Optimised all-to-all communication on multicore architectures applied to FFTs with pencil decomposition

Andreas Jocksch, Matthias Kraushaar
CSCS, Swiss National Supercomputing Centre
Via Trevano 131, 6900 Lugano, Switzerland
Email: andreas.jocksch@cscs.ch

David Daverio
Centre for Theoretical Cosmology, Department of Applied Mathematics and Theoretical Physics, University of Cambridge,
Wilberforce Road, Cambridge CB30WA, United Kingdom

***Abstract*—All-to-all communication is a basic functionality of parallel communication libraries such as the Message Passing Interface (MPI). Typically there are multiple different algorithms underlying which are chosen according to message size. We propose a communication algorithm which exploits the fact that modern supercomputers combine shared memory parallelism and distributed memory parallelism. The application example of our algorithm is FFTs with pencil decomposition. Furthermore we propose an extension of the MPI standard in order to accommodate this and other algorithms in an efficient way.**

***Keywords*-all-to-all communication; multicore; FFTs; MPI;**

I. Introduction

[1] Modern supercomputers are typically composed of many nodes connected with a fast network. The nodes are equipped with one or more multicore CPUs which share memory. In addition the nodes might have accelerators, such as Graphic Processing Units (GPUs). Popular programming models for such systems are the hybrid one, where on the node thread parallelism with, e.g., OpenMP [1] is applied and between the nodes the Message Passing Interface (MPI) [2] is used, and pure MPI, where cores on the same node also communicate with each other using MPI; the latter shall be the focus in this contribution. If GPUs are present they are often programmed hybridly with MPI parallelism between the nodes and on the node and OpenACC or CUDA for additional GPU thread parallelism on the node. The availability of well performing communication libraries is a key requirement. Collective communication operations are part of the MPI library. They provide fast message transfers for certain communication patterns of groups of MPI tasks by the means of optimised algorithms [3].

All-to-all personalised communication has been the topic of many optimisation studies for different network topologies, tree, mesh, hypercube and fully connected networks [4]. Our study focuses on fully connected networks. We assume that the transmission time of the interconnect can be described by a latency and a bandwidth. Thus messages can be classified according to their size as small, latency-dominated ones and large, bandwidth-dominated ones. The performance of the communication is influenced by the choice of the scheduling algorithm [5]. A simple example of a scheduling algorithm is cyclic scheduling, but more sophisticated scheduling schemes exist [6]. Another aspect is that for small messages and all-to-all communication it is efficient to apply store and forward algorithms such as the butterfly algorithm and Bruck's algorithm [7] which reduce the overall number of messages but increase the overall message size sent.

Current MPI implementations contain such algorithms; though MPICH (upon which Cray's MPI implementation is build), OpenMPI and MVAPICH apply Bruck's algorithm with all cores equally connected. We expect that the reason for this simplified implementation is the challenge of incorporating a more complex solution in the current MPI standard. For the more straightforward case of the collective operations allreduce, reduce, broadcast, scatter and gather, shared memory parallelism on the node has been exploited [8], [9].

In a previous paper we presented the special case of so-called multiple all-to-all communication, which occurs, e.g., for pencil decomposed FFTs, has been investigated for very short messages [10] but has not yet been implemented in MPI libraries. For the application FFTs of this case a library has been developed which exploits shared memory on the node, with only one task per node communicating with other nodes [11].

In this contribution, we consider the store and forwarding approach for all-to-all communication and the vector version of this communication all-to-allv with the properties of a hierarchical connection of processing units, i.e., shared memory on the nodes and a network between the nodes. Also, the general case of multiple all-to-all communication and multiple all-to-allv communication is considered for pencil decomposed FFTs with application in the cosmology code LATfield2 [12]. In order to accommodate the time intensive setup phase of the algorithm "plan" routines for the different cases are introduced which are called before the actual execution, which is frequently repeated. The all-to-all

[1]This paper has been accepted and will be published as an article in a special issue of Concurrency and Computation Practice and Experience on the Cray User Group 2018.

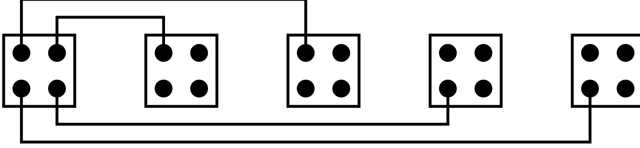


Figure 1. Communication between nodes

communication algorithm is also implemented for direct data transfer from the GPU of one node to the GPU of another node.

II. The hybrid all-to-all algorithm

A. The basic variant

We focus on the standard all-to-all communication where every process sends to every process a message of equal size. For this, we collect the messages on the node with a shared memory data segment. In a first step, which is executed locally on the nodes only, all messages from the source buffers which have the same source node and the same destination node are placed together in the shared memory data segment. The second step is the communication between the nodes from the source shared memory data segment to the destination shared memory data segment. Multiple cores of the node (the number being a parameter of the procedure) are used for the communication. Figure 1 shows an example with 5 nodes and 4 cores per node where all cores are used, the communication is illustrated for the first node only, send and receive is performed between the same partners. The collective operation might involve more nodes than cores per node plus one. In this case we try to use the optimum algorithm for the data exchange between the nodes (Sec. II-B). In the third step, the data from the shared memory is distributed locally to the target buffers of the cores of the node.

In our implementation we use non-blocking point-to-point communication which allows for the overlap of data copies and communication, and the communication of cores with multiple partners at the same time. Together with the necessary barriers the summarised algorithm is

1) MPI_Irecv
2) memcpy sendbuffer to shared sendbuffer
3) node_barrier
4) MPI_Isend
5) memcpy shared sendbuffer to shared receivebuffer, for data which remains on the node
6) node_barrier
7) MPI_Waitall
8) memcpy shared receivebuffer to receivebuffer

The scheduling of the sends and receives on the node is random between the different cores of the node. This is given by the race conditions between the different cores. The alternative would be to use only one core per node for sending and receiving between nodes. This shows to be slower (Sec. IV-B). We chose a cyclic scheduling with throttling with respect to the cores of the node, which might communicate more than one message (Sec. III-A).

