MPI Collectives and Datatypes for Hierarchical All-to-all Communication*

Jesper Larsson Träff
traff@par.tuwien.ac.at

Antoine Rougier
rougier@par.tuwien.ac.at

Vienna University of Technology (TU Wien)
Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16/184-5
1040 Vienna, Austria

ABSTRACT

With recent MPI 3.0 functionality for creating communicators that partly reflects the hierarchy of standard clusters of shared-memory nodes, hierarchical, collective algorithms can more conveniently be implemented by combinations of other collective MPI operations. On systems that support MPI 3.0, with `MPI_Alltoall` as a concrete example, we show that performance on par with or better than native MPI implementations is possible, thus illustrating that the provided hierarchy awareness can be an effective and portable means for applications to implement their own, efficient (non-MPI) collective operations. Parts of the `MPI_Alltoall` implementation relies on MPI derived datatypes; however, the MPI collective interfaces lack expressivity to take full advantage of the performance benefits offered by the derived datatype mechanism.

1. INTRODUCTION

One of the most important features of MPI is the support for library building, and for implementing library functionality efficiently so as to be able to exploit in a portable way and to a reasonably high degree, some of the specific properties of standard, parallel systems. In this paper, we investigate the MPI support for building efficient, hierarchical collective operations out of the building blocks provided by MPI, in particular the communicator manipulation functionalities, the collective operations, and the derived datatype mechanism. As a concrete example, we consider a hierarchical implementation of regular all-to-all communication (`MPI_Alltoall`) implemented solely using MPI (3.0) functionality.

Parallel systems and communication architectures typically have a hierarchical structure, e.g., by consisting of shared-memory nodes (which often themselves have a hierarchical memory system) with an interconnect that may or may not (e.g., tori) be hierarchical. Hierarchical collective communication takes this into account, typically by algorithms that do collective communication level by level in the hierarchy. Good MPI library implementations can be expected to implement their collective operations with hierarchy sensitive algorithms, and many have actually invested significantly in well-performing collectives of this kind [6, 8, 9].

This paper investigates whether the facilities provided by MPI suffice for the application specific library developer to implement efficient, hierarchy sensitive communication algorithms, and consider all-to-all communication as a test-case for this: if `MPI_Alltoall` can be implemented well, also in terms of performance when compared to the native MPI library implementation, it is likely that other collective patterns needed for the application can likewise be implemented well. As we will see, some of the new MPI 3.0 functionality is of crucial importance (and was for exactly this reason considered by the MPI Forum and included in MPI 3.0 [7]). However, MPI does lack in functionality, which makes correct and type safe (in the MPI sense) implementations tedious and less efficient. The discussion of these issues is the main contribution of this paper.

2. HIERARCHICAL ALL-TO-ALL COMMUNICATION

A paradigmatic, hierarchy-sensitive implementation of the `MPI_Alltoall` collective operation is shown in Figure 1. On a shared-memory node cluster, the algorithm gathers locally for each node all data elements to be exchanged with processes on other nodes, performs an all-to-all operation across the nodes, and scatters the received elements locally on all nodes. Depending on problem size, different algorithms for the gather, scatter, and all-to-all steps may have to be used, and pipelining may have to be employed at various stages. A hierarchical implementation can better utilize the non-homogeneous communication system, especially for small problems where communication latency is a factor. Many current MPI libraries implement `MPI_Alltoall` in this fashion, see, e.g., [6, 8, 9]. We make no claim that this is the theoretically best way to implement all-to-all communication on hierarchical systems.

*This work was co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598), and supported by the Austrian FWF project "Verifying self-consistent MPI performance guidelines" (P25530).

