Computing (2014) 96:263–278
DOI 10.1007/s00607-013-0327-z

Improving MPI communication overlap with collaborative polling

Sylvain Didelot · Patrick Carribault · Marc Pérache · William Jalby

Received: 14 December 2012 / Accepted: 26 April 2013 / Published online: 9 May 2013

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. We designed our approch for Infiniband into a thread-based MPI runtime called MPC. We evaluate the gain from Collaborative Polling on the NAS Parallel Benchmarks and three scientific applications, where we show significant improvements in communication times up to a factor of 2.

S. Didelot (✉) · P. Carribault · M. Pérache · W. Jalby
Exascale Computing Research Center, Versailles, France
e-mail: sylvain.didelot@exascale-computing.eu

W. Jalby
e-mail: william.jalby@exascale-computing.eu

S. Didelot · P. Carribault · M. Pérache · W. Jalby
Université de Versailles Saint-Quentin-en-Yvelines (UVSQ), Versailles, France

P. Carribault · M. Pérache
CEA, DAM, DIF F-91297 Arpajon, France
e-mail: patrick.carribault@cea.fr

M. Pérache
e-mail: marc.perache@cea.fr

Keywords HPC · Overlap · MPI · High-speed network · Polling

Mathematics Subject Classification 68N19 · 68N15

1 Introduction

The scalability of a parallel application is mainly driven by the amount of time 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, Fig. 1a 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. 1b).

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 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 do 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.

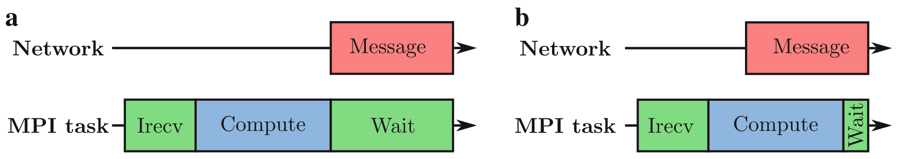

Fig. 1 Influence of communication/computation overlapping in MPI, **a** no overlapping, **b** overlapping

2 Related work

2.1 Message progression strategies

Previous work has shown 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 where an additional user-thread is created and blocked inside a `MPI_Recv` function. Hager et al. [7] investigate an 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, FG-MPI[9] extends the MPICH2 runtime and allows over-subscribed and non-preemptive MPI threads to share the same MPICH2 process. The proposed solution however limits the message progression strategy to a physical core whereas collaborative polling enables it at the compute node level. MPI libraries also investigate a threaded message 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 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 [10]. 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 where the same core is shared between the progression thread and the user thread. However the decision when and how often the polling function should be called is non-trivial. Too many calls may cause an 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 [11]. 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 [12,13]. 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

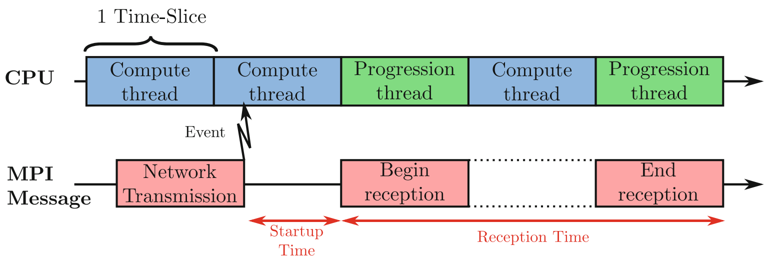

Fig. 2 Overheads in a threaded message progression

ms. 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 poll, match and, if needed, recopy the network message to the end-user buffer. These overheads are sequentially denoted as "Startup Time" and "Reception Time" on Fig. 2. 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 [14].

This work extends a previous study published by Didelot et al. [15]. In the following paper, we investigate deeper the gain of collaborative polling while experimenting it on more applications. The approach most closely related to ours is described in the I/O Manager PIOMan [16] 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 [17], AzequiaMPI [18], FG-MPI[9], MPC [3], TOMPI [19], TMPI [20], 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, a hybrid parallelism framework exposing a thread-based MPI 1.3 runtime. The supported programming languages are C, C++ and Fortan. According to our needs, MPC brings three following features:

– A customizable two-level thread scheduler. It helps for tuning the message progression strategies.

