Parallel Computing xxx (xxxx) xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Collectives in hybrid MPI+MPI code: Design, practice and performance

Huan Zhou *, José Gracia, Naweiluo Zhou, Ralf Schneider

High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 70569 Stuttgart, Germany

ARTICLE INFO

Keywords:
MPI
MPI shared memory model
Collective communication
Hybrid programming

ABSTRACT

The use of hybrid scheme combining the message passing programming models for inter-node parallelism and the shared memory programming models for node-level parallelism is widely spread. Existing extensive practices on hybrid Message Passing Interface (MPI) plus Open Multi-Processing (OpenMP) programming account for its popularity. Nevertheless, strong programming efforts are required to gain performance benefits from the MPI+OpenMP code. An emerging hybrid method that combines MPI and the MPI shared memory model (MPI+MPI) is promising. However, writing an efficient hybrid MPI+MPI program – especially when the collective communication operations are involved – is not to be taken for granted.

In this paper, we propose a new design method to implement hybrid MPI+MPI context-based collective communication operations. Our method avoids on-node memory replications (on-node communication overheads) that are required by semantics in pure MPI. We also offer wrapper primitives hiding all the design details from users, which comes with practices on how to structure hybrid MPI+MPI code with these primitives. Further, the on-node synchronization scheme required by our method/collectives gets optimized. The micro-benchmarks show that our collectives are comparable or superior to those in pure MPI context. We have further validated the effectiveness of the hybrid MPI+MPI model (which uses our wrapper primitives) in three computational kernels, by comparison to the pure MPI and hybrid MPI+OpenMP models.

1. Introduction

For decades the Message Passing Interface (MPI) [1] has been a dominant parallel programming model in the area of high-performance computing (HPC). It is widely utilized by applications of interest to various fields and will continue to be prosperous for its efficiency, adaptivity, and portability. Nowadays, the computational capability of a single processor grows in a way that increases its number of computational cores, which strengthens the hierarchical memory structure (shared memory within nodes and message passing across nodes). Memory technology, however, lags behind processor technology. This dilutes per-core-memory in the current commodity supercomputers. Traditionally, the applications that are written in pure MPI may face two problems. One is the latency, where extra memory copings are internally required by MPI semantics, and the other is the memory utilization, where some (on-node) copies of replicated data are needed when memory is partitioned across multiple cores for separate address space. In this regard, the reduced per-core-memory is abused. Partitioned Global Address Space (PGAS) and hybrid programming models could be the solutions to the above two problems. PGAS, such as Unified Parallel C (UPC) [2] and OpenSHMEM [3], provides convenient access to shared global address space. However, migrating existing MPI parallel programs to another PGAS model will burden the users with a large amount of rewriting work. Conversely, the hybrid model offers an incremental pathway to extend existing MPI programs by combining MPI (inter-node parallelism) and a shared memory programming approach (node-level parallelism). Open Multi-Processing (OpenMP) [4] is the most frequently-used shared memory programming model [5]. The simplest approach is to incrementally add OpenMP directives to the computationally-intensive parts of the existing MPI code, which is also called OpenMP fine-grained parallelism [6]. This approach can produce serial sections that are only executed by the master thread. Coupled with the extra overheads from shared memory threading, such hybrid implementation may hardly outperform the pure MPI implementation when the scaling of the MPI implementation is still good [7,8]. When the scalability of the pure MPI code suffers a lot, the hybrid one could perform better with less communication time [9,10]. There are still two new hybrid parallel programming methods: MPI+UPC [11] and MPI+OpenSHMEM [12]. They attract little attention, since a profound grasp of both MPI and OpenSHMEM or UPC is needed to write efficient applications.

Further, an innovative hybrid programming approach combining MPI and the MPI Shared Memory (SHM) model emerges (MPI+MPI [13]). The MPI SHM model [14–17] is process-based and introduced in the MPI-3 standard for supporting shared memory address

* Corresponding author.
E-mail addresses: huan.zhou@hlrs.de (H. Zhou), gracia@hlrs.de (J. Gracia), naweiluo.zhou@hlrs.de (N. Zhou), schneider@hlrs.de (R. Schneider).

https://doi.org/10.1016/j.parco.2020.102669
Received 8 November 2019; Received in revised form 13 July 2020; Accepted 20 July 2020
Available online 24 July 2020

space among MPI processes on the same node. In the hybrid MPI+MPI model, the on-node shared data is logically partitioned and a portion of it is affinity to each process. Compared with the MPI model, the computational parallelism in the MPI+MPI version stays unchanged and the on-node communication overhead is eliminated. Therefore, this hybrid scheme is expected to benefit performance, even when the pure MPI applications are already good in scalability. Nevertheless, there are so far very limited practices to guide the users in writing scalable as well as efficient hybrid MPI+MPI applications, except the study [13] that demonstrates a hybrid MPI+MPI programming paradigm featuring the point-to-point communication operations (e.g., halo exchanges). However, this paradigm does not strictly follow the shared memory programming scheme that demands only one (shared) copy of replicated data among on-node processes. Besides the point-to-point communication operations, MPI provides a rich suite of collective operations that involve a group of processes rather than a pair of processes. The MPI collectives are important, as they are frequently invoked in a spectrum of scientific applications or kernels [18]. They always appear in performance-critical sections of these applications. If the hybrid MPI+MPI code continues to harness the standard MPI collectives as the pure MPI code does, scalable performance is difficult to achieve. Therefore, designing hybrid MPI+MPI context-based collective communication operations and creating experience in writing scalable and efficient hybrid MPI+MPI programs including these collectives are inspired.

In the pure MPI version, the collectives give a copy of the result to every on-node process, which is dispensable in the hybrid MPI+MPI version when each process proceeds to read the result with *visible* or no change to it. This is the case in most of the existing applications or kernels containing the collective operations [18] and the benchmarks used in this paper as well. The *visible* change, as the name implies, the change is visible to other processes (shared between processes). Further, the *visible* changes to the same data can be synchronized by using the method proposed in [19]. Conversely, *invisible* change signifies private change, which entails a copy of the accessed data. Previously, we discussed the programmatic differences between the approach of collectives in the hybrid MPI+MPI context with the standard one in the pure MPI context [20]. In this paper, we (take *allgather*, broadcast and *allreduce* as concrete cases) further explore the challenges associated with designing the hybrid MPI+MPI context-based collective operations and writing an efficient hybrid MPI+MPI code with an acceptable number of lines. The main contributions of our work are fourfold:

1. Besides *allgather* and broadcast, we describe the design method of *allreduce* in the hybrid MPI+MPI context. We provide the users with fully-functional MPI wrappers that hide all the design details of our collectives and demonstrate the necessity of applying these wrappers to the hybrid MPI+MPI programming.
2. We highlight all synchronization points that are inherently required by our *allgather*, broadcast and *allreduce* to guarantee data integrity inside nodes. We then discuss how to implement them with minimal overhead.
3. We perform a series of micro-benchmarks to first quantify the implementation overhead brought by our collectives and then compare our collectives and the standard MPI collectives, under the same distribution of workload on all cores.
4. We conduct three case studies to show the benefit of the hybrid MPI+MPI code calling our collectives over the pure MPI and hybrid MPI+OpenMP code.

The paper is organized as follows. In the next section, we briefly give the related work. Section 3 describes the MPI SHM model that forms a basis for our hybrid MPI+MPI context-based collectives, and the skeleton of a simple hybrid MPI+MPI program. Section 4 starts with a description of our collectives, provides the users with wrapper functions, presents examples written in the hybrid MPI+MPI context and proposes a relatively light-weight synchronization method. In Section 5, the experimental results and analyses based on the micro- and kernel-level benchmarks are demonstrated. Section 6 discusses and concludes our paper.

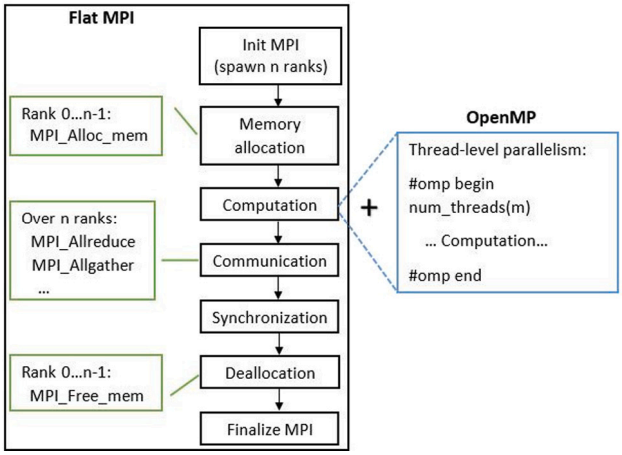


Fig. 1. The workflow of the hybrid MPI+OpenMP programming model with the fine-grained on-node parallelization approach.

2. Related work

In the early stage of optimizing MPI collective operations, the researchers put much effort into studying optimal algorithms. This leads to the coexistence between different algorithms catering to different message sizes and numbers of processes [21,22]. The high-performance implementations of MPI, such as MPICH [23], Intel MPI [24] and Open MPI [25], thus choose the most appropriate algorithm to use at runtime.

The prevalence of clusters of shared memory nodes highlights a hybrid architecture combining distributed (across nodes) and shared memory (constrained within a single node). The optimized works have shifted to distinguish between intra-node and inter-node communication (aka., hierarchical algorithm [26–28]). The hierarchical algorithm has been adopted by the existing MPI implementations and pays off. MPI collectives are expected to be highly tuned for shared memory as well as distributed architecture. In [29,30], the MPI collectives are optimized by using the shared cache as an intra-node data transfer layer. Nowadays, a typical shared memory node features non-uniform memory access (NUMA) architecture, the NUMA-aware shared memory MPI collectives are thus proposed to further minimize the inter-NUMA (intersocket) memory traffic [31]. Besides that, remote direct memory access (RDMA) is used for inter-node communication to improve performance [32,33]. All the aforementioned optimizations pave the way to the maturity of MPI collectives. Furthermore, the scalability of MPI+OpenMP hybrid application has been improved by making full use of idle OpenMP threads to parallelize the MPI collectives [34].

