Decomposing MPI Collectives for Exploiting Multi-lane Communication

Jesper Larsson Träff and Sascha Hunold
TU Wien, Faculty of Informatics, Vienna, Austria
traff,hunold@par.tuwien.ac.at

***Abstract*—Many modern, high-performance systems increase the cumulated node-bandwidth by offering more than a single communication network and/or by having multiple connections to the network, such that a single processor-core cannot by itself saturate the off-node bandwidth. Efficient algorithms and implementations for collective operations as found in, e.g., MPI, must be explicitly designed for exploiting such multi-lane capabilities. We are interested in gauging to which extent this might be the case. We systematically decompose the MPI collectives into similar operations that can execute concurrently on and exploit multiple network lanes. Our decomposition is applicable to *all standard MPI collectives* (broadcast, gather, scatter, allgather, reduce allreduce, reduce-scatter, scan, alltoall), and our implementations' performance can be readily compared to the native collectives of any given MPI library. Contrary to expectation, our *full-lane, performance guideline implementations* in many cases show surprising performance improvements with different MPI libraries on a dual-socket, dual-network Intel OmniPath cluster, indicating a large potential for improving the performance of native MPI library implementations. Our full-lane implementations are in many cases large factors faster than the corresponding MPI collectives. We see similar results on a larger, dual-rail Intel InfiniBand cluster. The results indicate considerable room for improvement of the MPI collectives in current MPI libraries including a more efficient use of multi-lane capabilities.**

I. Introduction

Current distributed memory (HPC) parallel computers are clusters with a clear, hierarchical structure, e.g., islands consisting of racks consisting of nodes consisting of sockets of multi-core processors. Hierarchy matters for performance, as the different hierarchy levels have different communication capabilities. Communication libraries must take the hierarchy into account, and, for any set of processes, communicate at the most efficient hierarchy levels. Collective operations, as eminently found in MPI [1], can be given algorithms that take the communication hierarchy into account, and high-quality MPI library implementations should utilize the best, such hierarchical algorithms. A motivation of this paper is to investigate in a portable manner whether this might be the case.

Traditional hierarchical collective algorithms for clustered, high-performance systems focus on minimizing contention on a single communication network through which the compute nodes are connected by letting a single, node-local *root process* on each node be responsible for the communication with other nodes, see, e.g., early work on clustered, wide-area systems [2]. The collective operations of MPI can readily be decomposed into hierarchical applications of similar collective operations as described and implemented in, e.g., [3]–[6] and many other works.

Many modern high-performance systems are equipped with multiple communication networks and/or multiple connections to the network(s) from the compute nodes or otherwise provide higher off-node bandwidth than can be saturated by a single processor-core. We will refer to systems with such capabilities as k- or *multi-lane* systems with k being the number of physical or virtual lanes over which processor-cores can communicate independently and simultaneously. Our assumption is that the cumulated bandwidth of the compute nodes is increased proportionally to the number of such lanes. As an example, the smaller cluster used in this study has compute nodes with two sockets, each connected to an own OmniPath network, and MPI processes on either socket can communicate independently of processes on the other with full network bandwidth. Note that these assumptions are quite different from the standard k-ported assumption where processors can use k communication ports in a single operation, e.g., [7], [8], and algorithm design under the k-lane assumptions is different from algorithm design under traditional, k-ported assumptions.

An approach to exploit potential multi-lane communication capabilities of modern cluster systems in MPI is to let several processes, each of which are close to a network lane, communicate concurrently. For this approach to be beneficial, data to be communicated across the nodes must be effectively distributed across the communicating processes, such that the total amount of data in and out of the cluster nodes does not exceed that of a traditional, hierarchical algorithm. This approach seems to have been pioneered by the group of Panda over a number of papers [9]–[12] addressing different (but not all) collectives of the MPI standard (broadcast, allgather, alltoall, allreduce). The idea of these *multi-root algorithms* is to divide the collective operation over several virtual roots per compute node, with proportionally smaller parts of the total data per virtual root. Similar decompositions were proposed and evaluated by [13], but without always distributing the amount of data, and therefore leading to inferior, bandwidth constrained algorithms in some cases. All these papers demonstrated improvements in applications by improved implementations of the MPI collectives considered. We do not repeat such application studies.

In this paper, we explore algorithms and implementations of the MPI collectives that can possibly exploit multi-lane capabilities. For each collective, our algorithms spread the data

978-1-7281-6677-3/20/$31.00

to be communicated evenly across the processes on the nodes, and employ collective operations concurrently on each data segment. These decompositions of the collectives are our *full-lane implementations*. In the best case, where each component collective is executed on a dedicated lane, we can possibly expect a k-fold speed-up over implementations using only a single lane for communication, or implementations where the complete data are communicated by many processes on the nodes. In contrast to the papers by the group of Panda, and motivated by our findings in Section II, we consider *full-lane implementations* that spread the data to be communicated evenly across *all* (and not just a subset of the) processes on the nodes. Our implementations are very compact and use available MPI functionality only (collectives, sub-communicators, derived datatypes, collective operations).

We give full-lane implementations for *all* regular (non-vector) MPI collectives, namely `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Alltoall`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter_block`, `MPI_Scan`, and `MPI_Exscan`. These implementations are intended for MPI communicators populating the compute nodes with the *same number* of MPI processes, *ranked consecutively*. We call such communicators *regular*, but our concrete code actually works for any communicator. Regular communicators are the common case on clustered systems, since `MPI_COMM_WORLD` is usually regular. We also compare to traditional, *hierarchical decompositions*, where a single process per node is responsible for all communication with other nodes [6], [14]. Such implementations can possibly benefit from the increased bandwidth of multiple, bundled, physical lanes (for large enough data).

Our implementations of the collectives in terms of other collectives in new contexts can be viewed as performance guidelines [15], [16] that formalize expectations on the performance of the MPI collectives. A performance guideline is typically an implementation of some MPI functionality, e.g., `MPI_Bcast`, in terms of other, similar MPI functionality, e.g., `MPI_Scatter` followed by `MPI_Allgather`. A good MPI library implementation of the native `MPI_Bcast` should reasonably be expected to perform at least as well as any such guideline implementation: If not, `MPI_Bcast` could readily be replaced with the guideline implementation. Our full-lane implementations are such guideline, *mock-up implementations* [17] for regular communicators that can benefit from multi-lane capabilities. It is important to note that our mock-ups are full-fledged, correct implementations for the corresponding collectives, and can thus readily be used to (auto)tune an MPI library that exhibits performance defects. In our implementations, we furthermore use MPI derived datatypes to perform the necessary reordering of data, thus our mock-ups are in almost all cases *zero-copy* [6], [18] with no explicit data movement operations before or after the constituent collective operations.

Overall, our paper makes the following contributions:

1) it empirically demonstrates that multiple, physical lanes of parallel systems can be effectively used for point-to-point and collective communication,
2) it proposes novel algorithmic variants for multi-lane collective communication and their respective performance guidelines, and
3) it presents an extensive experimental study, showcasing the possible performance improvements over existing MPI library implementations by exploiting the multi-lane capabilities of parallel systems.