For the communication part on the node, the time complexity is linear with respect to the number of cores per node. The time spent for communication between N nodes computes as for standard all-to-all communication and depends on the message size. For small messages it is $logN$, for large messages it is N^2. The algorithm requires temporary memory on the node. It is

$$mem = message_size \cdot cores_per_node^2 \cdot number_of_nodes \quad (1)$$

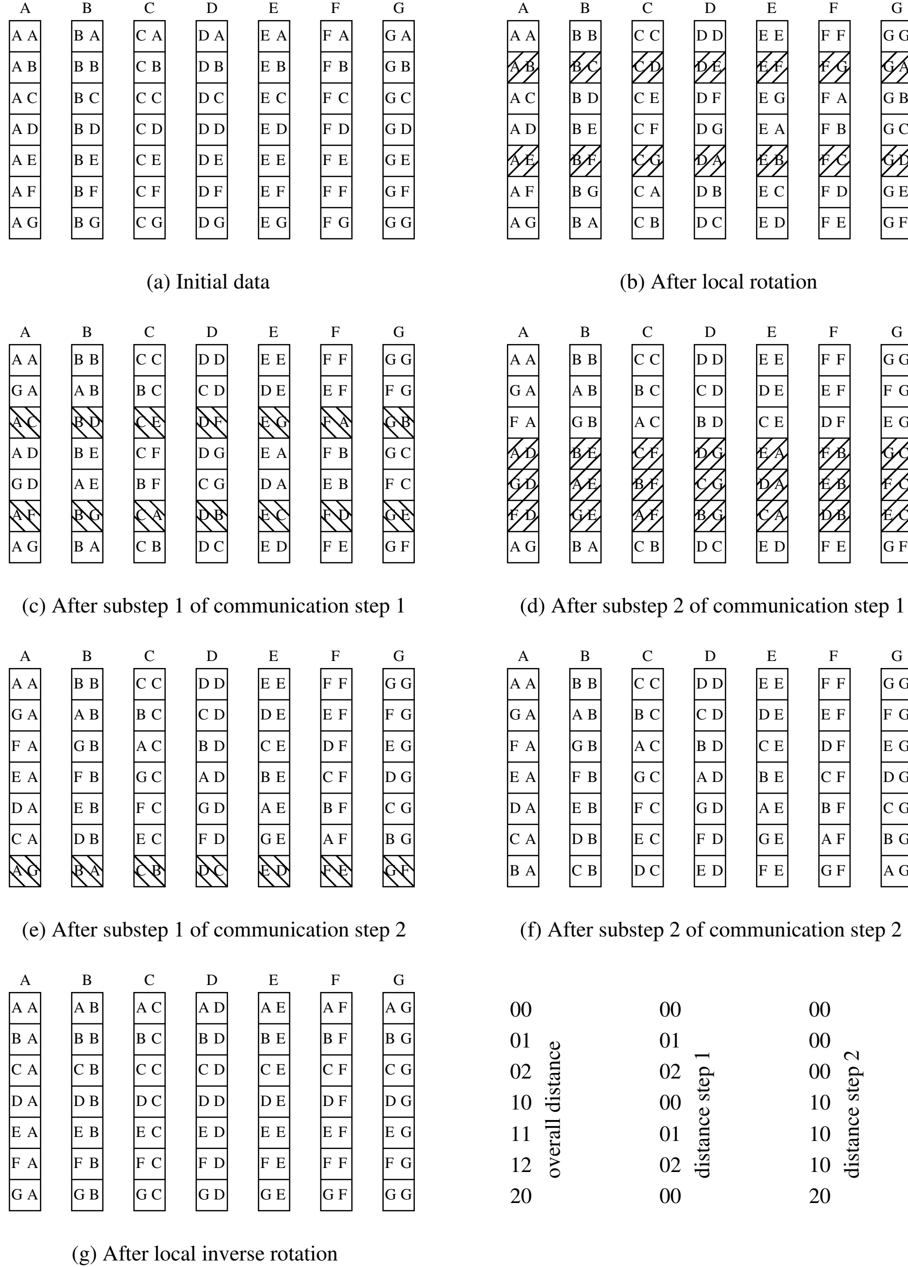
B. Bruck's algorithm

Irrespective of the application of the basic message bundling algorithm, the messages to be sent between the nodes might still be small. Thus for the message passing between nodes we apply Brucks's algorithm which covers short messages or long messages depending on its parameters. The algorithm is designed for multiple communication ports per nodes which is roughly our case of multiple cores per node. Figure 2 illustrates the algorithm. Its execution is done in the following phases, steps and substeps: In phase one the data is rearranged such that the position of the data local on the node corresponds to the distance of the target node assuming a cyclic node arrangement. In phase two the data is shifted cyclic to the target node, generally in multiple steps. In phase three the data local on the node is rotated inverse to the final destination.

Phase one and three are included in our copy of the data to the shared memory segment and from the shared memory segment. The number of ports is not necessarily the number of cores. For long messages the number of ports should equal the number of nodes minus one; every core performs multiple substeps.

In phase two the distance and the order of steps and substeps is determined based on a radix r equal to the number of ports plus one. If the distance is expressed as a number with this radix, the position of the digit is the step number and the number with all other digits zeroised gives the distance of the substep (Fig. 2 bottom right). For our case in Fig. 2 the number of ports is two and thus the radix is equal to three.

The data arrangement of the algorithm is ideal for message passing systems which do not have a penalty for discontiguous data sent by point-to-point communication. We assume that for discontiguous data a call of memcopy is necessary in order to make it contiguous, the same applies for the reversed operation. Whilst the MPICH (3.3a2) implementation of Bruck's algorithm uses MPI_Sendrecv with discontiguous datatypes, we do not take the scheme of the algorithm literally. We perform a minor optimisation and



(a) Initial data

(b) After local rotation

(c) After substep 1 of communication step 1

(d) After substep 2 of communication step 1

(e) After substep 1 of communication step 2

(f) After substep 2 of communication step 2

(g) After local inverse rotation

Figure 2. Bruck's algorithm, nodes A-G, AB indicates the message from A to B

use one send and two alternating receive buffers and arrange the data in the buffers contiguously. The receive buffers need to be copied back to the send buffer but this is one copy call compared to two implicit copy call for discontiguous datatypes.