3. MPI+OpenMP versus MPI+MPI

In this section we describe the MPI-3 shared memory model and two-level of communicator splitting. They are foundations of the hybrid MPI+MPI programming model. We further provide a brief comparison between two skeleton programs of MPI+OpenMP and MPI+MPI containing collective communication operations.

3.1. MPI+OpenMP

In the hybrid MPI+OpenMP hybrid model, MPI is used for communication across distributed memory nodes and OpenMP is responsible for on-node parallelization. Assuming there is a cluster of n shared memory nodes, each of which consists of m computational cores. Fig. 1

illustrates the workflow of writing a hybrid MPI+OpenMP program (run on the n nodes) by using the OpenMP fine-grained parallelization approach [6]. At the period of initialization, n MPI processes are spawned and each of them is allocated on distinct node. The left part shows that each MPI process allocates or frees memory which has its own address space (not addressable by each other). The computation component will resort to OpenMP directives, which is demonstrated in the right part of the figure. Here, each process spawns m threads (each thread is pinned to a core) executing the computation concurrently. In this scenario, the standard MPI collective communication operations over the n MPI processes are directly harnessed. The advantage of OpenMP offering incremental approach towards parallelization facilitates the porting work from MPI code to hybrid MPI+OpenMP one. In the hybrid MPI+OpenMP version, however, creating the same parallelism as in the MPI version is daunting and needs plenty of human efforts, which will, in turn, reduce the advantage in using OpenMP. Besides the fine-grained parallelism, another parallelism approach of coarse-grained is also studied but not so mature as the former. Therefore, the hybrid MPI+OpenMP program with fine-grained parallelism is considered as one of the baselines for evaluating the hybrid MPI+MPI program in Section 5.3.

3.2. MPI+MPI

MPI-3 extends the standard MPI with the shared memory programming that supports direct load/store operations on a single node. In this section, we introduce the shared memory and bridge communicators. The concept of the shared memory window is also critical for us to understand how MPI SHM exposes a global view of on-node memory to the users. With all these knowledge in hand, we introduce the workflow of writing a typical hybrid MPI+MPI program running on the aforementioned cluster of n nodes.

3.2.1. Two level of communicator splitting

The function *MPI_Comm_split_type* is called with the parameter of *MPI_COMM_TYPE_SHARED* to divide the communicator into discrete node-level communicators. Each node-level (aka. shared memory) communicator identifies a group of processes that are connected to the same shared memory system, inside which all processes can perform load/store operations instead of explicit remote memory access (RMA). Besides the shared memory communicator, the hybrid MPI+MPI programming model entails an across-node communicator to serve for the explicit communication between processes residing on different nodes. A process per node (often with the lowest rank) is chosen as a *leader* to take responsibility for the data exchanges across nodes, while the other on-node processes are viewed as its *children*. The across-node communicator acts as a bridge between nodes and thus is also called bridge communicator [28], which is formed by calling *MPI_Comm_split*.

3.2.2. MPI shared memory window

The usage of *MPI_Win_allocate_shared* is crucial to create a window spanning a region of addressable shared memory with an individual size that is contributed by each on-node process. By default, the memory in a window is allocated contiguously in hardware. Creating non-contiguous memory is also possible when the parameter of *alloc_shared_noncontig* is set to true. The function *MPI_Win_shared_query* is used to obtain the base pointer to the beginning of the shared memory segment contributed by another process. This base pointer allows the allocated memory to be accessed with immediate load/store instructions by all on-node processes. Intuitively, the function *MPI_Win_sync* is defined to synchronize between the private and public window copies. Nowadays, the majority of hardware architectures feature a unified memory model, where the public and private copies can be maintained consistent implicitly. Nevertheless, the usage of *MPI_Win_sync* is still valuable in achieving a memory synchronization when there are concurrent accesses to the same memory location by different on-node processes.

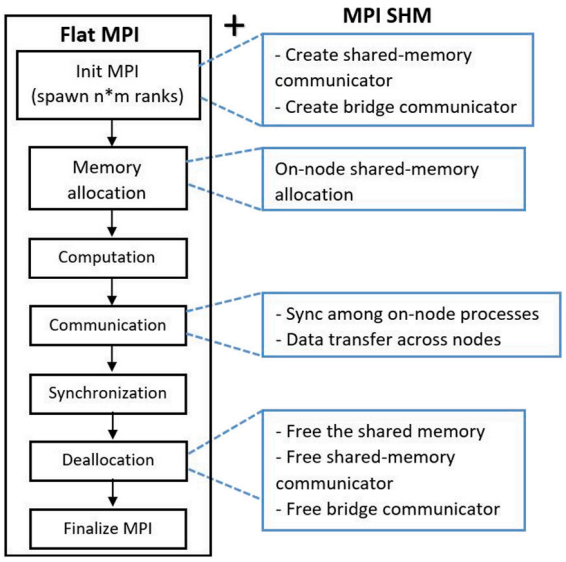


Fig. 2. The workflow of the hybrid MPI+MPI programming model.

3.2.3. Workflow

Theoretically, the migration from pure MPI programs to hybrid MPI+MPI ones should be smooth due to their interoperability. Fig. 2 presents the hybrid MPI+MPI programming pattern, where the right part presents the possible rewriting efforts for achieving this hybridization. Here, $n * m$ (equal to the number of available cores) MPI processes are spawned during initialization, where the shared memory and bridge communicators are required to be generated. The *leader* allocates the entire shared memory region for all on-node processes and then its *children* attach to a separate portion of this shared memory region. When a global communication operation happens, the shared region can be accessed by executing load/store instructions, with all the node-level synchronizations to guarantee its consistent status among on-node processes. This shared region can also certainly be touched by the processes residing on different nodes via RMA, collective and point-to-point communication operations. Before finalizing the program, the above-mentioned two communicators and the shared region should be deallocated.

Our comparison between hybrid MPI+MPI and pure MPI or hybrid MPI+OpenMP consists in two aspects: programmability and performance. In detail, preparing the aforementioned communicators and shared memory windows for a hybrid MPI+MPI program can be tedious. Moreover, a hybrid MPI+MPI program is error-prone when the users mishandle the node-level synchronization operations. The rewriting efforts are clearly not negligible. Therefore, wrapper functions encapsulating these rewriting details should be available to the users for well-modularized programming. Like MPI, the problems/tasks are also forced to be decomposed and evenly assigned to separate processes for locality in hybrid MPI+MPI. Hence unequal parallelism will not be the reason for the performance benefits of MPI over hybrid MPI+MPI. Besides measuring the performance of our collectives, their implementation overheads need to be considered when the holistic performance of a hybrid MPI+MPI program is assessed.

4. Implementation and practices

In this section, we present several generic wrapper interfaces that should always be included to enable a hybrid MPI+MPI program with the pattern shown in Fig. 2. We take three typical collectives (*MPI_Allgather*, *MPI_Allreduce*, and *MPI_Bcast*) for example, to describe our efforts in implementing the hybrid MPI+MPI context-based collectives by assuming that the block-style rank placement is employed.

```
/* The structure of data type comm_package */
struct comm_package
{
  MPI_Comm  shmem_comm;
  MPI_Comm  bridge_comm;
  int   shmemcomm_size;//Size of shared memory communicator
  int   bridgecomm_size;//Size of bridge communicator
};
/* Two level of communicator splitting */
void Wrapper_MPI_ShmemBridgeComm_create(MPI_Comm par_comm,
     struct comm_package *comm_handle);
/* Shared memory allocation */
void Wrapper_MPI_Sharedmemory_alloc(int msize, int bsize,
     int flag, struct comm_package *comm_handle,
     void **shmem_addr, MPI_Win *winPtr);
/* Affinity */
void Wrapper_Get_localpointer(void *start_addr,
     int rank, int dsize, void **local_addr);
/* Free shared memory and bridge communicators */
Wrapper_Comm_free(struct comm_package *comm_handle);
```

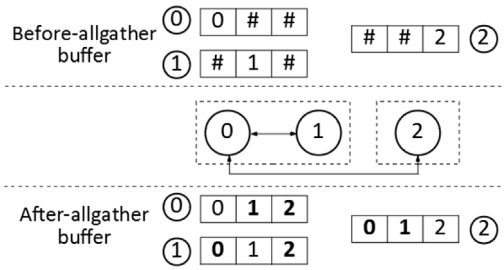
Fig. 3. The wrapper interfaces handling with communicators and regions of shared memory.

I.e. the consecutive ranks fill up each compute (shared memory) node before moving to the next. Each of the standard MPI collective communication interfaces referenced above has a counterpart in our hybrid approach. The counterparts change the parameters slightly. In addition, there could be specific wrapper functions contributing to their implementations. Based on the wrapper primitives, we give a practice in building a prototypical hybrid MPI+MPI code, where the *allgather* is involved. Furthermore, the link[1] provides examples describing the usage of our broadcast and *allreduce* in the hybrid MPI+MPI context. We prove that these wrapper interfaces play an important role in improving the productivity of the hybrid MPI+MPI application developers by unveiling the implementation details hidden in them. According to Fig. 2, on-node synchronization should be carefully considered for the hybrid approach inside the communication component. We thus discuss how the node-level synchronizations could be implemented for benefiting the performance of the hybrid MPI+MPI programs.