The remainder of the paper is structured as follows. In Section II, two benchmarks are used to explore whether multiple lanes can be exploited with MPI point-to-point communication, and whether multiple, physical lanes give the expected increase in cumulated communication bandwidth. Section III presents the MPI performance guideline, zero-copy, full-lane implementations, and analyze them under the most optimistic, best known assumptions on the component collective operations, which reveals bottlenecks and limitations to these implementations. Section IV gives an experimental comparison of the performance guideline implementations to the library native collectives, in many cases showing substantial and unexpected improvements.

Notation: The number of MPI processes is denoted by p, the number of compute nodes by N, and the number of MPI processes per node by n, such that $p = nN$. The number of (physical) lanes is denoted by k. All benchmarks communicate data as integers, `MPI_INT`. The amount of data *per process* is given as a count c of such integers (following MPI conventions). The total problem size is therefore often cp, e.g., for scatter/gather, allgather, alltoall and reduce-scatter operations.

II. Communication Performance with Multiple Lanes

For the evaluation of the full-lane collective algorithms, we first aim at determining whether a given system (and MPI library) can indeed support communication over more than a single lane, that is whether a k fold communication speed-up for some value of $k, k > 1$ can be achieved relative to communication by a single (MPI) process per node. For this we have run two communication benchmarks, the *lane pattern benchmark* and the *multi-collective benchmark* on two systems with multi-lane capabilities as summarized in Table I.

The first, smaller system called *Hydra* is an Intel Skylake dual-socket, dual-rail OmniPath cluster with two actual Omni-Path switches with $N = 36$ compute nodes, each with $n = 32$ cores. We also experiment on a larger InfiniBand, dual-socket, dual-rail (two HCA) cluster called *VSC-3* with about 2000 nodes and $n = 16$ cores per node For the *Hydra* system, our hypothesis is that MPI processes residing on the different sockets can communicate independently and effectively use the two independent OmniPath networks to achieve twice as high bandwidth as when only one process is communicating. Communication latency per process should stay the same. On the *VSC-3*, the two network ports can be used to better saturate the network, but possibly achieving less than double bandwidth.

TABLE I: The two systems (hardware and software), *Hydra* and *VSC-3* (see vsc.ac.at) used for the experimental evaluation.

Name	n	N	p	Processor	Interconnect	MPI library
Hydra	32	36	1152	Intel Xeon Gold 6130, 2.1 GHz Dual socket	Intel OmniPath Dual-rail, dual-switch	Open MPI 4.0.2, Intel MPI 2019.4.243 MPICH 3.3.2, MVAPICH2 2.3.3
VSC-3	16	2020	32320	Intel Xeon E5-2650v2, 2.6 GHz Dual socket	InfiniBand Dual-rail (HCA) Intel QDR-80	Intel MPI 2018

The *lane pattern benchmark* divides the processes into sets of processes for each compute node. Each compute node sends and receives a count of c data elements (of type `MPI_INT`). Sending and receiving is repeated (without any barriers) a certain number of times (here 100). This is done by the assumption that multiple lanes will be used in pipelined algorithms. A parameter k for the number of *virtual lanes* determines how the c data elements are sent and received from the compute nodes: The count c is divided evenly over the k first processes on each node, which then independently communicate these $\lfloor c/k \rfloor$ elements (with $c \bmod k$ extra elements for the first process). The process with rank i sends to process $(i + n) \bmod p$ and receives from process $(i - n) \bmod p$ using blocking `MPI_Sendrecv` operations (we have also experimented with other node communication patterns and operations, but did not observe major differences). We repeat this experiment 80 times (disposing of the first few warmup repetitions), each repetition separated by an `MPI_Barrier`. The completion time of an experiment is the completion time of the slowest process, and we report both the mean completion times over all repetitions and their 95% confidence intervals [19].

The question for the lane pattern benchmark is how many times faster the c elements per node can be communicated when sent and received over k virtual lanes per node. Our assumption is that the processes are assigned to the nodes in such a way that the number of available, physical lanes can be active simultaneously. For this, the MPI processes are assigned consecutively to the compute nodes, and are pinned alternatingly over the two sockets. In case there are k' physical lanes, we expect a k' fold speed-up, when our parameter k is chosen with $k \geq k'$.

The results for the *Hydra* cluster for $N = 36$ nodes with $n = 32$ processes with varying k and different c (chosen such that all active processes on the nodes communicate exactly the same count c/k of elements) with Open MPI 4.0.2 are shown in Figure 1. Results for *VSC-3* are qualitatively similar, and omitted here.

For very small data, there is little to almost no benefit from communication over k virtual lanes, but no latency degradation either. As data get larger, there is a large reduction in the running time of almost a factor 2 even with $k = 2$, which becomes even better and exceeding the factor 2 as k increases towards n. This is especially pronounced for the stronger *Hydra* system, but holds also for the *VSC-3* system. On both systems, there is no significant penalty by letting all n processes on the node communicate their share c/n of the data, which motivates our *full-lane implementations* of Section III where all n processes on the nodes are partaking in independent collective operations.

The *multi-collective benchmark* is used to estimate how many executions of the same collective operation (here `MPI_Alltoall`) over the lanes can be sustained concurrently at no extra cost in running time compared to only one execution. The benchmark splits the calling communicator into n communicators, each one spanning N compute nodes (see Section III). With k virtual lanes, the k first of these *lane communicators* execute the most communication intensive `MPI_Alltoall` collective with c being the total number of elements per MPI process. The measurement is repeated 80 times, and the running times are collected as explained above for the lane pattern benchmark. Our hypothesis for this benchmark is that a system with k' physical lanes can, for $k \geq k'$, sustain k' concurrent executions of the collective, that is, the running time for k concurrent executions is about k/k' times the time for one execution (preferably already with $k = k'$).

Results with the multi-collective benchmark are shown in Figures 2 and 3 for the two systems. On *Hydra*, it is noteworthy that for small counts $c = 1152$, up to $k = 8$ concurrent executions of `MPI_Alltoall` each with the same total message size c can be sustained at the same running time as only one execution. As the count grows large(r), clearly (more than) two concurrent executions can be sustained: The running time is less than k/k' times larger than the running time for one execution ($k = 1$) even with $k = k'$. On the *VSC-3*, for the small counts $c = 1600, c = 16000$, the system can sustain $k = 8, k = 4$ concurrent `MPI_Alltoall` operations. As c increases, at least $k = 2$ concurrent `MPI_Alltoall` operations can be sustained, with less than a factor of k/k' increase in running time compared to $k = 1$. Only for the very large count $c = 1600000$ the running time increase roughly matches the expected factor of 8.

Also these observations justifies the *full-lane implementations* which employ $k = n$ concurrent collective operations each on c/n of the input data.

III. Performance Guideline Implementations

In this section, we present our multi-lane implementations of the MPI collectives in terms of other, similar collective operations. Our assumption is that these implementations by construction will be able to exploit multi-lane capabilities of the cluster, similarly to the way the lane pattern and multi-collective benchmarks could do.