EuroMPI/ASIA '14, September 09-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642770.

```
int MPI_Alltoall(sendbuf,...,recvbuf,...,comm) {
  // split comm into local and bridge communicators
  // for this and next level
  MPI_Comm_rank(local,&localrank);
  // choose root in this level communicator
  // allocate send and receive buffers for this level
  MPI_Gather(sendbuf,...,sendlocal,...,localroot,local);
  if (localrank==localroot)
    MPI_Alltoall(sendlocal,...,recvlocal,...,bridge);
  MPI_Scatter(recvlocal,...,recvbuf,...,localroot,this);
}
```

Figure 1: Paradigm hierarchical `MPI_Alltoall` implementation. The gather, scatter and all-to-all calls implement good algorithms for local and bridge communicators, which could themselves be hierarchical if the application communicator spans more than two hierarchy levels.

We use p for the total number of MPI processes, N for the number of nodes (size of `bridge` communicator), and $n = p/N$ for the average number of processes per node (size of `local` communicator). The size of each data element to another process is m (for regular problems). Data elements can be arbitrarily structured as determined by the MPI type and count arguments. The *total problem size* is pm, since this is the volume of data that has to be sent and received per process (including an element to the process itself), regardless of element structure.

Here, we concentrate on small to medium problem sizes, where it indeed makes sense to gather all data locally on the nodes, and invoke a given, closed, all-to-all algorithm for communication across the nodes. We go through the steps required to fill the missing details in Figure 1.

2.1 Hierarchical communicator splitting

The first problem is to determine whether there is a natural splitting of the argument communicator into disjoint, shared-memory communicators. Prior to MPI 3.0 this was tedious, and hardly possible in a portable way (a possibility was using `MPI_Get_processor_name`). MPI 3.0 [7] exposes more of the communication hierarchy by a new communicator splitting operation, `MPI_Comm_split_type`, together with significantly extended one-sided communication functionality [4]. The `MPI_Comm_split_type` operation has a predefined type argument value that allows building subcommunicators consisting of the processes in the calling communicator that reside on the same shared-memory node. The code in Figure 2 accomplishes the two-level split required in Figure 1.

Currently, there are no further predefined split types for systems with more than two, natural hierarchy levels, e.g, for the cache- or memory-hierarchy for NUMA-nodes. Note that a trivial MPI 3.0 implementation may return a communicator equivalent to `MPI_COMM_SELF` for `this`.

Since MPI processes can be arbitrarily mapped to physical processors, the MPI ranks of the application communicator `comm` need not be consecutive on each shared memory node, and there is in general no method, bar maintaining a map, for an MPI process to determine which ranks belong to a particular node. Since the semantics of the MPI all-to-all operation is defined in terms of rank-order, at least the local root processes need to know which ranks reside on which

```
MPI_Comm_split_type(comm,
                MPI_COMM_TYPE_SHARED,0,MPI_INFO_NULL,
                &local);
MPI_Comm_rank(local,&localrank);
localroot = 0; // choose local root
MPI_Comm_split(comm,
            (localrank==localroot) ? 0 : MPI_UNDEFINED,0,
            &bridge);
```

Figure 2: Splitting argument communicator into node local communicators, and a bridge communicator for the communication between nodes.

nodes. An array storing the global `comm` ranks sorted in node order as determined by the ranks of the local root processes in the `bridge` communicator is needed. This can easily be computed by an (irregular) allgather operation over all local roots, each contributing the ranks in `comm` on its node. These arrays can be computed either by the local root with an `MPI_Group_translate_ranks` operation, or by a node-local gather.

In order to avoid computing this information at each (all-to-all) call, a library initialization function should precompute both hierarchy communicators and node-sorted global rank array, and cache this information as attribute to the library communicator. An attractive, and possibly more space efficient representation of the latter could be as an MPI derived (indexed-block) datatype; this datatype, representing a permutation of the global ranks, is needed anyway as explained in the next section. Unfortunately, this datatype would be specific for send or receive types in the subsequent `MPI_Alltoall` calls, and in the absence of persistent collective operations [13], precomputation is not possible.