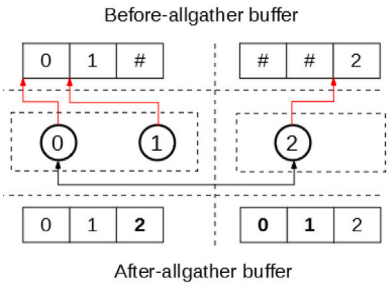
4.1. Common wrapper primitives

In all MPI+MPI programs embracing the collective operations, the steps manipulating communicators and shared regions are common places. To obviate code duplication, we wrap all the common steps into the corresponding wrapper functions, whose interfaces are demonstrated in Fig. 3. The structure *comm_package* defines variables associated with the shared memory and bridge sub-communicators. The function *Wrapper_MPI_ShmemBridgeComm_create* takes a communicator as input parameter and returns an instance of the above structure. Aside from the *MPI_COMM_WORLD*, other communicators deriving from it are supported by this function for complex use cases. The *msize*, *bsize* and *flag* – in function *Wrapper_MPI_ Sharedmemory_alloc* – are parameters defined to determine the total size (in bytes) of a shared region allocated in the *leader*. These two wrapper functions are both one-off activities whose overheads are evaluated in Section 5.2.1. The function *Wrapper_Get_localpointer* needs to be invoked to output a local pointer pointing to the shared memory location with affinity to the calling process. In the end, we need to explicitly deallocate the communicators via the wrapper function *Wrapper_Comm_free*.

[1] https://github.com/HyMPIMPIColl/BenchHyCollWithWrapper.



(a) *Allgather* in the pure MPI version



(b) *Allgather* in the hybrid MPI+MPI version

Fig. 4. Comparison of the pure MPI context-based and hybrid MPI+MPI context-based *allgather* according to the changes in buffers for each process. ♯: empty element; bold font: gathered element from other processes; black arrow: an inter-process communication; red arrow: a local pointer.

4.2. Allgather

This section first describes the implementation dissimilarities of the pure MPI context-based *allgather* (*MPI_Allgather*) and the hybrid MPI+MPI context-based *allgather* in Fig. 4. The latter is called *Wrapper_Hy_Allgather* in our hybrid MPI+MPI version. Our previous paper [20] can be referred to for a more elaborate description. Then we focus on the practices in writing hybrid MPI+MPI code based on our wrapper interfaces and compare it with the one without the use of them.

Both legends in Fig. 4 describe their implementation approaches according to the usage and status of buffer in each process instead of the switch between the *allgather* algorithms (e.g., recursive doubling or ring). Shown in Fig. 4(a), initially process i assigns a valid value to the *i*th element as its local data, that is about to be sent to other processes. After this operation, the message sent from each process is placed in rank order in all processes' after-allgather buffers, where the copies of replicated data inside node are noticed. Besides, the intra-node communication involves extra buffer allocation and copies, which are determined by the underlying MPI library and occur transparently to the user. Unlike the *allgather* in the pure MPI version, only one copy of buffer, which is allocated as a shared memory segment, is demanded on a node in our *allgather*, shown in Fig. 4(b). This buffer is shared among all the on-node processes and thus the intra-node communication is eliminated. Therefore, in our *allgather* the *leaders* (comprise process 0 and 2), as the representatives of the two nodes, are required to exchange all the valid messages. The irregular *allgather* variant (*MPI_Allgatherv*) is leveraged for this across-node data exchanges, due to that the valid message size could vary from one node to another. In order to achieve the same computational parallelism as the standard *allgather*, the on-node shared region is evenly partitioned into separate portions, each of which builds an affinity with a process via a pointer. This is done before the *Wrapper_Hy_Allgather* is executed.

Fig. 5 shows a complete and simple example (micro benchmark) of how to illustrate a hybrid MPI+MPI program containing an *allgather* operation by using our wrapper interfaces. Besides the common

```
1  struct comm_package comm_handle;
2  struct allgather_param param_handle;
3  MPI_Win win;
4  double *result_addr, *s_buf, *r_buf;
5  s_buf = r_buf = NULL;
6  int    nprocs, *sharedmem_sizeset, rank;
7  Wrapper_MPI_ShmemBridgeComm_create(MPI_COMM_WORLD,
8       &comm_handle);
9  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
10 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11 Wrapper_MPI_Sharedmemory_alloc(msg, sizeof(double),
12      nprocs, &comm_handle, (void**)&r_buf, &win);
13 Wrapper_ShmemcommSizeset_gather(&comm_handle,
14      &sharedmem_sizeset);
15 Wrapper_Create_Allgather_param(msg, &comm_handle,
16      sharedmem_sizeset, &param_handle);
17 Wrapper_Get_localpointer(r_buf, rank,
18      msg*sizeof(double), (void**)&s_buf);
19 for(int i = 0; i < msg; i++){ s_buf[i] = i; }
20 Wrapper_Hy_Allgather<double>(r_buf, s_buf, msg,
21      MPI_DOUBLE, &param_handle, &comm_handle);
22 MPI_Win_free(&win);//Free the allocated shared memory
23 Wrapper_Param_free(&comm_handle, &param_handle);
24 Wrapper_ShmemcommSizeset_free(&comm_handle,
25      sharedmem_sizeset);
26 Wrapper_Comm_free(&comm_handle);
```

Fig. 5. A simple hybrid MPI+MPI example including an *allgather* operation.

wrapper functions defined above, there are several wrapper functions and data structures specifically provided for implementing our *allgather* approach. The data structure *allgather_param* (line 2) stores two integer arrays (i.e., `recvcounts` and `displs`) specifying the receive counts and displacements. These two arrays are required by our template function *Wrapper_Hy_Allgather*, which is the counterpart to the *MPI_Allgather* used in pure MPI version and thus is the object to be measured in Section 5.2.2. Lines 13 and 14 generate an array (i.e., *sharedmem_sizeset*) that collects the sizes of all the shared memory communicators. The function *Wrapper_Create_Allgather_param* receives this array as input data and returns a value to *param_handle* of type struct *allgather_param* (lines 15 and 16). This function computes the sets of received counts and displacements for irregular *allgather* and is also a one-off, which could be amortized in the future by repeatedly invoking *Wrapper_Hy_Allgather* operation. In the end, the *sharedmem_sizeset* and *param_handle* should be properly freed.

Fig. 6 is added to expand the wrapper functions of relevance to Fig. 5 and it shows how our design is originally realized in the hybrid MPI+MPI context without our wrapper interfaces. Due to space limit, Fig. 6 skips the declarations of variables. Obviously, the program listed in Fig. 6 (hereafter called *verbose program*) produces more lines of code (LOC) than that demonstrated in Fig. 5 (hereafter called *wrapper program*). In order to better grasp the contribution of these wrapper interfaces, the positional correspondence between the functionalities of the above two programs is further generated and shown in Table 1, where the leftmost column lists the involved functionalities. On the right columns, the line numbers indicate the position of the given functionality in each program. We can observe that each functionality corresponds to one or several wrapper interfaces, which make the *wrapper program* more structured and readable. Conversely, the *verbose program* is prone to obscurity or even failure due to that it explicitly handles the details of all the listed functionalities. In addition, the hybrid MPI+MPI program developers can benefit from this mapping table that enables them to better apply our wrapper interfaces to their own applications. In short, the above study emphasizes the need for the use of the wrapper interfaces with proven benefits – better productivity and applicability – to the hybrid MPI+MPI users.

```
1  /* Hierarchical communicator splitting [28] */
2  comm = MPI_COMM_WORLD;
3  MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED,
4       0, MPI_INFO_NULL, &shmem_comm);
5  MPI_Comm_rank(shmem_comm, &shmemcomm_rank);
6  leader = 0;
7  MPI_Comm_split(comm,
8       (shmemcomm_rank==leader)?0:MPI_UNDEFINED,0,
9       &bridge_comm);
10 Every process gets shmemcomm_size and bridgecomm_size;
11 MPI_Comm_size(comm, &nprocs);
12 msgSize = (shmemcomm_rank==leader)?msg*nprocs:0;
13 MPI_Win_allocate_shared(msgSize, sizeof(double),
14      MPI_INFO_NULL, shmem_comm, &r_buf, &win);
15 if (shmemcomm_rank != leader){
16      MPI_Win_shared_query(win, leader, &r_buf);}
17 MPI_Comm_rank(comm, &rank);
18 if (bridge_comm != MPI_COMM_NULL){
19      sharedmem_sizeset = malloc(.);
20      recvcounts = malloc(.);displs = malloc(.);
21      MPI_Allgather(shmemcomm_size, sharedmem_sizeset,
22         bridge_comm);
23      for (int i = 0; i < bridgecomm_size; i++){
24          recvcounts = msg*sharedmem_sizeset[i];
25          displs[i] = 0;
26          for (int j = 0; j < i; j++)
27              displs[i] = recvcounts[j];}}
28 s_buf = r_buf + msg*rank;
29 for(int i = 0; i < msg; i++){ s_buf[i] = i; }
30 if (bridgeComm != MPI_COMM_NULL){// Leaders
31      MPI_Barrier(sharedmemComm);
32      MPI_Allgatherv(s_buf, r_buf, recvcounts, displs,
           bridgeComm);
33      MPI_Barrier(sharedmemComm);}
34 else{// Children
35      MPI_Barrier(sharedmemComm);
36      MPI_Barrier(sharedmemComm);}
37 MPI_Win_free(&win);
38 MPI_Comm_free(shmem_comm);
39 if (bridge_comm != MPI_COMM_NULL){
40      MPI_Comm_free(bridge_comm);free(sharedmem_sizeset);
41      free(recvcounts);free(displs);}
```

Fig. 6. Pseudo-code that illustrates how to implement the above example (see Fig. 5) without the wrapper interfaces.

```
void Wrapper_Get_transtable(MPI_Comm p_comm,
     const struct comm_package* comm_handle,
     int **shmem_transtable, int **bridge_transtable)
template<class myType>
void Wrapper_Hy_Bcast(myType** bcast_addr,
     myType* start_addr, int msize, int* shmem_transtable,
     int* bridge_transtable, MPI_Datatype data_type,
     int root, struct comm_package* comm_handle);
```

Fig. 7. The specific wrapper interfaces with respect to our broadcast.