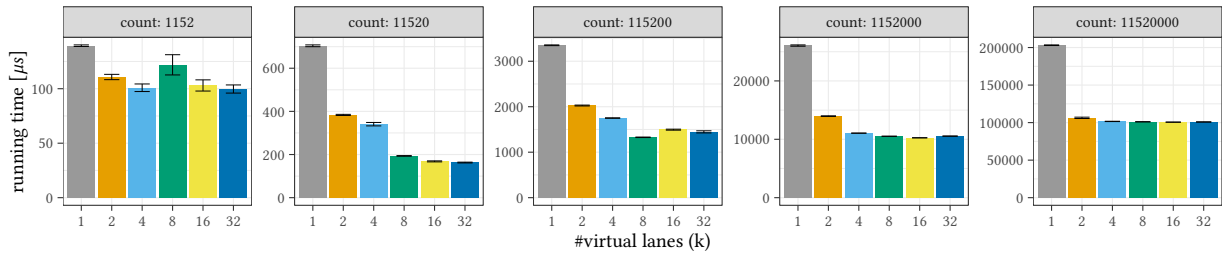


Fig. 1: Lane pattern benchmark results on *Hydra* with $N \times n = 36 \times 32$ processes for an increasing number of virtual lanes k. A count of c `MPI_INT` elements is sent and received by each node as c/k elements for the first k processes of the node. The MPI library used is Open MPI 4.0.2.

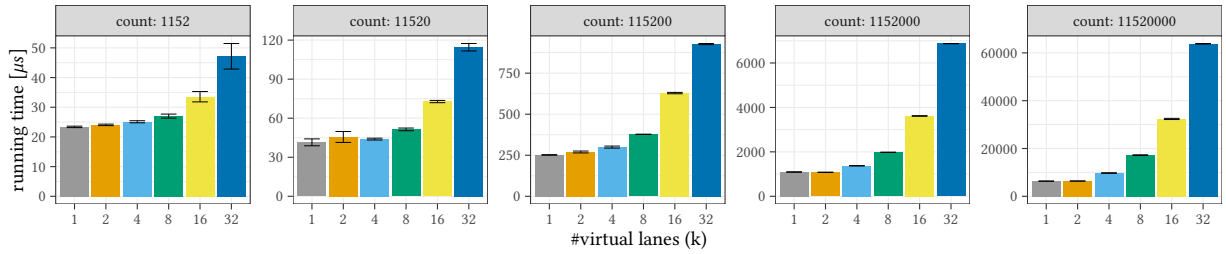


Fig. 2: Multi-collective pattern benchmark results on *Hydra* with $N \times n = 36 \times 32$ processes for an increasing number of virtual lanes k. For the k first processes on each node, collective calls are performed concurrently with a total count of c `MPI_INT` elements per call (total data volume per lane c and per node kc). The collective function on each lane is the most communication intensive `MPI_Alltoall` operation. The MPI library used is Open MPI 4.0.2.

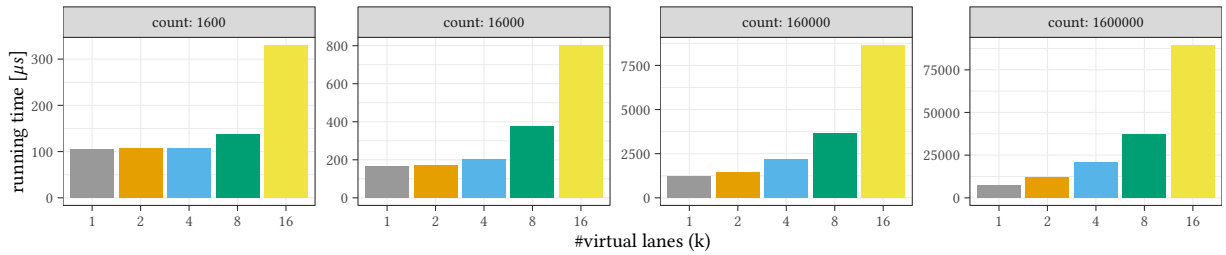


Fig. 3: Multi-collective pattern benchmark results on the *VSC-3* for an increasing number of virtual lanes k used for communicating the data (count c `MPI_INT`) per lane. The collective function is `MPI_Alltoall`. We used Intel MPI 2018 and $N \times n = 100 \times 16$ processes.

Our implementations assume *regular communicators* where all compute nodes host the same number of MPI processes, and the processes in the communicators are consecutively ranked over the nodes. Let `comm` be such a regular, consecutively ranked communicator. The MPI functionality `MPI_Comm_split_type` can be used to partition `comm` into disjoint *node communicators*, `nodecomm` (we assume that the given MPI library to split communicators non-trivially). Using `MPI_Comm_split` (or `MPI_Comm_create`), it is likewise easy to partition `comm` into as many disjoint *lane communicators* `lanecomm` as there are processes per node in `comm`. With this decomposition, shown in Figure 4, each MPI process in `comm` belongs to one `nodecomm` through which it can communicate with all other processes on the same node, and one `lanecomm` through which it can communicate with one process on each of the other nodes. In the code that follows `n<noderank`, `nodesize` ($= n$), and `lanerank`, `lanesize` ($= N$) will be the ranks and the number of processes in the `nodecomm` and `lanecomm` communicators, respectively. Note that we can check with a few allreduce operations whether `comm` is actually regular; if not, we let `lanecomm` be a duplicate of `comm` and `nodecomm` just a self-communicator with one process. This way, our implementations will work on any communicator `comm`.

The key idea of all the mock-up implementations is to divide the data evenly over the virtual lanes using a suitable collective operation on the node(s), perform collective operations concurrently over the lanes, and finally put the pieces together using again a suitable collective on the nodes. The decompositions are very similar to performance guidelines

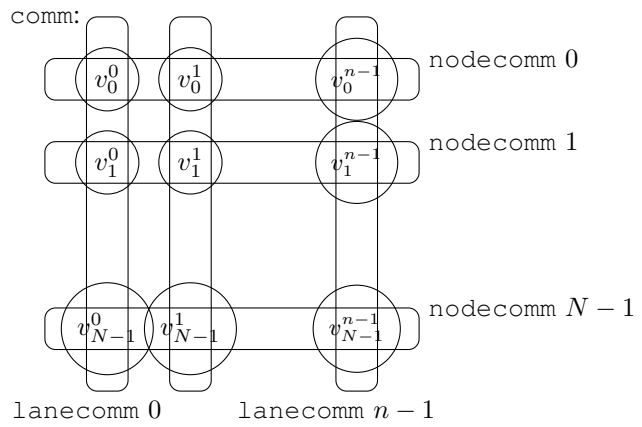


Fig. 4: The node and lane communicator decomposition of `comm` into disjoint `nodecomm` and `lanecomm` communicators as used in the full-lane collectives. Each MPI process v_j^i belongs to one of each such communicator, and has rank i in its `nodecomm` and rank j in its `lanecomm`.