For the special case $r = N - 1$ the algorithm does not perform message forwarding and reduces to the basic variant (Sec. II-A). As already mentioned, the number of ports in the algorithm is a parameter not connected with the number of cores per node.

C. The multiple all-to-all algorithm

For FFTs with pencil decomposition (for a recent summary about parallel FFTs see Ref. [13]), tasks communicate in multiple groups independently from each other [10]. Figure 3 shows a pencil decomposed 3D cubical domain with equal load for every core. Thin lines represent the borders of the cores and thick lines represent the borders of the nodes (3×4 cores topology on the nodes). Thus multiple all-to-all communications applied to the same nodes can be merged. We again apply the strategy of collecting messages

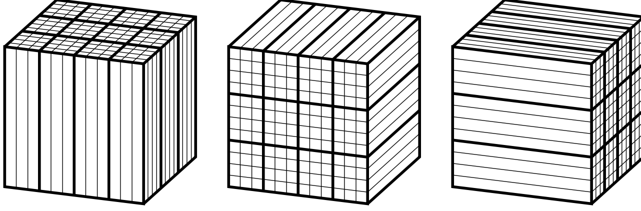


Figure 3. 2D (pencil) decomposition of the FFT

on the source node, to communicate them with different cores, and to distribute them on the destination node.

The memory usage of the algorithm on every node is

$$\begin{aligned} mem = message_size \\ \cdot cores_per_node_in_comm_direction^2 \\ \cdot cores_per_node_normal_to_comm_direction \\ \cdot number_of_nodes \end{aligned} \quad (2)$$

In more detail this communication pattern corresponds, e.g., to a 3D complex-to-complex FFT. For a 3D real-to-complex FFT and 3D complex-to-real FFT, as applied in LATfield2, one has to consider the additional data point which is created by padding the first (and the last for even resolutions) coefficient to a complex valued one. Thus a real space $n \times n \times n$ cube becomes a $(n/2+1) \times n \times n$ cube in spectral space [14]. Mostly the unequal message sizes are handled with the vector version of all-to-all, MPI_Alltoallv, or with MPI_Alltoall and padding of the messages to equal size. Since the all-to-allv communication pattern is hard to optimise without a plan routine, MPI libraries implement it typically with direct point-to-point communication without any message forwarding. This is effective for large messages. For very short messages it is more efficient to pad them to equal size and apply the algorithms used for all-to-all communication with equal message size. In our case the all-to-allv pattern complicates only the plan routine, the execution routine of the communication is straightforward.

III. Implementation

A. Blocking all-to-all communication

The choice of the communication algorithm and its parameters is dependent on the message sizes and the number of nodes/cores participating in the communication. For very small messages we apply message forwarding. Small messages are sent directly between nodes while the shared memory segment is used. For medium and large messages we refer to the original all-to-all implementation of the MPI library, one reason is that the buffer does not fit on the node for large message sizes.

In both cases, with and without message forwarding the single cores might send more than one message. Thus a proper scheduling scheme has to be chosen. We use a cyclic one, where every rank send in steps to the ranks with increasing rank number with respect to itself, applying a modulo operation with the total number of ranks. The corresponding receive operations are scheduled in decreasing order starting from the receivers rank number. As done in the reference implementation of MPICH, we apply a throttling of the communication by limiting the number of messages sent and received at the same time.

We introduce plan routines for specific communication patterns such as all-to-all and all-to-allv, which are called if this pattern is called repeatedly. The plan routine takes the send and receive buffer addresses and sizes, and the MPI communicator, as arguments. If the decision of the plan routine is to use our approach, a code generator encodes the algorithm in an array which is stored. A handle is given back which can be used to execute the code and if not needed any more to free the array and the associated data. The execution routine reads from the array and decodes the data and actually executes the algorithm. Thus every algorithm could be in the array as, e.g., the butterfly algorithm. An enrichment of the code base with commands to execute, e.g., cudamemcpy for GPU-direct (Sec. III-C) has no negative impact on performance.

For equal message sizes the all-to-allv routine produces the same code as the all-to-all routine, thus the execution time is equal.

The point-to-point communication that our algorithm is based on is performed with MPI_Isend, MPI_Irecv, and MPI_Wait. The alternative use of MPI_Sendrecv has not shown any significant difference in performance.

B. Non-blocking all-to-all communication

For the overlapping of communication and computation done for FFTs [15] and other applications a non-blocking version of our all-to-all algorithm is desirable. We introduce here a simple non-blocking version of it. If the special case of no message forwarding and no throttling is considered, the extension to non-blocking all-to-all communication is straightforward, since only one MPI_Waitall is present. Thus we split the execution in two parts. The part before the waitall starts the communication and the one from the waitall finishes it.

C. GPU-direct support

The MPI library of Piz Daint (Swiss National Supercomputing Centre, Cray XC50, 12 CPU cores and 1 NVIDIA Tesla P100 per node) supports GPU memory addresses for the data of the point-to-point and collective communications (GPU-direct) [16]. For our all-to-all and all-to-allv version we also support this feature. We assume that multiple MPI tasks use a single GPU. Two implementations are done: For very small messages – Brucks's algorithm is applied with multiple steps – all the data is transferred from the GPU memory to the CPU memory, communicated between the nodes, and transferred from the CPU memory to the GPU

memory. These communications are done with standard cudamemcpy. Larger messages (one step of Bruck's algorithm) are bundled directly on the GPU with so-called interprocess communication, communicated between the nodes, and distributed on the target GPU. The bundle and distribution operations are done with a CUDA kernel

```
1  __global__ void
2  cudacopykernel (char *data)
3  {
4    int size, num, max_size, index,
        offset, i = blockIdx.x *
        blockDim.x + threadIdx.x;
5    char *ldata, *p1, *p2;
6    ldata = data;
7    num = code_get_int (&ldata);
8    max_size = code_get_int (&ldata);
9
10   if (i < num * max_size)
11     {
12       index = i / max_size;
13       offset = i % max_size;
14       ldata += index * (sizeof (char *)
            * 2 + sizeof (long));
15       p1 = (char *) code_get_pointer
            (&ldata);
16       p2 = (char *) code_get_pointer
            (&ldata);
17       size = code_get_long (&ldata);
18       if (offset < size)
19   {
20     p1[offset] = p2[offset];
21   }
22     }
23 }
```

which copies from many source to many destination buffers at the same time in order to achieve a high occupancy of the GPU. While in the first approach the point-to-point communication is done with CPU addresses, in the second approach GPU addresses are used and thus the GPU-direct capability of the underlying MPI library is mandatory.