Table 1
Correspondence between *wrapper program* and *verbose program*.

Functionality	Lines	
	wrapper program	*verbose program*
Communicator splitting	7–8	2-10
Shared memory allocation	11–12	12-16
Fill `recvcounts` and `displs`	13–16	18-27
Get local pointer	17–18	28
Allgather	20–21	30-36
Deallocation	23–26	38-41

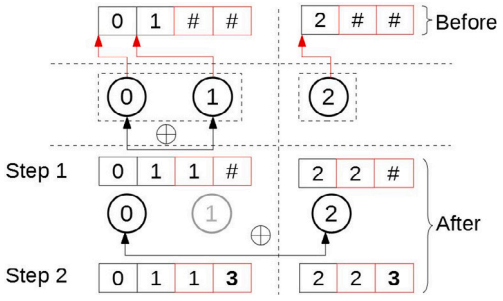


Fig. 8. Illustration of the hybrid MPI+MPI context-based *allreduce*. The input data is enclosed with black cubic and the reduced results – either locally or globally – are enclosed with red cubic, where the globally reduced ones are stressed with bold font. ⊕ means *MPI_SUM* operation, which is applied to the input and locally reduced data. Refer to Fig. 4 for the explanations of arrows.

4.3. Broadcast

A broadcast operation happens when one MPI process, called *root*, sends the same message to every other process. Likewise, in our MPI+MPI context-based broadcast approach, a region of memory is allocated to store the broadcast data in each *leader* and can be shared by its *children*. Only the *root* is eligible to alter the broadcast data according to the MPI broadcast semantics. All processes on the same node independently read the broadcast data via a local pointer to the beginning of this shared memory location. Here, performing the across-node broadcast operation (over all the *leaders*) is straightforward since the size of the broadcast message remains the same as that of the pure MPI context-based broadcast.

Broadcast operation is rooted and can only be performed when the *root*'s rank is correctly given. Every process can be the *root* in real world. This confronts us with the challenge of determining the relative rank of the *root* in both the shared memory and bridge sub-communicators. Two absolute-to-relative rank translation tables – *shmem_transtable* and *bridge_transtable* – are thus generated in function *Wrapper_Get_transtable*. It brings implementation overhead to our broadcast approach. The function *Wrapper_Hy_Bcast* receives the above two translation tables as input data to perform our hybrid broadcast operation. The above two primitives provided for our broadcast are shown in Fig. 7.

4.4. Allreduce

The implementation method of the hybrid MPI+MPI context-based *allreduce* is illustrated in Fig. 8. Each process points to an element (as an input), which is located in the shared region and supposed to be updated by its affiliated process. Besides, an output vector with 2 elements is appended to store the locally and globally reduced results, respectively. Rather, the reduction computation in step 1 proceeds at the node level. Step 2 is performed by all *leaders* and applies the sum operation to the first elements with the final result stored in the second element of the output vector. This output vector is shared and can be accessed by all processes on the same node, but only under a proper synchronization to secure the computational determinacy. The reduced result is thus not necessarily broadcast to all other on-node processes. Clearly, the order of operands in our *allreduce* approach is not defined to be in ascending order of process rank beginning with zero. In this example with block-style placement, we take advantage of the associativity of the sum operation to guarantee the correction of the reduced result. However, the operation should be both commutative and associative when a non-block-style placement is applied. This *allreduce* approach is thus valid for all predefined operations, which are assumed to be commutative as well as associative.

The template function *Wrapper_Hy_Allreduce* acts as the counterpart to the *allreduce* in the pure MPI version. Each node is required to contribute an intermediate result to the output vector in step 1, which can be completed in two ways. One is letting the *leader* serially perform the sum operation on an element-wise basis, which however leads to the issue of core idles and extra synchronization (explained in Section 4.5). The other is performing an *MPI_Reduce* to return the locally-reduced result to the *leader*, which implies a synchronization point among on-node processes, and however brings MPI internal memory copies. Step 2 proceeds with a standard *allreduce* operation called by all *leaders*. Fig. 9 shows the template interface to *Wrapper_Hy_Allreduce*, where the input parameters *sharedmem_rank* and *win* are responsible for the identification of local pointer in step 1 and the synchronization operations after step 2, respectively.

```
template<class myType>
void Wrapper_Hy_Allreduce(myType* start_addr,
     myType** result_addr, int sharedmem_rank,
     int msize, MPI_Datatype data_type, MPI_Op op,
     struct comm_package* comm_handle, MPI_Win win);
```

Fig. 9. The template interface to *Wrapper_Hy_Allreduce*.

```
if (comm_handle->bridge_comm != MPI_COMM_NULL){// Leaders
   sync(comm_handle->shmem_comm);
   MPI_Allgatherv(sbuf,rbuf,...,comm_handle->bridge_comm);
   sync(comm_handle->shmem_comm);}
else{// Children
   sync(comm_handle->shmem_comm);
   sync(comm_handle->shmem_comm);}
```

(a) *allgather*

```
if (comm_handle->bridge_comm != MPI_COMM_NULL){// Leaders
   MPI_Bcast(buf, ..., comm_handle->bridge_comm);
   sync(comm_handle->shmem_comm);}
else// Children
   sync(comm_handle->shmem_comm);
```

(b) broadcast

```
/* Step 1 */
Method 1:
MPI_Reduce(..., comm_handl->shmem_comm);
Method 2:
sync(comm_handle->shmem_comm);
Each leader applies the operation on node-level;
/* Step 2 */
if (comm_handle->bridge_comm != MPI_COMM_NULL)
   MPI_Allreduce(sbuf,rbuf,..., comm_handle->bridge_comm);
sync(comm_handle->shmem_comm);
```

(c) *allreduce*

Fig. 10. Three pieces of pseudo-code handling with the synchronization among on-node processes for our *allgather*, broadcast and *allreduce*.

```
Leader:
   status = 0;
   compute&communication;
   status++;
   MPI_Win_sync(win);
```

```
Children:
   ref = 0; ref++;
   while(1){
     MPI_Win_sync(win);
     if(status==ref) break;}
```

Fig. 11. Pseudo-code that demonstrates the spinning method.

4.5. Synchronization consideration

The synchronization and communication among processes are more decoupled in hybrid MPI+MPI, than those in pure MPI. Therefore, the synchronization operations need to be explicitly added to guarantee

the data integrity and support a deterministic computation in hybrid MPI+MPI. This section supplements the illustration of our collectives with due consideration of the node-level synchronization points, which are intuitively marked with *sync* in Fig. 10. The calls to *sync* are highlighted using two kinds of colors featuring different synchronous patterns.

Next, we shed lights on the two different synchronous patterns reflected in the above three MPI+MPI context-based collective functions (prefixed with `Wrapper_Hy`), which are assumed to execute on more than one node. We start with the implementation of the function *Wrapper_Hy_Allgather*, where two *sync* calls among all the on-node processes need to be added before and after the irregular *allgather* operation, respectively. The first *sync*, shown in red, guarantees that all processes finish the updates to the shared data that has affinity to them. The second *sync*, shown in yellow, is invoked to block the *children* until the *leaders* exit from the irregular *allgather* operation. The first plot in Fig. 10 shows its implementation. Then it comes to the function *Wrapper_Hy_Bcast*, in which a *sync* operation is needed after the broadcast operation to guarantee that the broadcast data is ready for all the on-node processes. The second plot in Fig. 10 shows the related pseudo-code. This is followed by the function *Wrapper_Hy_Allreduce*, characterizing the two methods of the intermediate reduction among on-node processes. The *method* 1 is adopted to return the reduced result to each *leader* for simplicity and flexibility, which could bring performance issues due to the MPI internal buffering policy. Instead of calling *MPI_Reduce*, we can use an ad hoc method (*method* 2). It adds a *sync* operation to guarantee that all the input data is ready to be used by the *leader* to compute the reduced result. In addition, the second *sync* comes to let the *children* wait for the completion of the *allreduce* operation called by *leaders*. Its implementation is illustrated by the last plot in Fig. 10.

Based on Fig. 10, we can draw a general conclusion that the *sync* in red entails a collective synchronization among a set of processes and the *sync* in yellow can be treated as a lightweight one in comparison to the former. Rather, with the *sync* in red, each process must stop at this point until all other processes reach this *sync*. The function *MPI_Barrier* is thus applicable to this *sync*. And yet all the *children* must pause until their *leader* reaches the *sync* in yellow. In short, the *sync* in yellow synchronizes the *leader* with its *children*. If this *sync* is also substituted by a barrier, the *children* will end up waiting for each other, which implies unnecessary handshaking and leads to severe degradation of performance. This *sync* occurs after a barrier point in terms of our *allgather* and *allreduce* approaches, where the expected wait time for the *leaders* should not be long. Because in this situation the process skew is caused by the fact that only the *leaders* participate in the collective operation. Therefore, spinning in a loop [35] could be a simple as well as a more efficient alternative synchronized method to the barrier.