- A support for a high-speed and scalable network. It provides an access to Infiniband networks using the OF Verbs library with an OS-bypass technology.
- An automatic privatization of user's global variables to thread-private variables using a patched version of GCC [21].

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: inter/intra-node communications, number of network hops.
- 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. Figure 3 compares the processing of messages arriving from a Network Interface Controller (NIC) with a regular message progression and with the collaborative polling method.

Figure 3 depicts an MPI application performing the following algorithm: 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

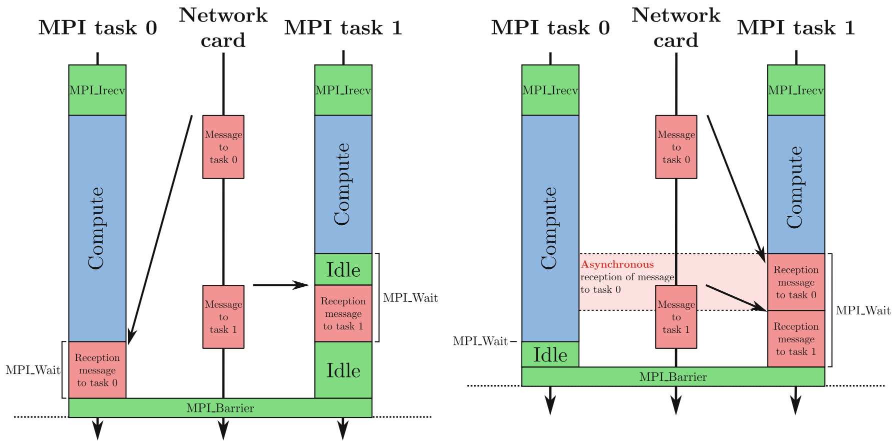

Fig. 3 MPI runtime without collaborative polling (*left*) and MPI with collaborative polling (*right*)

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 Sect. 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 is reduced compared to other methods. Collaborative polling also provides an auto-adaptive polling frequency. Indeed, the frequency of calls to the polling function is correlated with 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. 4, each MPI task implements one private pending list for point-to-point messages. An additional global pending list is dedicated to collective operations and may be concurrently accessed by several tasks. 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 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. If messages reside in the lists, they are processed until an expected MPI message is found. *Thirdly*, with collaborative polling, if a task does not find any message to match, it tries to steal a WR from a task located on the same NUMA node before lastly trying another NUMA node.

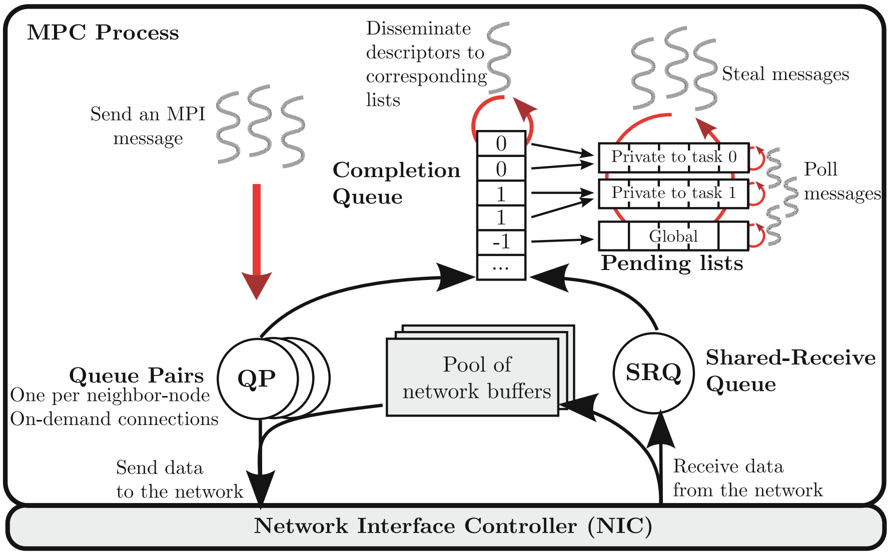

Fig. 4 Collaborative-polling implementation inside MPC Infiniband Module

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 first cumbersome job here is to extract the polling-related structures from the existing runtime and place them into the shared memory.