IV. Usage and performance

A. Application programming interface

The communication routines are written in ANSI C and have different interfaces, a native one where the algorithm with its parameters is accessed directly and an automatic one where only the problem is given and the wrapper chooses the algorithm – standard MPI or our ones – and the algorithm's parameters. The options of the latter can also be overwritten by environmental variables. All routines have C and Fortran interfaces; for brevity we show only the C interfaces here. The native one is:

```
int EXT_MPI_Alltoall_init_native
    (void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int cores_per_node_row,
    MPI_Comm comm_column, int
    cores_per_node_column, int
    num_ports, int num_active_ports,
    int chunks_throttle);
```

This function initialises the all-to-all communication. sendbuf points to the send buffer and sendcount specifies the number of entries of the MPI datatype sendtype. Likewise recvbuf, recvcount and recvtype specify the receive buffer, the number of elements to receive and the datatype, respectively. The MPI communicator comm_row represents the elements in the direction of the all-to-all communication and cores_per_node_row is the number of cores per node in this direction. If multiple all-to-all communications are to be executed in parallel comm_column is the communicator perpendicular to it and cores_per_node_column is the number of cores in this direction. The argument comm_column can be MPI_COMM_NULL. Contrary to the standard MPI functions for in-place operations sendbuf and recvbuf are just identical. The number of ports of Bruck's algorithm is specified with num_ports. The number of cores per node which participate in the node-to-node communication is set with num_active_ports. The number of chunks of the throttling are given by chunks_throttle. The return value of the function is the handle for the actual execution. Negative handles indicate an error.

```
int EXT_MPI_Alltoallv_init_native
    (void *sendbuf, int *sendcounts,
    int *sdispls, MPI_Datatype
    sendtype, void *recvbuf, int
    *recvcounts, int *rdispls,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int cores_per_node_row,
    MPI_Comm comm_column, int
    cores_per_node_column, int
    num_active_ports, int
    chunks_throttle);
```

This function is the vector version of the previous one. Again sendbuf specifies the address of the send buffer. The argument sendcounts is a vector of counts for the single elements, sdispls a vector with its offsets and sendtype is the MPI datatype. Likewise recvbuf, recvcounts, rdispls and recvtype specify these values for the elements to be received. The arguments comm_row, cores_per_node_row, comm_column, cores_per_node_column, num_active_ports,

chunks_throttle and the return value are identical to their counterpart in EXT_MPI_Alltoall_init.

```
void EXT_MPI_Ialltoall_init_native
    (void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int cores_per_node_row,
    MPI_Comm comm_column, int
    cores_per_node_column, int
    num_active_ports, int
    *handle_begin, int *handle_wait);
```

This is the non-blocking version of EXT_MPI_Alltoall_init. All parameters are identical except that it returns two handles for the last two argument pointers. The actual execution of the first handle begins the communication and the execution of the second handle acts as a wait until all communication is finished.

```
void EXT_MPI_Ialltoallv_init_native
    (void *sendbuf, int *sendcounts,
    int *sdispls, MPI_Datatype
    sendtype, void *recvbuf, int
    *recvcounts, int *rdispls,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int cores_per_node_row,
    MPI_Comm comm_column, int
    cores_per_node_column, int
    num_active_ports, int
    *handle_begin, int *handle_wait);
```

This is the non-blocking version of EXT_MPI_Alltoallv_init. Also here all parameters are identical except that two handles for the last two argument pointers are returned. The handles have the function as described for the previous function.

```
int EXT_MPI_Alltoall_exec_native (int
    handle);
```

This function executes the communication specified by handle. For the two handles of non-blocking communication it needs to be called separately. A negative return value indicates an error.

```
int EXT_MPI_Alltoall_done_native (int
    handle);
```

This function releases the handle given as argument. Note that for the non-blocking versions of the initialisation routines both handles must be released separately. Also here a negative return value indicates an error.

For this native interface the memory buffers are shared between handles. If for a new handle request larger buffers are required than currently allocated they are adapted. In case of the release of handles the memory size is not adapted. If the last handle is released all buffers are released and the environment is at the state before the first use. Special care has to be taken that it is not possible to perform non-blocking communication at the same time as any other communication, including non-blocking itself. Also the cores on the node participating in the communication have to be the same for all handles. If these requirements are not fulfilled the wrapper interface needs to be used. After the call of the initialisation routines the MPI communicators can be destroyed; they are not needed for the execution of the communication.