Listing 1: The full-lane broadcast guideline implementation.

```
int Bcast_lane(void *buffer,int count,MPI_Datatype datatype,
               int root, MPI_Comm comm) {
  rootnode = root/nodesize;
  noderoot = root%nodesize;

  block = count/nodesize;
  for (i=0; i<nodesize; i++) counts[i] = block;
  counts[nodesize-1] += count%nodesize;
  displs[0] = 0;
  for (i=1; i<nodesize; i++)
    displs[i] = displs[i-1]+counts[i-1];
  blockcount = counts[noderank];

  if (lanerank==rootnode) {
    void *recbuf = (noderank==noderoot) ?
      MPI_IN_PLACE : (char*)buffer+noderank*block*extent;
    MPI_Scatterv(buffer,counts,displs,datatype,
                 recbuf,blockcount,datatype,noderoot,
                 nodecomm);
  }
  MPI_Bcast((char*)buffer+noderank*block*extent,
            blockcount,datatype,rootnode,lanecomm);
  MPI_Allgatherv(MPI_IN_PLACE,blockcount,datatype,
                 buffer,counts,displs,datatype,nodecomm);
  return err_code; // MPI_SUCCESS
}
```

often stated for collective operations [15], [16]. We use MPI user-defined datatypes to reorder and avoid explicit copying of data between intermediate buffers.

In the following, we show the concrete decompositions for some representative collectives, namely `MPI_Bcast`, `MPI_Allgather`, `MPI_Allreduce`, and `MPI_Scan`, including some of the complementary, hierarchical decompositions. The full code for all regular collectives is available for use.[1]

A. Broadcast

The full-lane broadcast implementation shown in Listing 1 (with obvious declarations left out) first divides the data c to be broadcast from the root process evenly over the processes on the compute node hosting the root by an `MPI_Scatterv` operation. Each process on the root node, now responsible for c/n data elements, broadcasts its data on its lane communicator. Finally, all processes perform an `MPI_Allgatherv` operation on the node communicator to assemble the full c data elements. The mock-up is thus similar to the `MPI_Scatter` followed by `MPI_Allgather` performance guideline for `MPI_Bcast`, just with an `MPI_Bcast` on proportionally smaller data sandwiched inbetween. With regular communicators, the node hosting the broadcast root r is $\lfloor r/n \rfloor$, and the node rank of the root is $r \bmod n$. The irregular `MPI_Scatterv` and `MPI_Allgatherv` operations are used to cater for the case where c is not divisible by n. If n divides c, regular (non-vector) `MPI_Scatter` and `MPI_Allgather` collectives can be used instead and might perform better.

Listing 2: Hierarchical broadcast guideline implementation.

```
int Bcast_hier(void *buffer,int count,MPI_Datatype datatype,
               int root, MPI_Comm comm) {
  rootnode = root/nodesize;
  noderoot = root%nodesize;

  if (noderank==noderoot) {
    MPI_Bcast(buffer,count,datatype,rootnode,lanecomm);
  }
  MPI_Bcast(buffer,count,datatype,noderoot,nodecomm);
  return err_code; // MPI_SUCCESS
}
```

The best possible performance of this `MPI_Bcast` mock-up can be estimated as follows (cf. [8], [20]): An optimal `MPI_Scatter` algorithm on `nodecomm`, assuming fully connected, bidirectional send-receive communication, takes $\lceil \log n \rceil$ communication rounds, and communicates $\frac{n-1}{n}c$ data. A broadcast of the c/n data on a `lanecomm` takes another $\lceil \log N \rceil$ communication rounds (again, assuming fully connected communication), and the c/n data are sent once. The final `MPI_Allgather` in the best case takes $\lceil \log n \rceil$ communication rounds, and sends and receives $\frac{n-1}{n}c$ data elements. Thus the total number of communication rounds is $2\lceil \log n \rceil + \lceil \log N \rceil \leq \lceil \log p \rceil + 1 + \lceil \log n \rceil$, which is $1 + \lceil \log n \rceil$ rounds more than optimal. The total volume of data sent or received by a process is $2\frac{n-1}{n}c + c/n = 2c - c/n$. The latter is almost a factor of two off from what an optimal broadcast algorithm could do. However, the total amount of data broadcast from or into a node is $n(c/n) = c$, i.e., the c data elements are sent from the broadcast root node once in chunks over the n lane communicators. With k physical lanes, the n concurrent broadcast operations on the n lane communicators could be sped up by a factor of k (as seen with the lane pattern benchmark). The scatter and allgather operations on the node communicators thus turn out a bottleneck with increasing n. Worth noticing is also that our implementations all make heavy use of `MPI_IN_PLACE`. In the analysis, we assume that a block of data of size c/n, which does not have to be scattered and allgathered, is actually not copied.

A complementary, hierarchical decomposition of the broadcast operations is shown in Listing 2. First, the root process broadcasts the data to all other nodes using its `lanecomm` after

[1] https://github.com/parlab-tuwien/mpi-lane-collectives

Listing 3: The full-lane allgather guideline implementation.

```
int Allgather_lane(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  MPI_Comm comm) {
  MPI_Type_contiguous(recvcount,recvtype,&lt);
  MPI_Type_create_resized(lt,0,nodesize*recvcount*extent,
                          &lanetype);
  MPI_Type_commit(&lanetype);

  MPI_Type_vector(lanesize,recvcount,nodesize*recvcount,
                  recvtype,&nt);
  MPI_Type_create_resized(nt,0,recvcount*extent,&nodetype);
  MPI_Type_commit(&nodetype);

  MPI_Allgather(sendbuf,sendcount,sendtype,
                (char*)recvbuf+noderank*recvcount*extent,
                1,lanetype, lanecomm);
  MPI_Allgather(MPI_IN_PLACE,sendcount,sendtype,
                recvbuf,1,nodetype,nodecomm);
  return err_code; // MPI_SUCCESS
}
```

Listing 4: Hierarchical allgather guideline implementation.

```
int Allgather_hier(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  MPI_Comm comm) {
  if (sendbuf==MPI_IN_PLACE&&noderank!=0) {
    takebuf   = (char*)recvbuf+rank*recvcount*extent;
    takecount = recvcount; taketype = recvtype;
  } else {
    takebuf = sendbuf; takecount = sendcount;
    taketype = sendtype;
  }
  MPI_Gather(takebuf,takecount,taketype,
    (char*)recvbuf+lanerank*nodesize*recvcount*extent,
    recvcount,recvtype,0,nodecomm);

  if (noderank==0) {
    MPI_Allgather(MPI_IN_PLACE,nodesize*recvcount,recvtype,
                  recvbuf,nodesize*recvcount,recvtype,
                  lanecomm);
  }

  MPI_Bcast(recvbuf,size*recvcount,recvtype,0,nodecomm);
  return err_code; // MPI_SUCCESS
}
```

which a broadcast is performed on the nodes from the local node root $r \bmod n$. The total amount of data communicated from a node is in this implementation purely determined by the quality of the `MPI_Bcast` implementation. The number of communication rounds is in the best case one off from optimal, and there is no extra node internal communication overhead. Multi-lane capabilities can possibly be exploited by bundling the lanes in the single broadcast over the nodes.