The second difficulty is to by-pass the OS security which prevents several processes to share the same network endpoint. For Infiniband, the Protection Domain (PD) provides an increased level of protection against inadvertent and unauthorized accesses: a process cannot affect a QP in a different Protection Domain. As far as we know, two processes cannot share the same PD and the compliance C10-7 from the Infiniband Architecture Specification [22] requires that each QP in an HCA shall be associated with a unique Protection Domain. To address this issue, we propose an implementation guideline where the runtime spawns and pins for each process as many POSIX threads as physical cores on the compute node. When an MPI task is idle, it can wake and schedule a thread from another process running the same core than it. The newly scheduled thread then may call the progression function and handle incoming messages. This approach however requires $O(p)$ threads to be scheduled on each core where p is the number of processors on the compute node. An alternative approach would be to use the Linux XPMEM Kernel module that enables a process to expose its virtual address space to other MPI processes [23]. Since installing an external kernel module on an HPC center is discouraged for security reasons, we did not focus on this solution.

4.2 Extension to other high-speed interconnects

For the following paper, we designed collaborative polling for Infiniband networks. However, this approach would be implemented for any interconnect, in condition that the HCA does not support a fully independent message progression. In the case of MPI over Infiniband, computation parts such as message matching cannot be offloaded to the HCA and require the involvement of the host CPU to complete the reception. In addition, collaborative polling does not require the underlying network to support communication offload but should be more efficient on such networks.

5 Experiments

This section presents the impact of collaborative polling on three MPI applications: EulerMHD [24], the NAS Parallel Benchmark suite [25], and Gadget-2 [26] 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 X7560 processors clocked at 2.266 GHz, 128 GB of main memory, for a total of 32 cores per node. We compare our approach (MPC CP) against the regular version of MPC (MPC), MVAPICH2 1.7 (MV2), Open MPI 1.6.1 (OMPI) and Intel MPI 4.0.3.088 (IMPI) which is based on the MPICH runtime. Both, the application and the runtimes have been compiled using GNU GCC 4.4.0 and same compilation flags, except for the MPI runtime from Intel. The results are an average of three runs and the same nodes have been used for comparing the different runtimes.

5.1 NAS parallel benchmarks

The NAS parallel benchmarks (NPBs) are a collection of MPI applications that are distilled from real computational fluid dynamics applications. We omitted the EP benchmark from our study as it is exchanging a negligible number of MPI messages.

Figure 5 illustrates the results obtained running the NAS SP, MG, BT, FT, CG and IS with class D on 1,024 cores on several MPI implementations. It decomposes the time spent inside the MPI runtime from the computational time. For SP, MG and BT, collaborative polling significantly reduces the time in MPI communications. Apart from Intel MPI on MG where `MPI_Wait` and `MPI_Barrier` functions slow down the execution time, collaborative polling provides performance close to the related work. It respectively gives a speedup of 1.34, 1.25 and 1.69 on the communication time for SP, MG and BT. Figure 6 depicts how much time is spent for an MPI task to retrieve its own messages as well as to steal and process messages from another task. For these three benchmarks, we observe a large amount of time stolen by MPI tasks. It causes a significant reduction of the time spent for a task to receive its own messages. We also notice a slight overhead in the message processing. Since we do not have sufficient permissions on the cluster to access the hardware counters, we can only assume that this effect is due to NUMA effects. Indeed, the copy of network buffers to end-user buffers is more costly when it is processed by an MPI task located on a