The wrappers which choose between our routines and the standard MPI ones are:

```
1 int EXT_MPI_Alltoall_init_general
    (void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int
    my_cores_per_node_row, MPI_Comm
    comm_column, int
    my_cores_per_node_column, int
    *handle);
2 int EXT_MPI_Alltoallv_init_general
    (void *sendbuf, int *sendcounts,
    int *sdispls, MPI_Datatype
    sendtype, void *recvbuf, int
    *recvcounts, int *rdispls,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int
    my_cores_per_node_row, MPI_Comm
    comm_column, int
    my_cores_per_node_column, int
    *handle);
3 int EXT_MPI_Ialltoall_init_general
    (void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int
    my_cores_per_node_row, MPI_Comm
    comm_column, int
    my_cores_per_node_column, int
    *handle_begin, int *handle_wait);
4 int EXT_MPI_Ialltoallv_init_general
    (void *sendbuf, int *sendcounts,
    int *sdispls, MPI_Datatype
    sendtype, void *recvbuf, int
    *recvcounts, int *rdispls,
```

```
    MPI_Datatype recvtype, MPI_Comm
    comm_row, int
    my_cores_per_node_row, MPI_Comm
    comm_column, int
    my_cores_per_node_column, int
    *handle_begin, int *handle_wait);
5   int EXT_MPI_Alltoall_init (void
    *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm, int *handle);
6   int EXT_MPI_Alltoallv_init (void
    *sendbuf, int *sendcounts, int
    *sdispls, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts,
    int *rdispls, MPI_Datatype
    recvtype, MPI_Comm comm, int
    *handle);
7   int EXT_MPI_Ialltoall_init (void
    *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm, int *handle_begin, int
    *handle_wait);
8   int EXT_MPI_Ialltoallv_init (void
    *sendbuf, int *sendcounts, int
    *sdispls, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts,
    int *rdispls, MPI_Datatype
    recvtype, MPI_Comm comm, int
    *handle_begin, int *handle_wait);
9   int EXT_MPI_Alltoall_exec (const void
    *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm, int handle);
10  int EXT_MPI_Alltoallv_exec (const
    void *sendbuf, const int
    *sendcounts, const int *sdispls,
    MPI_Datatype sendtype, void
    *recvbuf, const int *recvcounts,
    const int *rdispls, MPI_Datatype
    recvtype, MPI_Comm comm, int
    handle);
11  int EXT_MPI_Ialltoall_begin (const
    void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void
    *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm
    comm, MPI_Request *request, int
    handle_begin);
12  int EXT_MPI_Ialltoall_wait
    (MPI_Request *request, int
    handle_wait);
13  int EXT_MPI_Ialltoallv_begin (const
    void *sendbuf, const int
    *sendcounts, const int *sdispls,
    MPI_Datatype sendtype, void
    *recvbuf, const int *recvcounts,
    const int *rdispls, MPI_Datatype
    recvtype, MPI_Comm comm,
    MPI_Request *request, int
    handle_begin);
14  int EXT_MPI_Ialltoallv_wait
    (MPI_Request *request, int
    handle_wait);
15  int EXT_MPI_Alltoall_done (int
    handle);
```

For all routines the return value is the error code, zero means no error. The handle is the last argument or in case of the initialisation the last two arguments are the handles. There are no restrictions with respect to the routines used at the same time, the additional memory required by the communication algorithm is in case of conflicts between different handles – non-blocking communication or different cores on the nodes are part of all-to-all – allocated separately. For the _general routines the number of cores in communication direction and perpendicular to it can be specified, alternatively routines are provided which set the maximum number of cores available for the communication direction and one perpendicular to it.

B. Benchmarks

Most of our benchmarks were performed on the GPU based partition of Piz Daint, or on Grand Tave (Cray XC 40 KNL) on which we used 12 cores per KNL (for consistency with the 12 cpu cores of Piz Daint, the cache memory with quadrant clustering was used). We assume the same qualtitative behaviour in the performance for both machines since they have the same interconnect. Figure 4 shows the timings for standard all-to-all communication for different node numbers and message sizes (Piz Daint). For 10 and 100 bytes our algorithm is always faster than the Cray MPICH (version 7.6.0) implementation. For 1000 bytes (and larger, not shown) our algorithm is slower than the reference ones. The bottleneck of the execution for these medium message sizes is the synchronisation on the node, where the scheduling of the messages seems to lead to long waiting times, visible by a large time spent in the node barrier. This can be mitigated by splitting the node of 12 cores in multiple smaller ones, thus the messages between the virtual nodes become smaller. In the limit of one core per node, one obtains approximately the performance of the MPICH all-to-

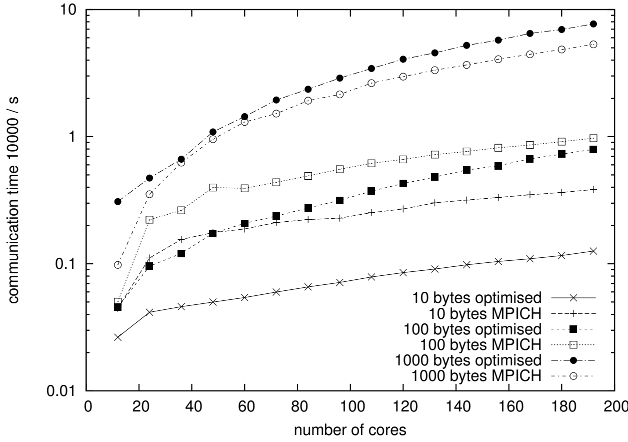


Figure 4. Performance comparison for standard all-to-all, 12 nodes, 12 cores per node

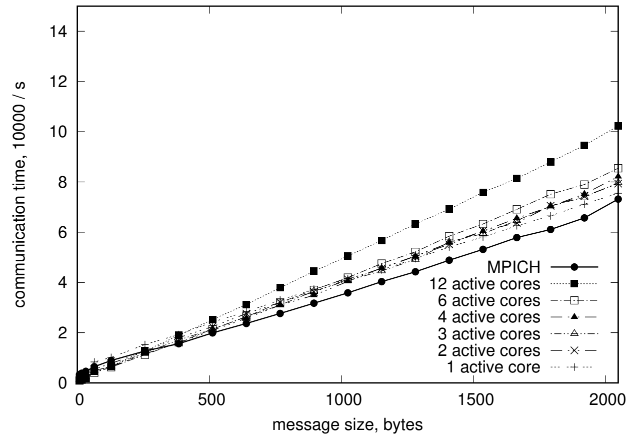


Figure 6. Performance comparison for standard all-to-all, variation of active number of cores, 12 nodes, 12 cores per node

