniband management queues: First investigation of the new support for network offloaded collective operations. In: International Conference on Cluster, Cloud and Grid Computing, CCGRID (2010)
9. Almási, G., Bellofatto, R., Brunheroto, J., Caşcaval, C., Castaños, J.G., Crumley, P., Erway, C.C., Lieber, D., Martorell, X., Moreira, J.E., Sahoo, R., Sanomiya, A., Ceze, L., Strauss, K.: An overview of the bluegene/L system software organization. In: Parallel Processing Letters (2003)
10. Amerson, G., Apon, A.: Implementation and design analysis of a network messaging module using virtual interface architecture. In: International Conference on Cluster Computing (2004)
11. Sur, S., Jin, H.W., Chai, L., Panda, D.K.: RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. Alternatives (2006)
12. Kumar, R., Mamidala, A.R., Koop, M.J., Santhanaraman, G., Panda, D.K.: Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 185–193. Springer, Heidelberg (2008)
13. Hoefler, T., Lumsdaine, A.: Message progression in parallel computing – to thread or not to thread? In: International Conference on Cluster Computing (2008)
14. Trahay, F., Denis, A.: A scalable and generic task scheduling system for communication libraries. In: International Conference on Cluster Computing (2009)
15. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: LCPC (2004)
16. Rico-Gallego, J.-A., Díaz-Martín, J.-C.: Performance Evaluation of Thread-Based MPI in Shared Memory. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 337–338. Springer, Heidelberg (2011)
17. 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)
18. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: International Conference on Supercomputing, ICS (2001)
19. Brightwell, R., Pedretti, K.: An intra-node implementation of openshmem using virtual address space mapping. In: Fifth Partitioned Global Address Space Conference (2011)
20. Wolff, M., Jaouen, S., Jourdren, H., Sonnendrcker, E.: High-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. Discrete and Continuous Dynamical Systems - Series S (2012)
21. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0 (1995)
22. Springel, V.: The cosmological simulation code gadget-2. Monthly Notices of the Royal Astronomical Society 364 (2005)