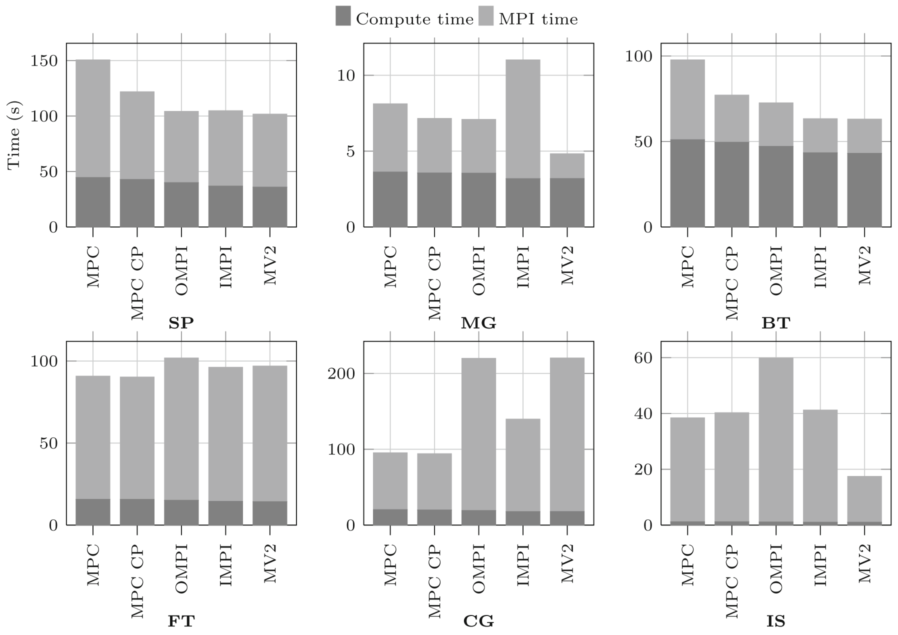

Fig. 5 NPB MPI evaluation. Class D on 1,024 cores

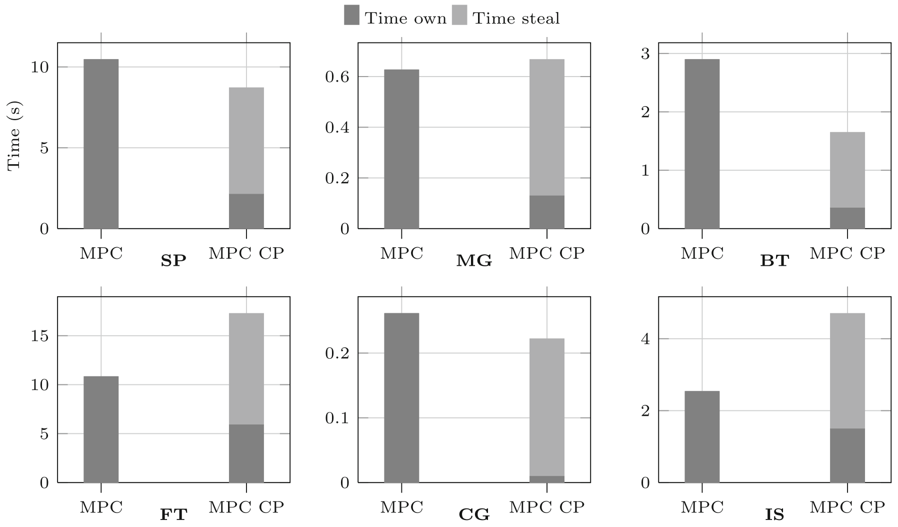

Fig. 6 NPB steal statistics. Class D on 1,024 cores

different NUMA node than the node where the end-user buffer is posted. However, this overhead does not negatively affect the total execution time as the message processing occurs during idle time.