B. Allgather

In the full-lane allgather implementation, all processes first perform an `MPI_Allgather` on their `lanecomm`, resulting in Nc elements gathered per process. Then all processes on each node perform an `MPI_Allgather` over their `nodecomm`, resulting in $nNc = pc$ data elements gathered per process. With best possible implementations of the component allgather operations, the number of communication rounds is at most $\lceil \log p \rceil + 1$ (at most one round off from optimal [8]), and the number of data elements sent and received by each process is exactly $(N-1)c + (n-1)Nc = (p-1)c$, which is optimal (all data, except the process' own block are sent and received once). The total amount of data communicated from and to a node is $n(N-1)c = (p-n)c$, thus with k physical lanes, a speed up of a factor of k is possible for the simultaneous `MPI_Allgather` on `lanecomm`. Unfortunately, again the `MPI_Allgather` on `nodecomm` sends and receives $(n-1)Nc$ data elements, which prevents k fold speed-up with increasing n.

The implementation shown in Listing 3 is completely zero-copy, meaning no explicit data movements and also no intermediate buffer space are used. This is possible by the regularity assumption for the communicator `comm`, by which the blocks to be gathered on the lane communicators are spaced nc elements apart in the final receive buffer. Such strided data block layouts can be expressed with MPI derived vector data types, taking care to set the datatype extents correctly such that the `MPI_Allgather` operations can tile the received data blocks.

The complementary, hierarchical decomposition of the allgather operation is shown in Listing 4. This takes two node local collective operations, namely an initial gather and a final broadcast, inbetween which the allgather operation over the nodes on `lanecomm` 0 is performed. As was the case with the full-lane broadcast, this implementation cannot be optimal when compared to the round and volume lower bounds for fully connected systems [8] because of the two collective operations on the nodes.

C. Reduction

The full-lane reduction implementations rely on the observation that reduction can be performed as a reduce-scatter followed by an (all)gather operation (yet another performance guideline). The full-lane `MPI_Allreduce` first performs an `MPI_Reduce_scatter` on the `nodecomm` communicator followed by an `MPI_Allreduce` on the `lanecomm` of c/n data elements, finally followed by an `MPI_Allgatherv` to get the final result together. A mock-up implementation for `MPI_Allreduce` is shown in Listing 5. As was the case for the broadcast mock-up, the irregular `MPI_Reduce_scatter` and `MPI_Allgatherv` operations can be replaced by their regular counterparts `MPI_Reduce_scatter_block` and `MPI_Allgather` when c is divisible by n which might perform better.

Under best-case assumptions, the implementation takes at most $2(\lceil \log p \rceil + 1)$ communication rounds, with $2\frac{p-1}{p}c$ data elements being exchanged ($\frac{n-1}{n}c$ elements for `MPI_Reduce_scatter_block` and `MPI_Allgather` on `nodecomm`, and $2\frac{N-1}{N}c/n$ for `MPI_Allreduce` on `lanecomm`). This is the same as the best known allreduce algorithms. The bottleneck for achieving a k fold speed-up is again the collective operations on `nodecomm`.

For `MPI_Reduce`, the `MPI_Allreduce` on `lanecomm` is replaced by an `MPI_Reduce` operation, and the final `MPI_Allgatherv` by an `MPI_Gatherv` operation. This

Listing 5: The full-lane allreduce guideline implementation.

```
int Allreduce_lane(void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm) {
  block = count/nodesize;

  for (i=0; i<nodesize; i++) counts[i] = block;
  counts[nodesize-1] += count%nodesize;
  displs[0] = 0;
  for (i=1; i<nodesize; i++)
    displs[i] = displs[i-1]+counts[i-1];

  MPI_Reduce_scatter(sendbuf,
                     (char*)recvbuf+noderank*block*extent,
                     counts,datatype,op,nodecomm);
  MPI_Allreduce(MPI_IN_PLACE,
                (char*)recvbuf+noderank*block*extent,
                counts[noderank],datatype,op,lanecomm);
  MPI_Allgatherv(MPI_IN_PLACE,counts[noderank],datatype,
                 recvbuf,counts,displs,datatype,nodecomm);
  return err_code; // MPI_SUCCESS
}
```

Listing 6: The full-lane scan guideline implementation.

```
int Scan_lane(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op,MPI_Comm comm) {
  block = count/nodesize;

  for (i=0; i<nodesize; i++) counts[i] = block;
  counts[nodesize-1] += count%nodesize;
  displs[0] = 0;
  for (i=1; i<nodesize; i++)
    displs[i] = displs[i-1]+counts[i-1];

  takebuf = (sendbuf==MPI_IN_PLACE) ? recvbuf : sendbuf;
  MPI_Reduce_scatter(takebuf,
                     (char*)tempbuf+noderank*block*extent,
                     counts,datatype,op,nodecomm);

  MPI_Exscan(MPI_IN_PLACE,
             (char*)tempbuf+noderank*block*extent,
             counts[noderank],datatype,op,lanecomm);

  MPI_Scan(sendbuf,recvbuf,count,datatype,op,nodecomm);
  if (lanerank>0) {
    MPI_Allgatherv(MPI_IN_PLACE,counts[noderank],datatype,
                   tempbuf,counts,displs,datatype,
                   nodecomm);

    MPI_Reduce_local(tempbuf,recvbuf,count,datatype,op);
  }
  return err_code; // MPI_SUCCESS
}
```

can be further improved by replacing the `MPI_Reduce_scatter` on the root node by a final `MPI_Gather` and local reductions on the root process at the root node. The full-lane `MPI_Reduce_scatter_block` implementation decomposes the operation into two `MPI_Reduce_scatter_block` operations on `nodecomm` and `lanecomm`, but requires process local reorderings of the input data. The concrete implementations are not shown here.

D. Scan

A full-lane `MPI_Scan` implementation is shown in Listing 6. Again, a node-local reduce-scatter operation is used to split and reduce the data into blocks of c/n elements. Exclusive scans, concurrently on all `lanecomm` communicators of n/c data elements, can be used with the result of a node local scan on the c input elements to compute the final scan result for each process. The overhead, compared to a best possible implementation on a fully connected, homogeneous system is an extra `MPI_Allgatherv` operation.

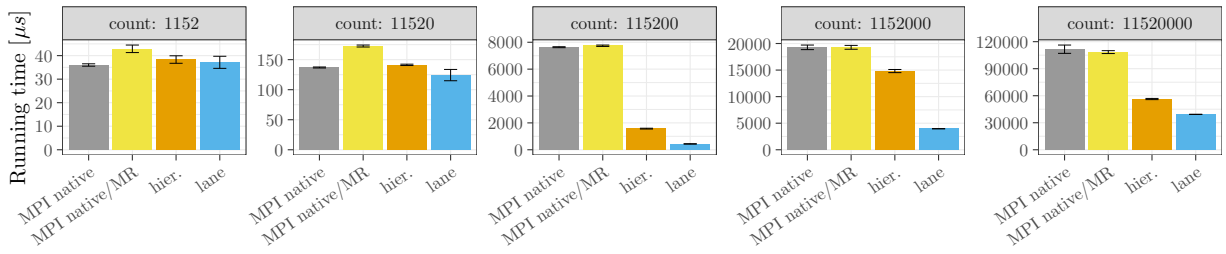
IV. Experimental Results

We have benchmarked the mock-up implementations of the collectives described in Section III, primarily on the *Hydra* system, see Table I. We have tested our approach with different MPI libraries (Intel MPI, MPICH, Open MPI, MVAPICH2), but primarily report for Open MPI 4.0.2. The findings with the other libraries are qualitatively comparable. On the *VSC-3*, we have used Intel MPI 2018. Our full-lane and hierarchical implementations were benchmarked as performance guidelines against the native MPI implementations of the corresponding collectives. This reveals differences between three different implementations of the same functionality, the native and closed, and the open multi-lane and hierarchical open mock-ups of Section III. Results do not show whether differences are due to different ways of exploiting the multi-lane capabilities of the system, but point to defects in the native implementations in cases where the mock-ups perform significantly better.