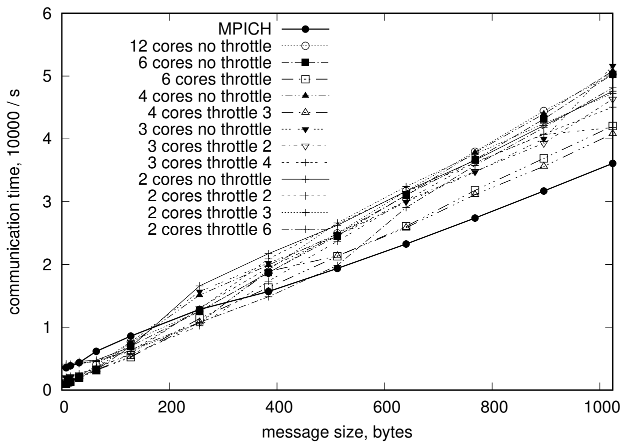


Figure 5. Performance comparison for standard all-to-all, variable tile size, 12 nodes, 12 cores per node

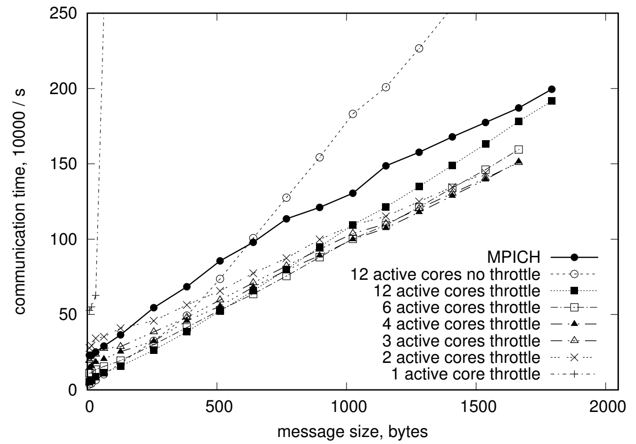


Figure 7. Performance comparison for standard all-to-all, 156 nodes, 12 cores per node

all communication; the additional cost of memcpy depending on the implementation is small. An alternative would be to keep the node and to break the messages, e.g., for prime core numbers. Figure 5 shows the performance for different sizes of the tiles. Small tiles are better for medium message sizes, large tiles are advantageous for small message sizes. We also apply the throttle to the communication, which means multiple waits in the execution (Sec. III-A). It is mostly the best to apply as many wait operations as possible. A further parameter is the number of cores used in the communication (Fig. 6). For this, the results show, that a certain reduction of the number of active cores improves the performance for medium message sizes while for small message sizes the performance is the best if all cores of the node are used.

The performance study using a larger number of nodes (Figs. 7, 8, Grand Tave) reveals that with increasing node number our agorithm is faster than the one of MPICH (version 7.6.2 on Tave), 12 nodes (see Fig. 4-6) seems to be a minimum of performance with respect to MPICH.

We also performed tests on a second Cray platform, the CS-Storm system Piz Kesch of Meteoswiss (Mellanox FDR InfiniBand interconnect), where we used all five postprocessing nodes (Fig. 9). For all message sizes measured, from 8 bytes to 2048 bytes, our routine is faster than the one from the MVAPICH2 2.2 library.

For the GPU to GPU internode communication (Sec. III-C) without message forwarding the second approach of interprocess communication on the GPU is faster than the cudaMemcpy (not shown). The benchmarks show (Fig. 10) that without message forwarding the second approach is faster. Even for larger message sizes our all-to-all implementation is faster than the one provided by the standard MPI library. There seems to be no scheduling problem since the messages are presumably sent in chunks.

The timings for the special case of multiple all-to-all communication are shown in Fig. 11. The cores are arranged

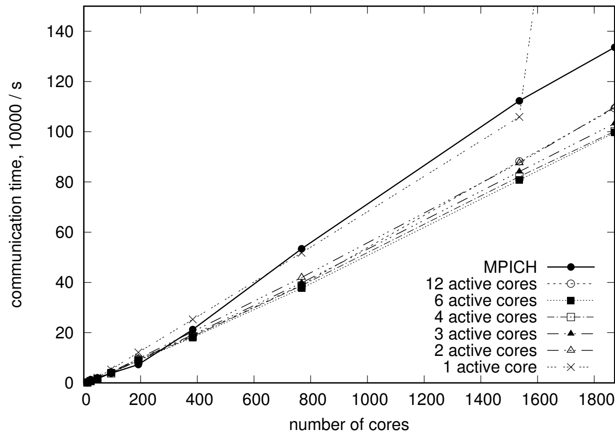


Figure 8. Performance comparison for standard all-to-all, 1024 bytes, 12 cores per node, throttle