On NAS CG, MPC with collaborative polling behaves like the regular version of MPC. Some messages are stolen but the stolen time does not accelerate the execution of

Table 1 BT MPI time showdown (class D)

Function	MPC	MPC CP	Speedup
Execution time	97.69	77.09	1.27
MPI time	46.70	27.58	1.69
Compute time	50.99	49.51	1.03
MPI_Wait	30.58	12.73	2.40
MPI_Waitall	12.59	12.47	1.01
MPI_Isend	1.22	1.33	0.92
MPI_Irecv	1.83	0.67	2.75

the application, probably because the workload is well balanced across the tasks. The same benchmark shows an overhead for Open MPI and MVAPICH2 due to a slowdown in the `MPI_Send` function. On the other hand, NAS FT and IS exhibit an overhead using collaborative polling. These benchmarks mostly communicate using collective operations like `MPI_AlltoAll`, `MPI_AlltoAllv` and `MPI_Allreduce`. The MPC implementation of collective operations uses point-to-point messages and tree-like communications. Collective communication patterns consist of multiple communication stages (a.k.a *rounds*): let assume a communication round k, each MPI task waits a message from the taks involved in round $k-1$ after sending a message to the tasks of round $k+1$. When a task steals a message from a collective operation, it does not emit the messages corresponding to the next round of the stolen task. In this case, collaborative polling cannot benefit from idle time to recover the time lost while stealing messages. A look at Open MPI and MVAPICH2 shows that, in this configuration, they are both penalized on CG because of an high amount of time spent in `MPI_Send`. Furthermore, Open MPI gets an high overhead in `MPI_AlltoAllv` on IS.

5.1.1 Block tridiagonal solver (NAS-BT)

In this section, we focus on the Block Tridiagonal Solver (BT). This benchmark solves three sets of uncoupled systems of equations. It uses a balanced three-dimensional domain partition in MPI and performs coarse-grained communications.

Table 1 exposes the details of the time spent in the MPI runtime. The gain in MPI time 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. 7 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 shows that the number of stolen messages is high, leading to the acceleration of the wait functions.

5.2 EulerMHD

EulerMHD is an MPI application solving both the Euler and the ideal magnetohydrodynamics (MHD) equations at high order on a two dimensional Cartesian mesh. At

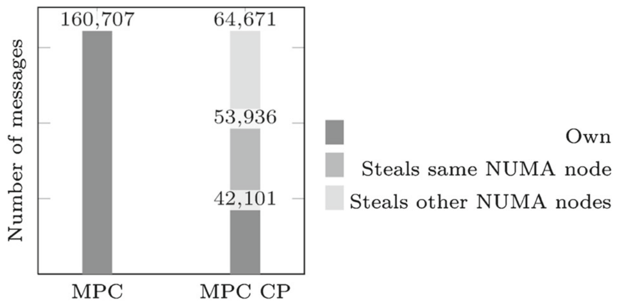

Fig. 7 Steal statistics (BT)

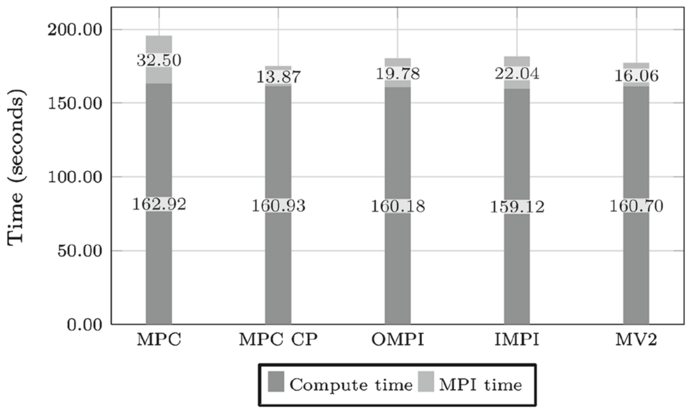

Fig. 8 EulerMHD evaluation

Table 2 EulerMHD MPI time showdown

Function	MPC	MPC CP	Speedup
Execution time	195.43	174.80	1.12
MPI time	32.50	13.87	2.34
Compute time	162.92	160.93	1.01
MPI_Wait	26.27	10.36	2.53
MPI_Allreduce	4.17	2.63	1.58
MPI_Irecv	1.24	0.18	6.84
MPI_Isend	0.83	0.69	1.19

each iteration, the ghost cells are manually 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.