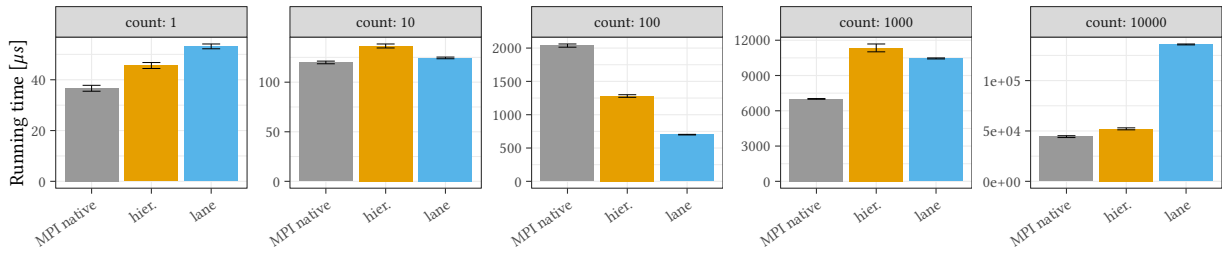
As we benchmark on two dual-rail systems, it is important to point out details of the experimental setup. All MPI libraries on *Hydra* use the PSM2 provider for the communication in the OmniPath interconnect. Similarly, the MPI libraries on *VSC-3* use the PSM provider. In a dual-rail setup, the mapping of MPI processes to cores is important. For example, if only two MPI processes are running on a compute node, they must be mapped to different sockets. Only then, they will communicate over different network cards. Hence, we always map MPI processes cyclically to the sockets on a compute node using the appropriate options in SLURM. For MVAPICH2, we set `MV2_CPU_BINDING_POLICY=scatter`, which achieves the same result. Notice that by using this configuration, the native MPI collectives can utilize the full potential of the dual-rail setup out-of-the-box. However, as we will see, restructuring the algorithms in a lane-based way improves their performance significantly.

A. Broadcast

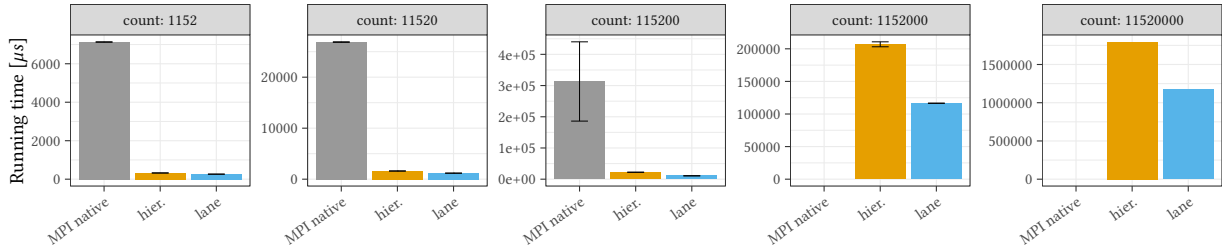
We have benchmarked with c divisible by n and N and ranging from 1152 to 11 520 000 `MPI_INT`s on *Hydra*. The results with the Open MPI 4.0.2 library are shown in Figure 5a. Even for the small $c = 1152$ count, the mock-up `Bcast_lane` implementations is better than the native `MPI_Bcast` operation, and as c grows, becomes so by a factor of about three. A particularly drastic result is for $c = 115\,200$ where the native `MPI_Bcast` is more than a factor of 20 off from the full-lane mock-up; this points to a severe defect in the MPI library broadcast implementation. Also the hierarchical broadcast of Listing 2 is better than the native `MPI_Bcast`, but consistently worse than the full-lane implementation. For comparison reasons, we also added the "multi-rail" results for the native MPI library, labeled with "MPI native/MR". In this case, we set `PSM2_MULTIRAIL=1`, which enables the communication of individual messages across multiple network



(a) Bcast variants.



(b) Allgather variants.



(c) Scan variants.

Fig. 5: Results for the native `MPI_Bcast`, `MPI_Allgather`, and `MPI_Scan` with $N \times n = 36 \times 32$ processes, compared against the mock-up guideline implementations on *Hydra*. The MPI library used is Open MPI 4.0.2. Note that for the large counts for `MPI_Scan`, the native implementation was so slow that the corresponding bar has been left out.

devices. It can be observed that enabling this option will only introduce an additional overhead to `MPI_Bcast`.

On the *VSC-3*, we have used $N = 100$ nodes and counts c ranging from 16 to 1 600 000 and which are divisible by the $n = 16$ processes per node. We can observe in Figure 6a that, starting from $c = 1600$, the mock-up performs better than the native `MPI_Bcast`, for $n = 160\,000$ by a large factor of more than seven, indicating again a problem with the broadcast implementation for the Intel MPI 2018 library.

B. Allgather

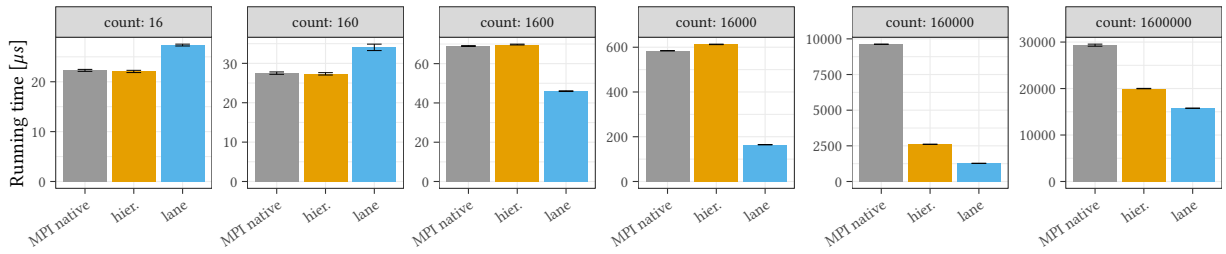
The results for the allgather implementations are shown in Figure 5b for *Hydra* and Figure 6b for the *VSC-3*.