2.2 Regular all-to-all on Regular communicators

By a *regular cluster*, respectively *regular communicator* in a hierarchical system, we mean a cluster respectively communicator in which each shared-memory node has the same number of processes. Since MPI allows generation of arbitrary communicators not all MPI communicators are regular, even if a cluster typically is. We first consider the regular all-to-all communication problem for regular communicators. In this case, each local root has to send n^2 elements of the same size m to each other node. Since the MPI ranks of the communicator may be arbitrarily permuted over the shared-memory nodes spanned by the communicator, we need to be careful how data from the non-root local processes are gathered at the node-local root when instantiating the paradigm of Figure 1.

If we view the all-to-all problem as a data redistribution problem, a hierarchical approach involves the steps outlined in Figure 3. Let $m_{I\to J}^{i\to j}$ denote the data element from local rank i on node I to local rank j on node J, where $0 \leq i, j < n$ and $0 \leq I, J < N$. In the first step, each local process contributes its elements to the other nodes, sorted in node-order, as shown by state (1) in Figure 3. This sorting can be implicitly accomplished by using a derived send datatype in the `MPI_Gather` operation which accesses the elements in node-sorted order. An indexed-block type will do this, using the node-sorted array of global ranks as index array. State (2) illustrates the order in which the all-to-all operation over

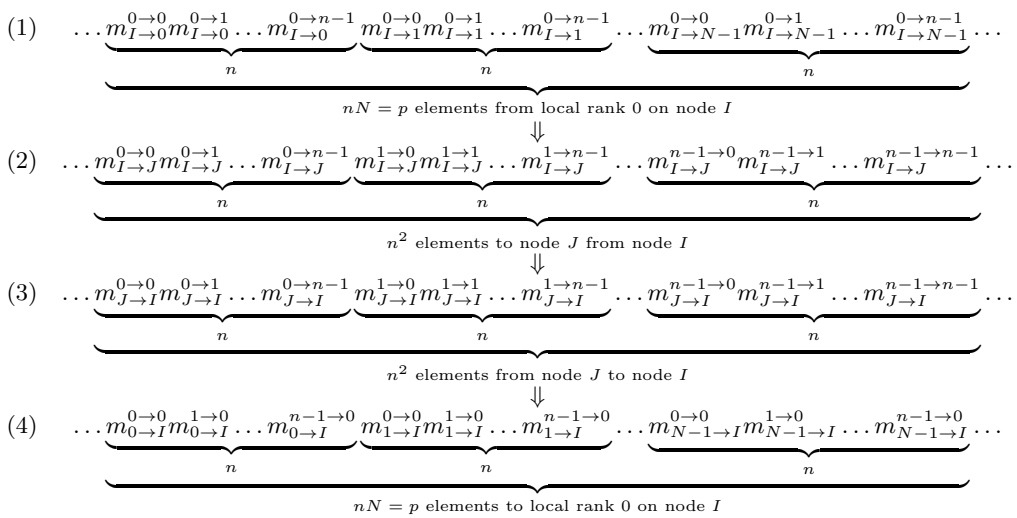

Figure 3: Steps involved in element redistribution in hierarchical all-to-all communication. The transition from state (1) to state (2) is accomplished by node-local gather; the transposition from state (2) to state (3) by all-to-all; and the transition from state (3) to state (4) by node-local scatter. The sorting into node order before state (1), and reordering after state (4) into global rank order as accomplished by index-block datatypes is not shown.

the nodes shall access the elements: these are grouped into blocks of n^2 elements for each node. The transposition from state (2) to state (3) is accomplished by the `MPI_Alltoall` operation on the `bridge` communicator. State (4) shows the order of received elements required for the redistribution over the node-local processes; the reorganization into global rank order is again done by the `MPI_Scatter` collective with an indexed-block derived datatype as process-local receive type.