In these experiments, we use a mesh of size 4,096 $\times$ 4,096 for a total of 1,024 MPI tasks and 193 timesteps. As depicted in Fig. 8, the collaborative polling decreases the time spent in MPI functions by a factor of 2. Details of time decomposition is illustrated in Table 2. The first time-consuming MPI call, the `MPI_Wait` function,

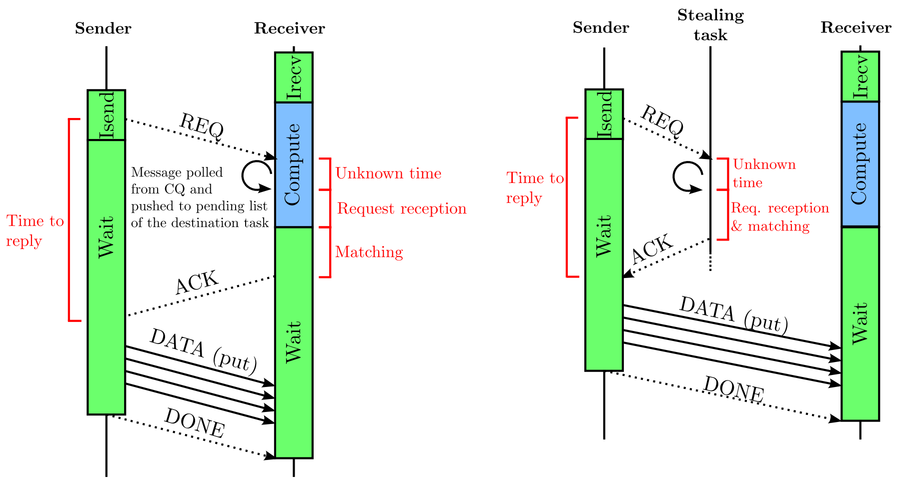

Fig. 9 The Rendezvous protocol with collaborative polling (*left*) and without (*right*). With collaborative polling, an idle MPI task may steal a rendezvous control message, match and send the `ACK` to the sender

shows a significant speedup by more than 2.5. Surprisingly, the `MPI_Allreduce` function highlights a speedup of 1.58 in this application. It can be easily explained: with collaborative polling, faster MPI tasks already inside `MPI_Allreduce` may help the progression of tasks that did not yet reach this function. Thus, collaborative polling aims to diminish the imbalance across MPI tasks and so the time in global synchronization points such as `MPI_Allreduce`.

The computation loop is also impacted and exhibits a minor improvement. With collaborative polling enabled, the polling function is less aggressive while waiting messages. This aims to reduce the overall memory traffic.

5.2.1 The Rendezvous protocol

MPC implements the Rendezvous protocol which requires a two-sided synchronization between the sender and the receiver. It combines a lazy deregistration and a registration cache to re-use existing registered addresses and reduce the impact of memory registration [27]. In addition, no intermediate copy is allocated, meaning that the receiver waits the receive buffer to be posted before sending the `ACK` message and proceeding to a RDMA write operation.

Figure 9, left part, depicts the reception of a Rendezvous message without collaborative polling. While computing, the receiver cannot handle the `REQ` message. As a result, the matching and the reply only occur inside the `wait` function. With collaborative polling (right part), an idle MPI task may steal the `REQ` message and reply the `ACK` to the sender. The message transfer can even begin whereas the receiver is still computing.