For element block counts $c = 100$ (meaning that a total of $pc = 115\,200$ elements per process are gathered), the full-lane mock-up performs better than the native `MPI_Allgather` by a considerable factor of more than three, and also considerably better than the hierarchical implementation. As the block count increases, `MPI_Allgather` becomes (considerably) better than the mock-up, for $c = 10\,000$ by a factor of almost three. The hierarchical implementation of Listing 4 performs better than the full-lane implementations for the large block count, despite having a higher node-local overhead of two collective operations (gather and broadcast). The increased performance of the multi-lane allgather (cf. Listing 3) stems from an increased running time of the node-local allgather call, which uses a derived datatype. We therefore measured the running time of allgather on the 32 cores of a compute node without derived datatypes, and surprisingly, this call was by a factor of three faster than the allgather call that uses a derived datatype [21], which explains the performance loss for large counts.

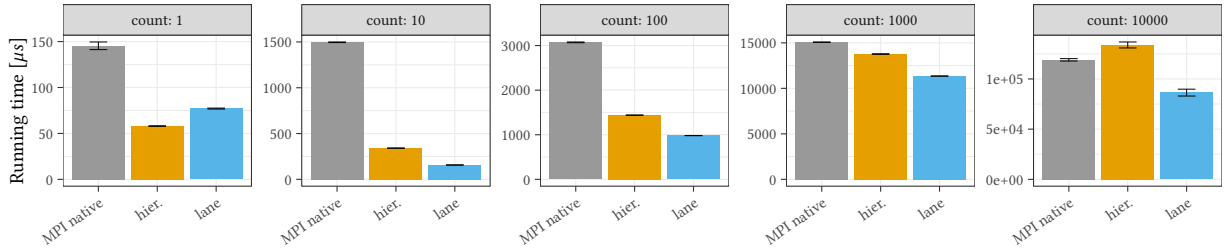
For the *VSC-3* the mock-up is in all cases better than the `MPI_Allgather` operation of Intel MPI 2018, for $c = 100$ by a factor of almost three, see Figure 6b. This could be an indication that the native Intel MPI 2018 library does not use a multi-lane algorithm, whereas the mock-up implementation can exploit multiple lanes as suggested by the multi-lane benchmark of Section II.

C. Scan

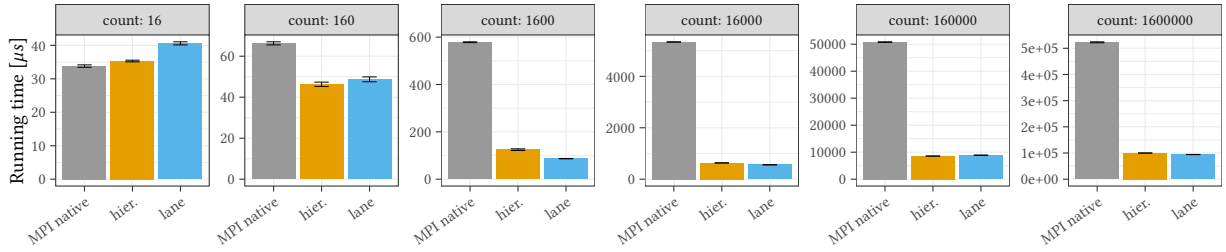
The results for the full-lane and hierarchical `MPI_Scan` implementations shown in Figure 5c and Figure 6c have been



(a) Bcast variants.



(b) Allgather variants.



(c) Scan variants.

Fig. 6: Results for the native `MPI_Bcast`, `MPI_Allgather`, and `MPI_Scan` with $N \times n = 100 \times 16$ processes, compared against the mock-up guideline implementations on the *VSC-3*. The MPI library used is Intel MPI 2018.

included to show particularly grave performance problems with this operation in the Open MPI 4.0.2 library (other libraries also have severe problems). For *Hydra*, not only is the completion time off from `MPI_Allreduce` by a factor of 50 or more, but also the two mock-up implementations are factors of 10 to 20 faster than the native `MPI_Scan`. Also for the *VSC-3*, scan time compared to allreduce is much too high, and the mock-ups perform by a factor of three and more better than the native `MPI_Scan` of Intel MPI 2018.

D. Allreduce

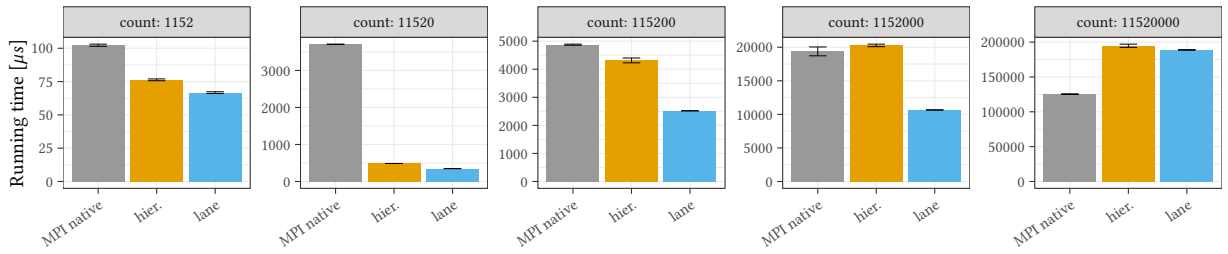
Results for the `MPI_Allreduce` collective for the *Hydra* system with the four MPI libraries Open MPI 4.0.2, MVAPICH2 2.3.3, MPICH 3.3.2, and Intel MPI 2019.4.243 are shown in Figure 7. The first observation is that the four libraries perform very differently, both quantitatively and qualitatively. The library behaving most closely to our expectations is MPICH 3.3.2, see Figure 7c, with our full-lane mock-ups being for all counts (roughly) a factor of two faster than the library native implementation (which performs similarly to our hierarchical implementation). For MVAPICH2 2.3.3, native and full-lane implementations are on par for counts $c = 11520$ and $c = 1152000$, in the other cases, our full-lane implementation is (about) a factor two better. The Open MPI 4.0.2 library has a severe `MPI_Allreduce` performance problem for $c = 11520$, and for unexplained reasons, our full-lane and hierarchical implementations are worse for the extremely large count $c = 1152000$. For the Intel MPI 2019.4.243 library, the full-lane implementation performs to expectation for the medium to large counts, being a factor of not quite 2 better than the native `MPI_Allreduce`.

E. Evaluation Summary

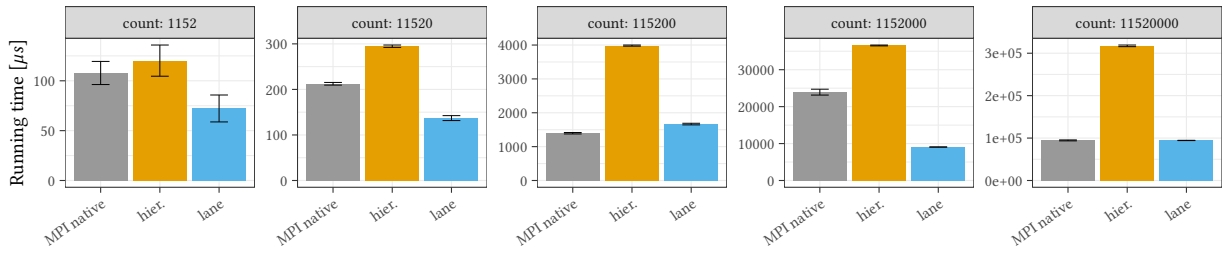
The comparison of the mock-up guideline implementations against the MPI library native collectives showed quite severe violations in many cases, often more than can be accounted for by failing multi-lane utilization. Other MPI libraries (MPICH, Intel MPI) on the *Hydra* system show similar (but quantitatively different) results.