To implement the spinning method, a shared variable (named *status*) is defined, which can only be updated by the *leader*. The *children* check the shared variable by spinning in a polling loop. In other words, The *children* do not exit the loop until the update to the shared variable meets an exit condition. This spinning method is worthwhile only if the update to the shared variable takes low clock cycles, otherwise performance issues will be caused, since many cores waste time doing useless computation. We thus simply use the increment (++) operator to modify the shared variable. MPI establishes a restriction [1] on the concurrent access to the same shared memory location as it does not support atomic operators (such as increment) on numeric values requiring more than one byte. This restriction permits polling on a shared memory location for a change from one value to another value rather than comparing them. In our implementation, the shared variable is contained within an MPI shared memory window. Hence, the above restriction must be considered to guarantee a definite outcome and prevent the *children* from being stuck in an endless loop. The exit condition is then expressed as 'the shared variable == a certain value', rather than as 'the shared variable $\geq/\leq$ a certain value'. Note that the routine *MPI_Win_sync* must be included by both the *leader* and its *children* to achieve a processor-memory barrier (see Section 3.2.2). The spinning method is evaluated together with our *allreduce* approach and implemented as shown in Fig. 11.

5. Evaluation

In this section, we compare the performance characteristics of the hybrid MPI+MPI programs (including our collectives) with the pure MPI and hybrid MPI+OpenMP programs (containing the standard MPI counterparts). Our studies were conducted on two parallel clusters by measuring the latencies with a varying number of cores and different message sizes. We first briefly describe our experimental testbed, then discuss the overheads of the aforementioned one-off activities, and finally evaluate the performance of the micro-benchmarks and application kernels. These micro-benchmarks were developed according to the OSU benchmark[2] and averaged over 10,000 executions. The kernel-level experiments consist of a computation with Scalable Universal Matrix Multiplication Algorithm (SUMMA), 2D Poisson solver and Bayesian Probabilistic Matrix Factorization (BPMF). We used the default MPI rank placement scheme – block-style – to run all these benchmarks.

5.1. Experimental setup

We used a Cray XC40 and a NEC cluster for our experiments:

1. Cray XC40 (aka. Hazel Hen): Each of the Hazel Hen compute nodes has 24 Intel Haswell cores running at 2.5 GHz with 128 GB of DDR4 main memory. The cores are organized as two sockets with 12 cores per socket (each socket is seen as a NUMA domain). The nodes are connected with dedicated Cray Aries network which has a dragonfly topology. The GNU programming environment 6.0.5 and the version of cray-mpich/7.7.6 were applied to this system.
2. NEC cluster (aka. Vulcan): Vulcan consists of several compute nodes of different types. We used SandyBridge (SB) and Haswell compute nodes. Each of the SB compute node has in total 16 SB cores running at 2.6 GHz with 64 GB DDR3 main memory (8 cores per NUMA domain). The configuration of the Haswell compute node is the same as above. The applied GNU compiler version was 8.3.0. The nodes are connected via the InfiniBand network. The version of Open MPI/4.0.1 was run.

5.2. Microbenchmark evaluation

All the microbenchmark evaluations were executed on both of the two clusters and described in two aspects: the overhead caused by our design (called implementation overhead below) and the performance comparison between the standard MPI collectives and their counterparts (our approaches) in the hybrid MPI+MPI context. For brevity we mostly present the evaluation results on Vulcan with Haswell compute nodes. But we will go into details when the results on Hazel Hen are different from those on Vulcan. The labels prefixed with `Wrapper_Hy` in the following figures indicate our collectives, otherwise they refer to the standard ones.

[2] http://mvapich.cse.ohio-state.edu/benchmarks/.

Table 2
One-off overheads associated with the hybrid MPI+MPI programs containing collective operations.

Primitives		#Cores			
		16	64	256	1024
		Mean (us)	Mean (us)	Mean (us)	Mean (us)
Common	`Communicator`	64.8	170.9	413.7	1098.7
	`Allocate`	188.3	262.5	307.1	311.8
`Bcast_transtable`		0.7	9.2	95.9	1462.8
`Allgather_param`		0.3	2.9	7.1	19.9

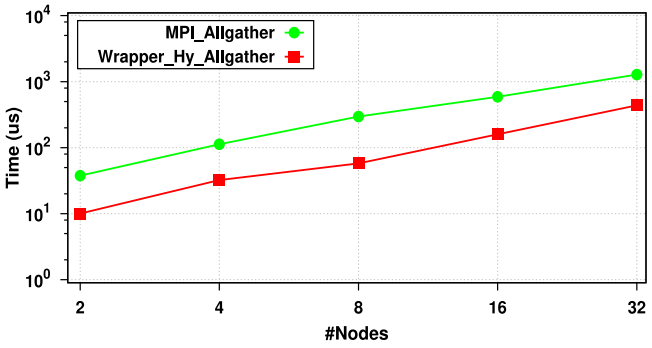


Fig. 12. The performance comparison between *Wrapper_Hy_Allgather* and *MPI_Allgather* on Hazel Hen with varying number of nodes. The size of the gathered message from every process is 800 B.

5.2.1. Implementation overhead

Table 2 displays the implementation overhead imposed by our design on Vulcan with the leftmost columns listing the primitives. Besides the common primitives of two-level communicator splitting (`Communicator`) and shared memory allocation (`Allocate`), the primitives of *Wrapper_Get_transtable* and *Wrapper_Create_Allgather_param* are abbreviated as the `Bcast_ transtable` and `Allgather_ param`, respectively. Their overheads are subject to the number of cores rather than message sizes. Hence, their overheads over the number of cores (16, 64, 256 and 1,024) are given to investigate the scalability of these wrapper functions.

The rows of `Communicator` and `Bcast_transtable` show that their overheads increase nearly proportionally to the number of cores. The `Allocate` shows good scalability but its overhead should still be analyzed in a hybrid MPI+MPI program. The overhead in regard to our *allgather* approach – shown in the last row – is almost negligible. The evaluation on Hazel Hen shew similar implementation overheads as were shown on Vulcan, except the overheads for `Communicator` and `Bcast_transtable` were one magnitude fewer. All of these overheads can be treated as one-offs, which means they will not repeatedly be added up to the total elapsed time of a hybrid MPI+MPI program. Nevertheless, we need to check the effectiveness of applying the hybrid MPI+MPI mechanism to an application by analyzing the occurrence frequency and accumulated overhead of the collective operation. We should thus guarantee that the implementation overheads are traded for the greater performance benefits of our collectives' counterparts (i.e., *Wrapper_Hy_Allgather*, *Wrapper_Hy_Bcast* and *Wrapper_Hy_Allreduce*).

5.2.2. Allgather comparison

Fig. 12 shows the time performance comparison between *MPI_Allgather* and *Wrapper_Hy_Allgather* on 2, 4, 8, 16 and 32 nodes for a fixed message length of 800 B. Here each of the nodes was populated with 24 processes. The same number of processes in different nodes leads to a regular *allgather* problem. We can observe the advantage of our *allgather* due to its constant lower latencies. However, the study of the performance characteristics of our proposed *Wrapper_Hy_Allgather* does not merely discuss the regular problems. The *MPI_Allgatherv* suffers a performance penalty, since its performance is determined by the maximum amount of data to be received by a node [36]. The irregular problem, where the number of MPI processes varies from node to node, is however a commonplace for our *allgather* approach. Hazel Hen is equipped with non-power-of-two cores (24) nodes throughout the system, leading to irregularly-populated nodes when we request power-of-two processes. Our previous work [20] confirms a comprehensive insight into the performance benefits of our *allgather* approach for irregular as well as regular problem on Vulcan and Hazel Hen clusters.

5.2.3. Broadcast comparison

The Open MPI/4.0.1 supports several implementation algorithms for each of the collective communication operations. The decision to switch between them depends on the size of communicator as well as the message size. More precisely, two message size thresholds, 2 KB and ~ 362 KB, are used in the Open MPI broadcast implementation. We decided upon defining small, medium, and large message as $\leq$ 2 KB, $>$ 2 KB and $\leq$ 362 KB, and $>$ 362 KB for the purpose of this experiment.

Fig. 13 compares the time performance of *MPI_Bcast* and *Wrapper_Hy_Bcast*, with 16, 64, 256 and 1,024 cores on Vulcan. We varied the numbers of the broadcast elements of double precision floating pointer (8 B) from 2^0 to 2^{17} in this benchmark. For brevity, we reported only the latency results for element counts of 2^2, 2^9, 2^{14} and 2^{16}, which represent small (32 B), medium (4 KB and 128 KB) and large messages (512 KB), respectively. The current version of *Wrapper_Hy_Bcast* replaces the synchronization point with a barrier operation. We observe that our proposed broadcast approach offers significantly lower latency than the standard one, except for the small message running on 64 cores. This is probably because the synchronization overhead contributes more to the latency of broadcast than the data transfer overhead. The impact of the synchronization point on the performance of our collectives will be discussed at length in Section 5.2.4. On Hazel Hen, the standard broadcast was always inferior to our approach. The first subplot shows the results running on 16 cores, where all the MPI processes reside on the same node and thus no inter-node data exchanges will be involved. In this scenario, only an *MPI_Barrier* is called by the on-node processes and as we expected, its latency almost keeps constant, regardless of the message lengths. The remaining three subplots show the latency results across different nodes, wherein the curves go up steadily as the broadcast message size grows except when the message size reaches 512 KB. Such exception happens, since the broadcast algorithm is changed from *split binary tree* [37] to *pipeline*.

5.2.4. Allreduce comparison

The intermediate message threshold (~ 9 KB) defined in the Open MPI *allreduce* implementation roughly determines whether the involved message in this experiment is small, medium or large.