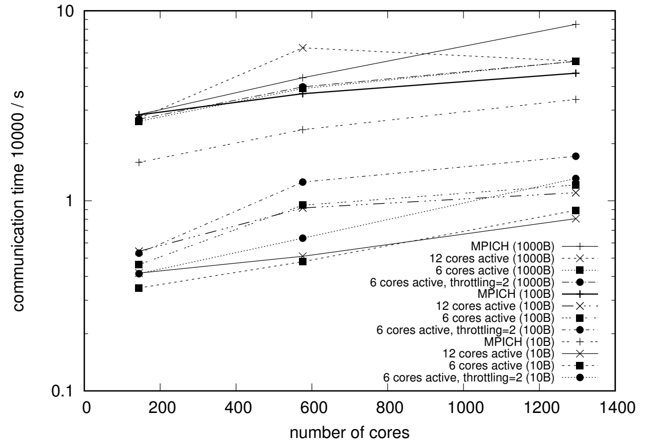


Figure 11. Performance comparison for multiple all-to-all 3×4 cores

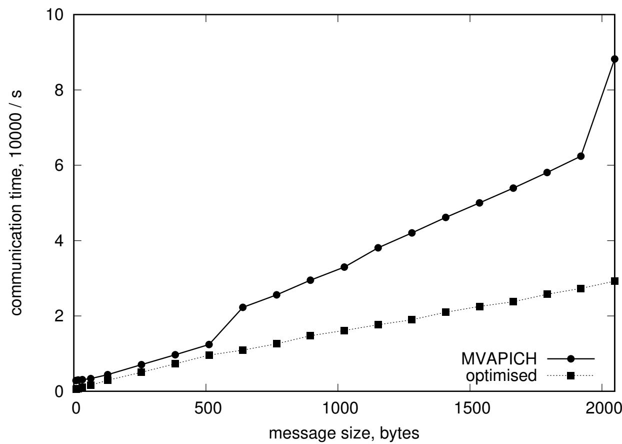


Figure 9. Performance of all-to-all on Piz Kesch, 5 nodes, 12 cores per node

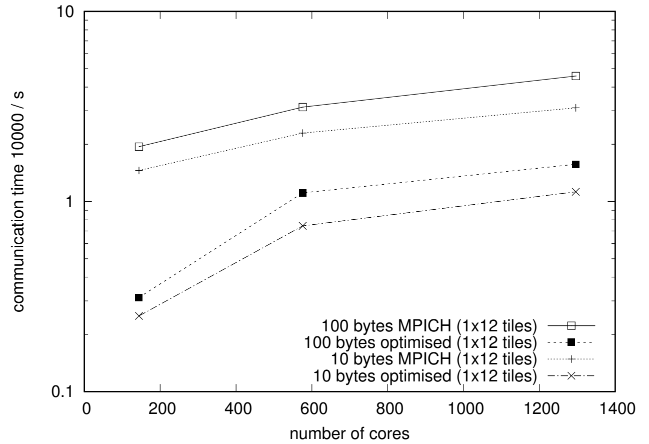


Figure 12. Performance comparison for multiple all-to-all 1×12 cores

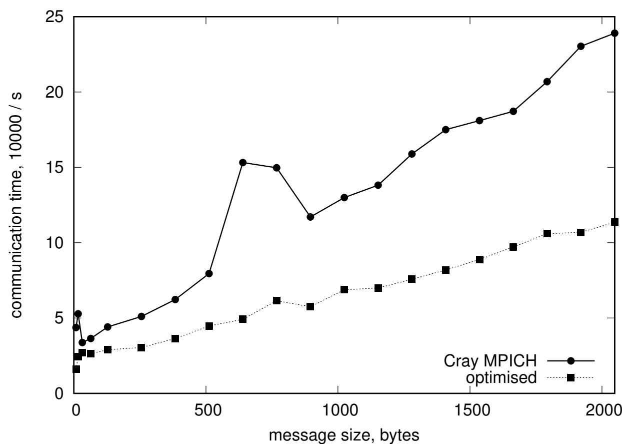


Figure 10. Performance comparison for GPU-direct, 12 nodes, 12 cores per node

in a 3×4 topology on each node. For the message sizes 10, 100 and 1000 bytes our multiple all-to-all algorithm is faster than MPI all-to-all. Again, for larger messages (not shown) our algorithm loses its advantage in performance. For this special case of 144 cores only, a topology of 1×12 shows better performance (Fig. 12), with our algorithm being faster. However, for more than 144 cores (12 nodes) the 3×4 topology is the better choice.

V. Applications

A. *LATfield2*

LATfield2 is a c++ library aiming to allow fast development of classical field theory coupled to a n-body problem simulation on a lattice. The cosmological n-body solver gevolution [17], which is based on general relativity, was created using the LATfield2 library. Gevolution is based on spectral methods leading to a large fraction of the execution time being spent in FFTs, in a typical run approximately 40%. It is important to note the following: A typical run of

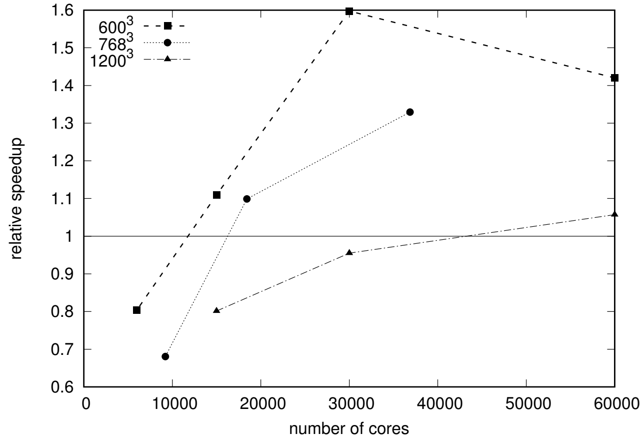


Figure 13. Relative speedup of our all-to-all communication for FFTs with respect to MPICH all-to-all communication

gevolution represents around 30 TB of evolved data, which should be kept in memory in order to avoid intensive i/o operations. This implies a large number of nodes participating in the computation with a mesh of moderate size. Thus the choice of the pencil decomposition for FFTs. The scaling of the code is limited by the scaling of the FFTs. Therefore improving the all-to-all execution time is extremely relevant for such simulations.

Figure 13 presents the benchmarks of a forward real-to-complex Fourier transform followed by a backward one. Three different lattices sizes were used: 600^3, 768^3 and 1200^3 and the parameters of our all-to-all communication routine were 3×4 cores per node with all cores being active, the throttle was set to three waits per communication. The results in Fig. 13 show that one can get an improvement of the speedup of up to 60% for the FFTs when using the algorithm proposed in this paper. This represents a 25% speedup for a simulation run with gevolution.

B. ORB5

Another potential application currently being investigated is the plasma physics code ORB5 [18]. This code is a gyrokinetic particle in cell code where particles can move between all subdomains of a so-called clone of the mesh. Multiple clones exist and many different complex communication patterns are present in the code. The metadata (a few bytes) for the amount of particles communicated between subdomains is exchanged with MPI_Alltoall. Since the placement of the ranks on the clones is typically such that for specific subdomains all clones are on the same node and the all-to-all is executed for all clones it is a multiple all-to-all. Whether multiple all-to-all or normal all-to-all is the most beneficial approach needs to be tested since the multiple all-to-all introduces an unwanted synchronisation between clones. The particle data itself is communicated with MPI_Isend, MPI_Irecv and MPI_Waitall between subdomains. These calls could be integrated in our multiple all-to-allv routine, which needs some further optimisation for this particular case, in order to accommodate changing message sizes between the calls.