V. Conclusion

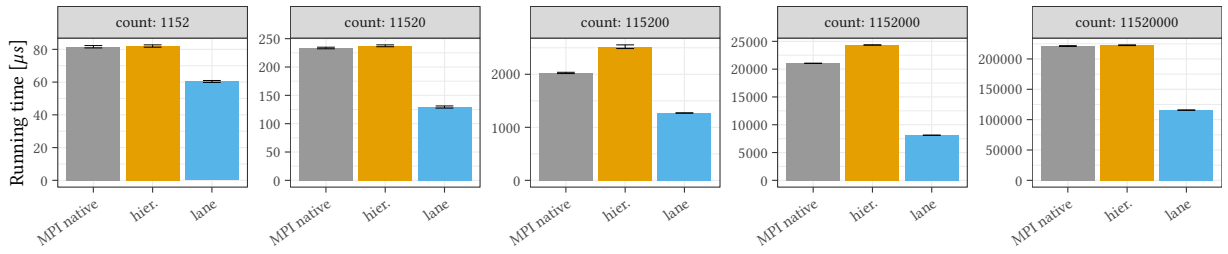
The two top ranked systems on the most recent TOP500 list (November 2019) both are dual-rail systems. We addressed the question of whether MPI libraries for such systems efficiently



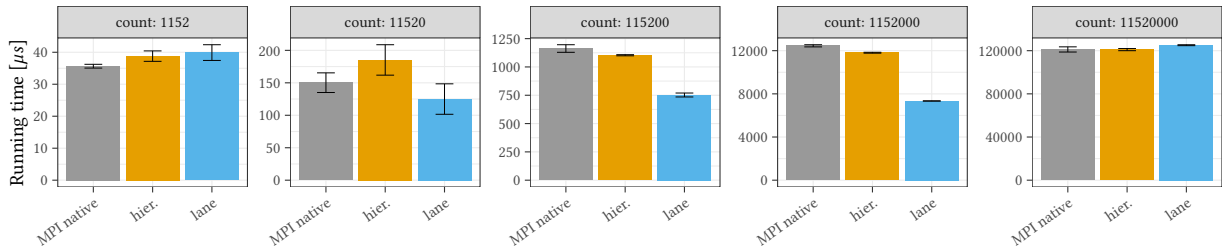
(a) Open MPI 4.0.2



(b) MVAPICH2 2.3.3



(c) MPICH 3.3.2



(d) Intel MPI 2019.4.243

Fig. 7: Comparison of the native `MPI_Allreduce` and the mock-up guideline implementations with $N \times n = 36 \times 32$ processes on *Hydra* with four different MPI libraries.

exploit multi-lane communication capabilities by proposing our full-lane implementations for the MPI collectives. It would be interesting to try out the proposed full-lane performance guidelines on TOP500 systems with a dual-rail setup.

On small(er) HPC systems, we demonstrated severe violations of the full-lane performance guidelines for the regular MPI collectives, indicating room for improvement that can possibly take better advantage of multi-lane capabilities. Our implementations assumed regular MPI communicators. It is an interesting question how collective algorithms and implementations can look for the cases where processes are not consecutively numbered and where compute nodes do not carry the same number of MPI processes. Likewise, we did not consider implementations for the irregular (vector) MPI collectives.

Theoretically (and practically) interesting questions are how to model realistically systems with k-lane capabilities, and how provably good algorithms will look in such a model. We note that a k-lane model where k processors on a node can communicate simultaneously with processors on other nodes (while possibly doing other things on the node at the same time) is different from the traditional k-ported model where every processor can communicate with k other processors at the same time. We will address these questions in future work.

REFERENCES

[1] MPI Forum, *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015, www.mpi-forum.org.

[2] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman, "Network performance-aware collective communication for clustered wide-area systems," *Parallel Computing*, vol. 27, no. 11, pp. 1431–1456, 2001.

[3] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, 2000, pp. 377–386.

[4] H. Ritzdorf and J. L. Träff, "Collective operations in NEC's high-performance MPI libraries," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, p. 100.

[5] R. Thakur, W. D. Gropp, and R. Rabenseifner, "Improving the performance of collective operations in MPICH," *International Journal on High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[6] J. L. Träff and A. Rougier, "MPI collectives and datatypes for hierarchical all-to-all communication," in *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA)*. ACM, 2014, pp. 27–32.

[7] A. Bar-Noy and C.-T. Ho, "Broadcasting multiple messages in the multiport model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 500–508, 1999.

[8] 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.

[9] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 64:1–64:11.

[10] K. C. Kandalla, H. Subramoni, G. Santhanaraman, M. J. Koop, and D. K. Panda, "Designing multi-leader-based allgather algorithms for multi-core clusters," in *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009, pp. 1–8.

[11] R. Kumar, A. R. Mamidala, and D. K. Panda, "Scaling alltoall collective on multi-core systems," in *Proceedings of the 8th Workshop on Communication Architectures for Clusters (CAC) at 22nd International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–8.

[12] A. Ruhela, B. Ramesh, S. Chakraborty, H. Subramoni, J. Hashmi, and D. Panda, "Leveraging network-level parallelism with multiple process-endpoints for mpi broadcast," in *Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, Nov. 2019.

[13] M. Kühnemann, T. Rauber, and G. Rünger, "Optimizing MPI collective communication by orthogonal structures," *Cluster Computing*, vol. 9, no. 3, pp. 257–279, 2006.

[14] J. L. Träff and A. Rougier, "Zero-copy, hierarchical gather is not possible with MPI datatypes and collectives," in *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA)*. ACM, 2014, pp. 39–44.

[15] S. Hunold, A. Carpen-Amarie, F. D. Lübbe, and J. L. Träff, "Automatic verification of self-consistent MPI performance guidelines," in *Proceedings of the Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 9833, 2016, pp. 433–446.

[16] J. L. Träff, W. D. Gropp, and R. Thakur, "Self-consistent MPI performance guidelines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 698–709, 2010.

[17] S. Hunold and A. Carpen-Amarie, "Tuning MPI collectives by verifying performance guidelines," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*, 2018, pp. 64–74.

[18] T. Hoefler and S. Gottlieb, "Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes," in *Proceedings of the 17th European MPI Users' Group Meeting (EuroMPI)*, ser. Lecture Notes in Computer Science, vol. 6305. Springer, 2010, pp. 132–141.

[19] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[20] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[21] A. Carpen-Amarie, S. Hunold, and J. L. Träff, "On expected and observed communication performance with MPI derived datatypes," *Parallel Computing*, vol. 69, pp. 98–117, 2017. [Online]. Available: https://doi.org/10.1016/j.parco.2017.08.006