Fig. 14 compares the time performance of *MPI_Allreduce* and *Wrapper_Hy_Allreduce* on Vulcan, as either the message length or the number of cores grows. We ran this experiment with the increasing number of elements of double precision floating point, where 2^2, 2^9, 2^{15} and 2^{17} were chosen as the representatives for the small (32 B), medium (4 KB) and large message (256 KB and 1 MB). The version of *Wrapper_Hy_Allreduce* for Fig. 14 used *method* 1 to implement step 1 and replaced the synchronization point in step 2 with a barrier call. Not surprisingly, our *allreduce* approach fails to significantly outperform the standard one for small messages on 16 cores. Otherwise, speedups (range from 27.2% to 82.5%) of our *allreduce* over the standard approach can be achieved anywhere. On Hazel Hen, our *allreduce* performed worse than the standard approach for small-size messages up to 2 KB on all the above number of cores. The inferiority of our *allreduce* for small messages is attributed to inadequate methods for step 1 or inefficient synchronization implementation. The data transfer latency for small messages is very low and thus strongly affected by the overhead of

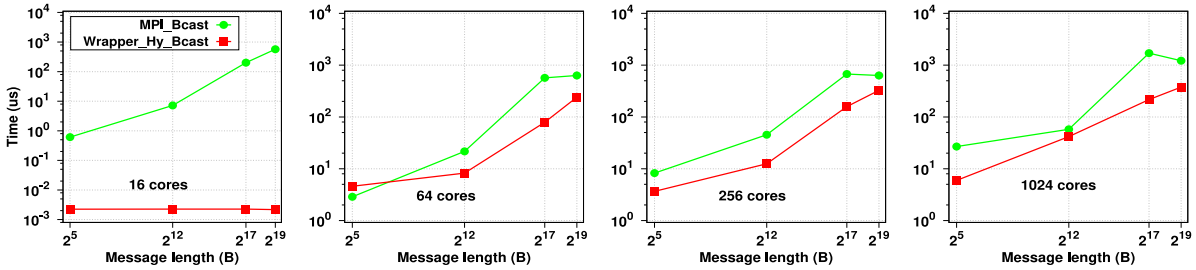


Fig. 13. The time performance comparison between *Wrapper_Hy_Bcast* and *MPI_Bcast* on Vulcan with varying numbers of cores and message lengths.

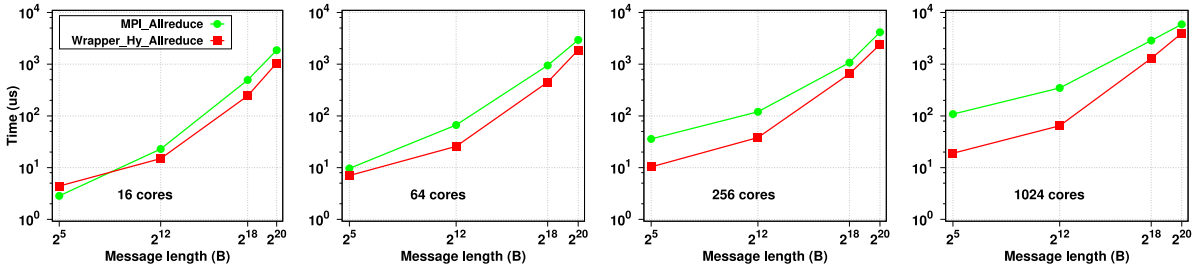
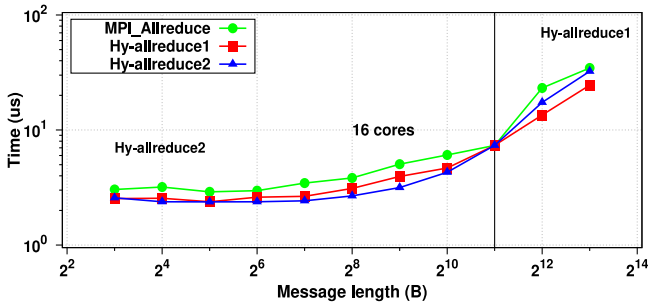


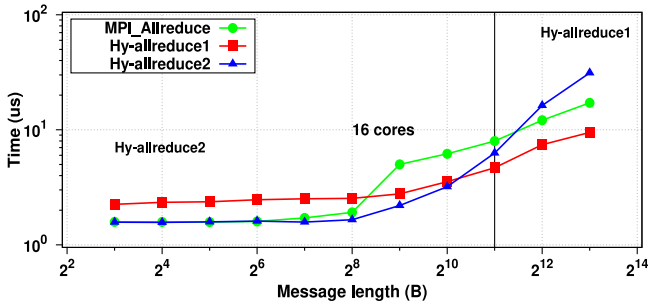
Fig. 14. The time performance comparison between *Wrapper_Hy_Allreduce* and *MPI_Allreduce* on Vulcan with varying numbers of cores and message lengths.

the synchronization operation. However, in an application the involved *allreduce* operation could be used with messages centering on the sizes smaller than 1 KB [38]. This drives us to evaluate the performance of the *method* 2 (for step 1) and the spinning method (for step 2). Using *method* 2 instead will slightly improve the performance of our approach on both Vulcan and Hazel Hen only for a range of small message sizes. Then again, adopting the spinning method can noticeably lessen the latency of our *allreduce*, especially for small messages, on both Vulcan and Hazel Hen. For large messages, we observed that our *allreduce* latencies of both versions using barrier and spinning are at the same level. This is because, for large messages the synchronization time becomes insignificant and then the data transfer overhead dominates the latency of our *allreduce*. Therefore, the current version of *Wrapper_Hy_Allreduce* replaces the synchronization point in step 2 with the spinning method and chooses between *method* 1 and *method* 2 in terms of message sizes for optimal performance.

Next, we develop two versions of our *allreduce*, one with *method* 1 and the other with *method* 2, to determine the cut-off value of the message size for switching from *method* 2 to *method* 1 in the optimal version of our *allreduce*. Henceforth the versions with *method* 1 and *method* 2 are abbreviated as *Hy-allreduce1* and *Hy-allreduce2*, respectively. Fig. 15 compares the latency of *Hy-allreduce1* and *Hy-allreduce2* on 16 cores, which all reside on the same node. This experiment ran with the message sizes ranging from 8 B to 8 KB on Vulcan and Hazel Hen. Besides, the performance curve for *MPI_Allreduce* is added as baseline. Obviously, the cut-off value of the message size is 2 KB, which is marked with a vertical line. The *Hy-allreduce2* performs slightly better before the cut-off point and becomes worse when it is surpassed. Therefore, our *allreduce* is further optimized to use *method* 2 and *method* 1 before and after the cut-off point, respectively. Fig. 14 already shows us the scalability of the initial version of our *allreduce* with the increasing number of cores on Vulcan. It was therefore necessary to reevaluate the scalability after the above tuning. The curve trends presented in the new plots (omitted for brevity), that we obtained during reevaluation, coincided well with those displayed in Fig. 14, except for the first subplot with 16 cores. Therefore, the results reflected in Fig. 14 are also partially fit for further reference in Section 5.3.2. We then compute the performance gap between our optimized *allreduce* and the standard approach on Hazel Hen for 64, 256 and 1,024 cores respectively. The results are shown in Fig. 16, where the label *MSG* denotes the message length in bytes. From this figure, we can observe that the standard



(a) Vulcan



(b) Hazel Hen

Fig. 15. The time performance comparison of *Hy-allreduce1*, *Hy-allreduce2* and *MPI_Allreduce* for small messages on a single node (16 cores) on Vulcan and Hazel Hen.

allreduce still slightly outperforms our *allreduce* for 8 B and 32 B. The negative performance gap at 128 B means that our *allreduce* starts to perform better than the standard approach.

5.3. Kernel-level benchmarks

In this section, we consider three benchmarks – SUMMA, 2D Poisson solver and BPMF – that have different collective communication operations interweaving with real computations. The BPMF was executed on Hazel Hen with Haswell compute nodes (each contains 24 cores)

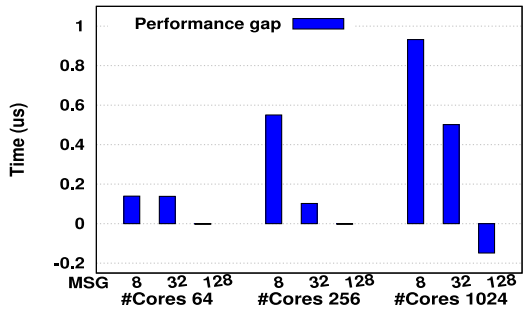


Fig. 16. The performance gap between *MPI_Allreduce* and the optimized *Wrapper_Hy_Allreduce* on Hazel Hen.