We run EulerMHD with the same dataset as previous but we disable the Buffered protocol and force MPC to switch to Rendezvous protocol. In this configuration, 97 %

Fig. 10 EulerMHD evaluation

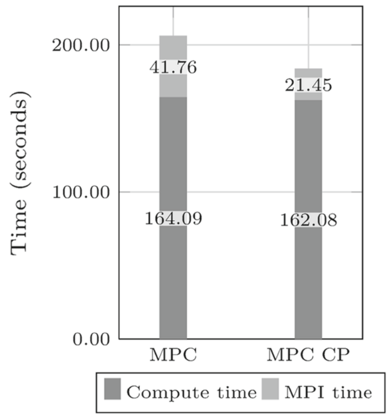

Table 3 EulerMHD rendezvous timers

Function	MPC	MPC CP
Time to reply	27.68	9.98
Matching	13.56	5.50
Request reception	6.27	0.08

of MPI messages are exchanged using Rendezvous. Figure 10 decomposes the time spent inside the MPI runtime from the computational time and it clearly shows that collaborative polling reduces the time to communicate. For a depth investigation, the Rendezvous interface of MPC has been instrumented with three timers:

1. *Time to reply* time between the `REQ` and the `ACK` messages at the sender side.
2. *Request reception* time to handle the message while it as already been polled from the CQ at the receiver side.
3. *Matching* time to match the message at the receiver side. This timer also includes the time spent while waiting for the matching function to execute.

Table 3 reports the results of these timers on EulerMHD with and without collaborative polling. At the sender side, the time to reply expresses a speedup of 2,77 using collaborative polling. At the receiver side, because an idle task may handle the `REQ` message from a computing task immediately after it has been polled from the CQ, the request reception time is significantly faster with collaborative polling. Since collaborative polling allows multiple tasks to handle messages for the same remote task, several Rendezvous messages can be matched in parallel, reducing the time required for matching messages.

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

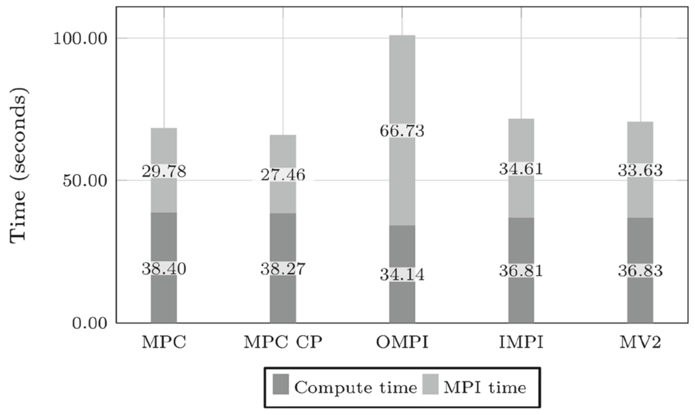

Fig. 11 Gadget evaluation

Table 4 Gadget MPI time showdown

Function	MPC	MPC CP	Speedup
Execution time	68.18	65.73	1.04
MPI time	29.78	27.46	1.08
Compute time	38.40	38.27	1.00
MPI_Allgatherv	9.51	8.86	1.07
MPI_Allgather	9.34	8.41	1.11
MPI_Sendrecv	3.75	3.47	1.08
MPI_Barrier	3.06	3.04	1.01
MPI_Allreduce	2.03	2.12	0.95
MPI_Recv	0.91	0.69	1.31
MPI_Reduce	0.76	0.52	1.47
MPI_Bcast	0.19	0.15	1.29
MPI_Ssend	0.14	0.14	0.99

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. 11). Open MPI gets an abnormal slow-down of approximatively 10 on the `MPI_Allreduce` function compared to the other runtimes. Table 4 details the time acceleration of MPI functions: collaborative polling allows speed-up on `MPI_Recv` and `MPI_Sendrecv` functions leading to a 8 % improvement for the MPI time compared to regular MPC run.

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. Regular blocking/non-blocking point-to-point communications can benefit from this optimization. Collaborative polling may also reduce the imbalance across MPI tasks, diminishing the idle time spent inside global collective operations like barrier, alltoall 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 intend to focus on hybrid MPI/OpenMP codes where idle OpenMP tasks (i.e: tasks blocked in a barrier) would participate to collaborative polling and progress messages of any MPI task located on the same compute node.

Acknowledgments 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 Bruyères-le-Châtel.