VI. Implications for communication libraries

The implementation of the proposed single all-to-all algorithm in the current MPI standard [2] would be to use the standard communicators. However, this has the consequence of a slowdown of communicator creation in the initialisation of the library and in routines such as MPI_Comm_split, since for every new communicator the setup phase of the algorithm needs to be executed regardless of its actual usage. For the multiple all-to-all algorithm it is even more complicated. The standard communicators do not cover such an algorithm. One would need to implement it for a “graph communicator” as “neighbourhood collective communication” [19]. The communicator creating routine would need to detect the multiple all-to-all communication pattern [20]. This is in general computationally expensive. One solution could be the use of the MPI_Info object for the creation of the communicator in order to pass the details about the collectives to the library [21].

There are several proposals for the further development of MPI in the literature: one is the introduction of persistant collectives [22]. For that — as in our implementation — planner routines take as arguments the parameters of the standard collectives all-to-all, ..., which are repeatedly executed by calling a separate “execute” routine. We propose an extension of the persistant collectives which takes into account multiple collective communication — at least the all-to-all case.

The discussion regarding the MPI standard also applies to other standards such as HPX.

VII. Future work

The present implementation of all-to-all communication can be further improved by, e.g., the following features:

1) Flexible core count within one communicator
2) Wrapper for flexible array addresses
3) Bruck’s full algorithm for variable message size
4) Butterfly algorithm for suitable number of nodes
5) Further optimisation of the implementation of Bruck’s algorithm – decoupling of message passing
6) Better integration between FFTs and communication algorithm with a further reduction of memcopies by, e.g., computation on the shared memory segment, as done in Ref. [11].

VIII. Conclusions

We propose a short message all-to-all communication approach for networks with shared memory nodes. The current Cray MPICH implementation is outperformed a factor of 2.9

for a message size of 10 bytes. On an infiniband cluster our algorithm also outperforms the standard MVAPICH implementation for small message sizes. The implementation for GPUs is advantageous for small and medium message sizes. The application to FFTs such as those used in LATfield2 demonstrates its strength. The algorithm and its implementation could become part of any established communication library or could be used as an independent layer on top of any point-to-point communication realisation.

The source code of our communication routines is publicly available on GitHub ETH-CSCS [23].

ACKNOWLEDGEMENT

The authors would like to thank Guilherme Peretti-Pezzi, Vasileios Karakasis, Tim Robinson, John Biddiscombe (CSCS), Torsten Hoefler (ETH Zurich) and Peter Messmer (NVIDIA) for helpful discussions.

REFERENCES

[1] "OpenMP," 2015. [Online]. Available: http://www.openmp.org

[2] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard, version 3.1," 2015.

[3] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[4] P. Sanders and J. Träff, "The hierarchical factor algorithm for all-to-all communication," *Euro-Par 2002 Parallel Processing*, pp. 17–51, 2002.

[5] A. Faraj, X. Yuan, and P. Patarasuk, "A message scheduling scheme for all-to-all personalized communication on ethernet switched clusters," *IEEE Transactions on parallel and distributed systems*, vol. 18, no. 2, pp. 264–276, 2007.

[6] C. Qiao, X. Zhang, and L. Zhou, "Scheduling all-to-all connections in wdm rings," in *All-Optical Communication Systems: Architecture, Control, and Network Issues II*, vol. 2919. International Society for Optics and Photonics, 1996, pp. 218–230.

[7] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[8] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 85–96.

[9] S. Li, T. Hoefler, C. Hu, and M. Snir, "Improved MPI collectives for MPI processes in shared address spaces," *Cluster computing*, vol. 17, no. 4, pp. 1139–1155, 2014.

[10] A. Jocksch, *FFTs and Multiple Collective Communication on Multiprocessor-Node Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 163–172.

[11] N. Li and S. Laizet, "2decomp & fft-a highly scalable 2d decomposition library and FFT interface," in *Cray User Group 2010 conference*, 2010, pp. 1–13.

[12] D. Daverio, M. Hindmarsh, and N. Bevis, "Latfield2: A c++ library for classical lattice field theory," *arXiv preprint arXiv:1508.05610*, 2015.

[13] A. G. Chatterjee, M. K. Verma, A. Kumar, R. Samtaney, B. Hadri, and R. Khurram, "Scaling of a fast Fourier transform and a pseudo-spectral fluid solver up to 196608 cores," *Journal of Parallel and Distributed Computing*, vol. 113, pp. 77–91, 2018.

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[15] J. H. Göbbert, H. Iliev, C. Ansorge, and H. Pitsch, "Overlapping of communication and computation in nb3dfft for 3d fast Fourier transformations," in *Jülich Aachen Research Alliance (JARA) High-Performance Computing Symposium*. Springer, 2016, pp. 151–159.

[16] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[17] J. Adamek, D. Daverio, R. Durrer, and M. Kunz, "General relativity and cosmic structure formation," *Nature Phys.*, vol. 12, pp. 346–349, 2016.

[18] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T.-M. Tran, B. McMillan, O. Sauter, K. Appert, Y. Idomura, and L. Villard, "A global collisionless pic code in magnetic coordinates," *Comput. Phys. Commun.*, vol. 177, no. 5, pp. 409–425, 2007.

[19] T. Hoefler and J. L. Träff, "Sparse collective operations for MPI," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[20] S. H. Mirsadeghi, J. L. Träff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize mpi neighborhood collectives," in *IEEE 24th International Conference on High Performance Computing (HiPC), 2017*, 2017.

[21] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012*. IEEE, 2012, pp. 1–10.

[22] J. L. Träff, A. Rougier, and S. Hunold, "Implementing a classic: Zero-copy all-to-all communication with mpi datatypes," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 135–144.

[23] 2018. [Online]. Available: https://github.com/eth-cscs/ext_mpi_collectives