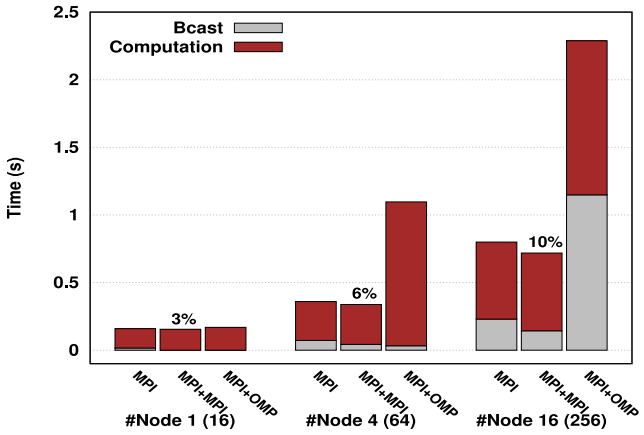


Fig. 17. The time performance comparison between different implementations of SUMMA on Vulcan.

while the SUMMA and 2D Poisson solver were run on Vulcan with SB compute nodes, of which each contained power-of-two (16) cores. Each node was fully populated with MPI processes or OMP threads when the three benchmarks were executed below. The experimental results for SUMMA and 2D Poisson solver were the average of at least 20 runs and those for BPMF were the average of 3 runs. All of them shew standard deviations of only a few percentages. For each benchmark we assessed the time performance and LOC of the pure MPI, hybrid MPI+OpenMP and hybrid MPI+MPI implementations, where the former two utilized the standard MPI primitives to implement the relevant collective operations and the last one utilized our wrapper primitives. Specifically, we simply used the loop-level parallelization in the hybrid MPI+OpenMP implementations without putting great efforts into achieving optimal performance. It is to be noted that we only launched one MPI process per node and then this MPI process spawned threads fully populating the available core resources on each node when running them, regardless of whether they were run on Vulcan or Hazel Hen. A thread was pinned to a specific core and this pinning went successively through available cores. To achieve this, the environment variables `OMP_PLACES` and `OMP_PROC_BIND`, and the option of `--map-by` needed to be correctly set on Vulcan and the `aprun` option of `-d` was given to specify the number of threads per MPI process on Hazel Hen.

The *total* time shown in Figs. 17 to 19 is described as the sum of computation overhead and relevant collective communication latency. This can facilitate us to intuitively comprehend the impacts of the latter on the *total* performance and scalability of our benchmarks. The *total* here denotes the core part including intensive computation and collective communication operations in each benchmark.

5.3.1. SUMMA

SUMMA multiplies two dense matrices by using a scalable universal algorithm [39]. In this kernel, two square matrices of the same type (double-precision) and size are required as input data and evenly decomposed into blocks, each of which is assigned to an MPI process. This kernel is a typical example of supporting multiple communicators in our design. Herein we first logically laid out the *MPI_COMM_WORLD* into a two-dimensional Cartesian grid and then created sub-communicators for rows and columns. This kernel consists of multiple core phases, whose elapsed time is our measurement target. In each core phase, two broadcast operations on the row and column sub-communicators are triggered due to the dependencies on the blocks living on the other MPI processes.

We ran all the three implementations (pure MPI, MPI+MPI, and MPI+OpenMP) on Vulcan using three matrices of size 1024×1024, 2048×2048 and 4096×4096, each with 1, 4 and 16 nodes, respectively. The corresponding number of cores are indicated in parenthesis, shown in Fig. 17. The same is true of Figs. 18 and 19. Fig. 17 demonstrates the elapsed time of the core phases of the three SUMMA implementations, where the broadcast message size is 512 KB. The comparison results on Hazel Hen can be found in [20]. We observe that the hybrid MPI+OpenMP implementation indeed brings the minimal broadcast overhead, but its computation overhead is greater than the other two implementations. However, the broadcast message size in the hybrid MPI+OpenMP implementation is always larger than those in the other two implementations due to the fewer number of MPI processes. This disparity can lead to an exception – the hybrid MPI+OpenMP implementation delivers larger broadcast overhead than the other two – on 16 nodes. More significantly, the hybrid MPI+MPI implementation consistently has the best performance of the three SUMMA implementations. Further, the improvements (i.e., 3%, 6%, and 10%) of the hybrid MPI+MPI implementation over the pure MPI one are explicitly given in this figure. This is not unexpected, since the hybrid MPI+MPI implementation constantly delivers less broadcast overhead in terms of the lower height of *Bcast* bar. After revisiting Fig. 13, it can be found that our broadcast (*Wrapper_Hy_Bcast*) outperforms at 512 KB, from which we can infer that the superiority of this hybrid MPI+MPI implementation over the pure MPI one is due to the usage of our broadcast method. Compared with LOC for the pure MPI implementation, the hybrid MPI+MPI implementation brings 6 additional LOC for an increase of 2% in program size.

5.3.2. 2D Poisson solver

This kernel solves the 2D Poisson equation in an iterative way. A square grid holding elements of floating point is initialized and evenly decomposed by rows among the MPI processes. In an iteration each MPI process first uses the Gauss–Seidel method to do a five-point stencil computation on the current grid, and then locally computes the maximum difference between the updated and exact grid, and finally collectively calls the *allreduce* operation to obtain the global maximum difference among all MPI processes. This iteration is repeated until the global maximum difference is less than a predefined convergence value. In this experiment, the Gauss–Seidel module contains data transfers between adjacent processes as well as the five-point stencil computation. The data transfers are performed using a pair of MPI point-to-point routines (i.e, *MPI_Send* and *MPI_Recv*). The computation proceeds in the form of two nested loops. We started our timing at the beginning of iterations and stopped it until the convergence is reached. We used three input grids of size 256×256, 512×512 and 1024×1024, each running on the number of nodes – 1, 4 and 16, respectively. In Fig. 18, we discuss the performance of the 2D Poisson solver kernel. The involved *allreduce* operation is always used with small message of 8 B (aka. global maximum difference), regardless of the grid size or node counts. The curves in Figs. 14 and 15 reveal that the performance benefits of our *allreduce* over the standard approach are marginal on the small system (i.e., smaller than 64 cores) and increase as the system

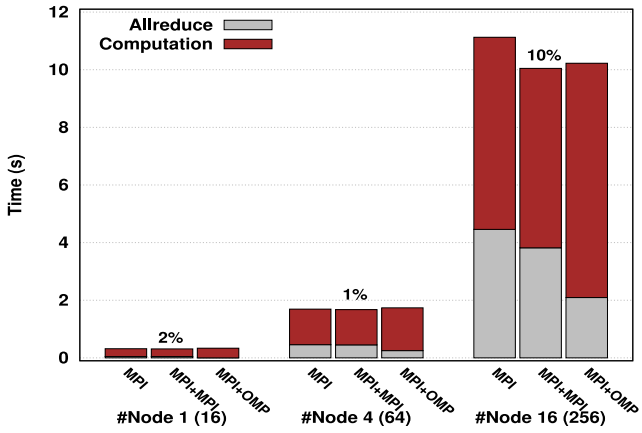


Fig. 18. The time performance comparison between different implementations of 2D Poisson solver on Vulcan.

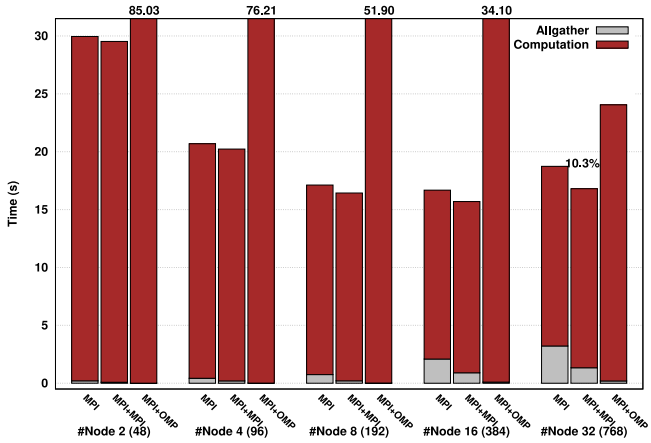


Fig. 19. The time performance comparison between different implementations of BPMF on Hazel Hen.

size grows, for small messages (i.e., smaller than 32 B). This, in turn, explains that on 16 nodes the hybrid MPI+MPI implementation yields a 10% time performance improvement over the pure MPI one, while it brings smaller performance gains of 2% and 1% on 1 and 4 nodes, respectively. We can also learn that these performance advantages offered by the hybrid MPI+MPI implementation are credited to the application of our *allreduce*. The hybrid MPI+MPI implementation adds 7 more additional LOC for a code size increase of 1.6%, by comparison to the pure MPI one.

5.3.3. BPMF

The BPMF kernel [40,41] predicts compound-on-target activity in chemogenomics based on machine learning. The number of iterations to be sampled was set to be 20 for this experiment. Each iteration consists of two distinct sampling regions on compounds and on-target activities followed by a prediction. Both regions end with three calls to the regular *allgather* operation. In the three *allgather* operations, the sizes of the gathered messages from every process are 80000 B, 800 B and 8 B, respectively. The link[3] provides more information about the BPMF code. The strong scaling performance of the BPMF kernel with three different implementations is demonstrated in Fig. 19. Here, the elapsed time of the sampled 20 iterations is evaluated. We used the *chembl_20* as our input training dataset, which is a sparse matrix converted from ChEMBL publicly available data. This kernel was run on different numbers of nodes, as shown in Fig. 19. It is obviously observed that the hybrid MPI+MPI implementation is constantly superior to the other two implementations. The hybrid MPI+OpenMP implementation fails to be comparable to the other two, although the performance gap between them shrinks as the system size increases. The performance of both the pure MPI and hybrid MPI+MPI implementation degrades when the node count increases from 16 to 32, since the increased *allgather* overhead overrides the decreased computation time. Still, it is true that the performance of the pure MPI implementation deteriorates more and the improvement of the hybrid MPI+MPI implementation over the pure MPI one increases to 10.3% on 32 nodes. Fig. 12 implies that the application of our *allgather* can take credit for the performance advantage of the hybrid MPI+MPI implementation over the pure MPI one. The hybrid MPI+MPI implementation brings 12 extra LOC with an increase of less than 0.1% in code size, compared with the pure MPI one.

[3] https://github.com/ExaScience/bpmf/.