References

1. Iii JBW, Bova SW (1999) Where's the overlap? An analysis of popular MPI implementations. Technical report (August 12 1999)
2. Brightwell R, Riesen R, Underwood KD (2005) Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. IJHPCA
3. Pérache M, Carribault P, Jourdren H (2009) MPC-MPI: an MPIimplementation reducing the overall memory consumption. In: PVM/MPI
4. Bell C, Bonachea D, Nishtala R, Yelick KA (2006) Optimizing bandwidth limited problems using one-sided communication and overlap. In: IPDPS
5. Subotic V, Sancho JC, Labarta J, Valero M (2011) The impact of application's micro-imbalance on the communication-computation overlap. In: Parallel, distributed and network-based processing (PDP)
6. Thakur R, Gropp W (2007) Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In: PVM/MPI. pp 46–55
7. Hager G, Jost G, Rabenseifner R (2009) Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of Cray User Group
8. Graham R, Poole S, Shamis P, Bloch G, Bloch N, Chapman H, Kagan M, Shahar A, Rabinovitz I, Shainer G (2010) 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)
9. Kamal H, Wagner A (2012) Added concurrency to improve MPI performance on multicore. In: ICPP, IEEE Computer Society, pp 229–238
10. Almási G, Bellofatto R, Brunheroto J, Caşcaval C, Castaños JG, Crumley P, Erway CC, Lieber D, Martorell X, Moreira JE, Sahoo R, Sanomiya A, Ceze L, Strauss K (2003) An overview of the bluegene/L system software organization. Parallel Process Lett

11. Amerson G, Apon a (2004) Implementation and design analysis of a network messaging module using virtual interface architecture. In: International conference on cluster computing
12. Sur S, Jin Hw, Chai L, Panda DK (2006) RDMA read based Rendezvous protocol for MPI over infiniBand: design alternatives and benefits. Alternatives
13. Kumar R, Mamidala AR, Koop MJ, Santhanaraman G, Panda DK (2008) Lock-free asynchronous rendezvous design for MPI point-to-point communication. In: PVM/MPI
14. Hoefler T, Lumsdaine A (2008) Message progression in parallel computing to thread or not to thread?. In: International conference on cluster computing
15. Didelot S, Carribault P, Pérache M, Jalby W (2012) Improving MPI communication overlap with collaborative polling. In: EuroMPI
16. Trahay F, Denis A (2009) A scalable and generic task scheduling system for communication libraries. In: International conference on cluster computing
17. Huang C, Lawlor O, Kalé LV (2004) Adaptive MPI. In: LCPC
18. Rico-Gallego JA, Martín JCD (2011) Performance evaluation of thread-based MPI in shared memory. In: EuroMPI
19. Demaine E (1997) A threads-only MPI implementation for the development of parallel programming. In: Proceedings of the 11th international symposium on high performance computing systems
20. Tang H, Yang T (2001) Optimizing threaded MPI execution on SMP clusters. In: International Conference on Supercomputing (ICS)
21. Carribault P, Pérache M, Jourdren H (2011) Thread-local storage extension to support thread-based MPI/openMP applications. In: Chapman BM, Gropp WD, Kumaran K, Müller MS (eds) IWOMP. Lecturen notes in computer science. Springer, Berlin, pp 80–93
22. InfiniBand Trade Association: InfiniBand architecture specification
23. Brightwell R, Pedretti K (2011) An intra-node implementation of openshmem using virtual address space mapping. In: Fifth partitioned global address space conference
24. Wolff M, Jaouen S, Jourdren H, Sonnendrcker E (2012) High-orderdimensionally split lagrange-remap schemes for idealmagnetohydrodynamics. Discrete and Continuous Dynamical Systems -Series S
25. Bailey D, Harris T, Saphir W, van der Wijngaart R, Woo A,Yarrow M (1995) The NAS Parallel Benchmarks 2.0
26. Springel V (2005) The cosmological simulation code gadget-2. Monthly Notices of the Royal Astronomical Society 364
27. Tezuka H, O'Carroll F, Hori A, Ishikawa Y (1998) Pin-down cache: A virtual memory management technique for zero-copy communication. In: IPPS/SPDP, pp 308–314