6. Discussion and conclusion

This paper proposes an innovative design method of the collective communication operations (such as broadcast, *Allgather* and *Allreduce*) that adapts to the hybrid MPI+MPI context, and then describes them by assuming the block-style MPI rank placement. With the other MPI rank placement schemes, our previous work [20] discusses the measures that can be taken to ensure the validity of our method. Unlike the standard MPI collectives, our collectives only maintain one copy of replicated data shared by all on-node processes. In this way, the explicit on-node inter-process data transfers are completely eliminated. However, synchronization calls need to be adequately added to guarantee the determinacy of the shared data among the on-node processes. The micro-benchmark evaluations present the overheads imposed by our implementation and reveal that our collectives are on par with or outperform those in the pure MPI context on Hazel Hen and Vulcan. The synchronization overhead and its influence on the time performance of *allreduce* are also analyzed. The application kernel evaluations show the superiority of the hybrid MPI+MPI implementations to the other two implementations – pure MPI and hybrid MPI+OpenMP – in time performance, which is credited to our collectives. Further, the productivity gained from the hybrid MPI+MPI model can be comparable to that gained from the pure MPI model in terms of LOC, owing to the wrapper primitives that encapsulate all the implementation details of our design from programmers.

For the evaluation results illustrated in Section 5.3, further explanations are given. First, the speedups of the hybrid MPI+MPI implementations are clearly quantified, from which we observe that they are insignificant on a smaller number of nodes. This does not necessarily mean that the performance advantage of our approach increases as the system grows but instead the proportion of time spent in the collectives is greater. Second, note that the obtained performance gains are kept above the overall implementation overhead (see Table 2) in all the three hybrid MPI+MPI implementations. Otherwise, the performance benefits caused by our collectives are pointless.

Our design method lacks in the distinction between the intra- and inter-NUMA data accesses, since all the *children* – no matter they and their *leader* are located in the same NUMA domain or not – are limited to access the shared data allocated in the *leader*'s memory space. To enable NUMA awareness, the most intuitive solution is to elect a *leader* in each NUMA domain at the price of maintaining a copy of replicated data in it. This comes with extra memory copes and thus needs to be further investigated. Note that our collectives may not apply to the applications using the master/slave pattern, where the master needs to broadcast/gather a large amount of data to/from its slaves. This is due

to that the allocation of the MPI shared memory window fails when the total amount of shared memory required by on-node processes exceeds a limit of size determined by MPI implementation. E.g., with Open MPI, the available shared memory space on a Haswell compute node (see Section 5.1) is in the order of 63 GB. Therefore, the use of our collectives is limited, but to a lesser extent.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank Eric Gedenk for proofreading the article and Tom Vander Aa for offering the pure MPI implementation of the BPMF benchmark. The comments made by the editor and reviewers are deeply appreciated. Part of this work was supported by the European Union's Horizon 2020 POP project [grant numbers 676553, 824080].

References

[1] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.1, High Performance Computing Center Stuttgart (HLRS), 2015, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, (Accessed 13 July 2020).

[2] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, K. Warren, Introduction to UPC and Language Specification, Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[3] S. Poole, O. Hernandez, J. Kuehn, G. Shipman, A. Curtis, K. Feind, Openshmem - toward a unified RMA model, in: D. Padua (Ed.), Encyclopedia of Parallel Computing, Springer US, 2011, pp. 1379–1391.

[4] L. Dagum, R. Menon, Open MP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1998) 46–55, http://dx.doi.org/10.1109/99.660313.

[5] D.E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R.E. Grant, T. Naughton, H.P. Pritchard, M. Schulz, G.R. Vallee, A survey of MPI usage in the US exascale computing project, Concurr. Comput.: Pract. Exper. (2017).

[6] F. Cappello, D. Etiemble, MPI versus MPI+ OpenMP on IBM SP for the NAS benchmarks, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 2000, p. 12.

[7] INTERTWinE, Best Practice Guide to Hybrid MPI + OpenMP Programming, Tech. Rep., 2017.

[8] G. Krawezik, F. Cappello, Performance comparison of MPI and OpenMP on shared memory multiprocessors, Concurr. Comput.: Pract. Exper. 18 (1) (2006) 29–61.

[9] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: D.E. Baz, F. Spies, T. Gross (Eds.), PDP, IEEE Computer Society, 2009, pp. 427–436.

[10] N. Drosinos, N. Koziris, Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters, in: IPDPS, IEEE Computer Society, 2004.

[11] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, R. Thakur, Hybrid parallel programming with MPI and unified parallel C, in: Proceedings of the 7th ACM International Conference on Computing Frontiers, ACM, 2010, pp. 177–186.

[12] J. Jose, K. Kandalla, M. Luo, D.K. Panda, Supporting hybrid MPI and OpenSHMEM over InfiniBand: Design and performance evaluation, in: 2012 41st International Conference on Parallel Processing, IEEE, 2012, pp. 219–228.

[13] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur, MPI+ MPI: A new hybrid approach to parallel programming with MPI plus shared memory, Computing 95 (12) (2013) 1121–1136.

[14] M. Brinskiy, M. Lubin, An Introduction to MPI-3 Shared Memory Programming, Tech. Rep., Intel Corporation, 2017.

[15] H. Zhou, K. Idrees, J. Gracia, Leveraging MPI-3 shared-memory extensions for efficient PGAS runtime systems, in: J.L. Träff, S. Hunold, F. Versaci (Eds.), Euro-Par, in: Lecture Notes in Computer Science, vol. 9233, Springer, 2015, pp. 373–384.

[16] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B.W. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur, Leveraging mpi's one-sided communication interface for shared-memory programming, in: J.L. Träff, S. Benkner, J.J. Dongarra (Eds.), EuroMPI, in: Lecture Notes in Computer Science, vol. 7490, Springer, 2012, pp. 132–141.

[17] D. Karlbom, A performance evaluation of MPI shared memory programming, 2016.

[18] NASA, NASA Parallel benchmarks, 2019, https://www.nas.nasa.gov/publications/npb.html, (Accessed 13 July 2020).

[19] H. Zhou, Y. Mhedheb, K. Idrees, C.W. Glass, J. Gracia, K. Fürlinger, DART-MPI: An MPI-based implementation of a PGAS runtime system, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, 2014, pp. 1–11.

[20] H. Zhou, J. Gracia, R. Schneider, MPI collectives for multi-core clusters: Optimized performance of the hybrid MPI+ MPI parallel codes, in: Proceedings of the 48th International Conference on Parallel Processing: Workshops, ACM, 2019, p. 18.

[21] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.

[22] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, J.J. Dongarra, Performance analysis of MPI collective operations, Cluster Comput. 10 (2) (2007) 127–143.

[23] MPICH, 2020, https://www.mpich.org/, (Accessed 13 July 2020).

[24] Intel MPI, 2020, https://software.intel.com/en-us/mpi-library, (Accessed 13 July 2020).

[25] Open MPI: Open source high performance computing, 2020, https://www.open-mpi.org/, (Accessed 13 July 2020).

[26] K. Hasanov, Hierarchical Approach to Optimization of MPI Collective Communication Algorithms, (Ph.D. thesis), University College Dublin, Ireland, 2015.

[27] H. Zhu, D. Goodell, W. Gropp, R. Thakur, Hierarchical collectives in MPICH2, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2009, pp. 325–326.

[28] J.L. Träff, A. Rougier, MPI Collectives and datatypes for hierarchical all-to-all communication, in: EuroMPI/ASIA, ACM, 2014, p. 27.

[29] R.L. Graham, G.M. Shipman, MPI Support for multi-core architectures: Optimized shared memory collectives, in: A.L. Lastovetsky, M.T. Kechadi, J. Dongarra (Eds.), PVM/MPI, in: Lecture Notes in Computer Science, vol. 5205, Springer, 2008, pp. 130–140.

[30] A.R. Mamidala, R. Kumar, D. De, D.K. Panda, MPI Collectives on modern multicore clusters: Performance optimizations and communication characteristics, in: CCGRID, IEEE Computer Society, 2008, pp. 130–137.

[31] S. Li, T. Hoefler, M. Snir, NUMA-Aware shared-memory collective communication for MPI, in: M. Parashar, J.B. Weissman, D.H.J. Epema, R.J.O. Figueiredo (Eds.), HPDC, ACM, 2013, pp. 85–96.

[32] A.R. Mamidala, A. Vishnu, D.K. Panda, Efficient shared memory and RDMA based design for mpi_allgather over infiniband, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2006, pp. 66–75.

[33] Y. Qian, A. Afsahi, RDMA-Based and SMP-aware multi-port all-gather on multi-rail qsnet^{II} SMP clusters, in: ICPP, IEEE, 2007, p. 48.

[34] A. Mahéo, P. Carribault, M. Pérache, W. Jalby, Optimizing collective operations in hybrid applications, in: J. Dongarra, Y. Ishikawa, A. Hori (Eds.), EuroMPI/ASIA, ACM, 2014, p. 121.

[35] W. Xiong, S. Park, J. Zhang, Y. Zhou, Z. Ma, Ad hoc synchronization considered harmful, in: OSDI, vol. 10, 2010, pp. 163–176.

[36] J.L. Träff, Relationships between regular and irregular collective communication operations on clustered multiprocessors, Parallel Process. Lett. 19 (01) (2009) 85–96.

[37] J. Pjesivac-Grbovic, Towards automatic and adaptive optimizations of MPI collective operations, 2007.

[38] H.P.C. Advisory Council, HYCOM performance benchmark and profiling, Tech. Rep., 2010.

[39] R.A. Van de Geijn, J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, Concurrency, Pract. Exp. 9 (4) (1997) 255–274.

[40] R. Salakhutdinov, A. Mnih, Bayesian probabilistic matrix factorization using Markov chain Monte Carlo, in: Proceedings of the International Conference on Machine Learning, 2008, vol. 25.

[41] T.V. Aa, I. Chakroun, T. Haber, Distributed Bayesian probabilistic matrix factorization, in: CLUSTER, IEEE Computer Society, 2016, pp. 346–349.