For the concrete implementation the aim is to avoid explicit node local element reorganizations, and let all state transitions be accomplished by the three collective operations gather, all-to-all, and scatter. To achieve this, we rely on the derived datatype mechanism to capture the four states of Figure 3. Such an implementation is said to be *zero-copy*. We exemplify with the gather operation (scatter is similar), and present two solutions.

The **first alternative** is to let each node-local process contribute its elements in node-sorted order to the local root, which stores these contributions consecutively in local rank order, as shown explicitly as state (1). State (2) is implicit, and the ordering into node-order is done by the all-to-all operation which accesses the elements first by local destination rank $j, 0 \leq j < n$, then local source rank $i, 0 \leq i < n$, and then destination node J. A vector datatype with n blocks of n elements at a stride of p elements with an extent of n elements as sendtype for `MPI_Alltoall` will accomplish this.

The **second alternative** is to make the gather operation take care of receiving the elements from the local ranks grouped into node-order as required in state (2). To do this a vector datatype with N blocks of n elements with a stride of n^2 elements and an extent of n elements is used as receive type for the node-local gather operation.

For both alternatives an MPI vector constructor describes the required access order; resizing, an MPI technicality, must be applied to set the extent properly to achieve the desired tiling. Note, that this only works because all elements have the same size m, and because all nodes have the same number of local ranks n. The vector constructor is space efficient, and vectors can normally be processed efficiently by MPI libraries.

To save space (and time) it is desirable that the local root buffer needed for gather/scatter and all-to-all store data in a contiguous fashion. Since the data elements can be arbitrarily structured as determined by the `sendtype` arguments, it would be desirable to have a contiguous but typed description of the input and output data, for instance in the form of a signature type as advocated in [13]. In the absence of this (there is no such MPI functionality), we treat intermediate buffers as having `MPI_BYTE` type, although this is strictly incorrect, and loses type information.

The order of elements contributed per node-local process is determined by the node-sorted process map, and described by an MPI indexed-block datatype. Type normalization [2, 12] might be able to simplify such a type of p blocks into a more efficient representation. In the special case of consecutive process ranks on the nodes a contiguous type suffices; the implementation could easily do this case analysis when the hierarchical communicators and rank map is created.

2.3 Irregular communicators

When the user communicator is not regular, things get more complicated. First we note that the all-to-all problem to be solved over the nodes is no longer regular [10], therefore either `MPI_Alltoallv` or `MPI_Alltoallw` has to be used for the node-wise communication. Both interfaces are non-scalable [1], in particular the latter, and for both, algorithms and MPI library implementations are typically worse than for `MPI_Alltoall`. For now, the focus is on accomplishing the transitions of Figure 3.

The second solution which directly gathers the node-local elements into node-order cannot work because the blocks of

elements for each node have different number of elements, namely n_I^2 for node I, and are therefore not uniformly strided. An indexed type could be used for each node-local process, but the data semantics of `MPI_Gather` is vector-like and enforces the gathered blocks to be stored at evenly spaced indices, and would therefore not correctly interleave the elements in node order. The irregular `MPI_Gatherv` operation does not help; it allows different offsets for the blocks of elements from each node-local process, but what is required is that each block of elements from a node-local process be described by an own datatype. An `MPI_Gatherw` operation is not in MPI 3.0, and we do not recommend it.

The first solution, which leaves it to the all-to-all operation to pick the elements for each node, can on the other hand be made to work. The p elements from each node-local process are gathered consecutively, and the all-to-all operation sets up a datatype for each node that picks out the elements for that node: n_I blocks of n_J elements to node J with a stride of p elements. However, unlike the regular communicator case, the datatypes for each node J are different with a different blocksize (and therefore tiling cannot be employed: the block for node J is not a simple, constant shift of the datatype for node 0), which necessitates using the non-scalable `MPI_Alltoallw` operation in this case.

If node local space consumption is not an issue, a third solution is possible which makes it possible to use the potentially better, more scalable `MPI_Alltoallv` call. Allocate for each node-local process space for N blocks of $n = \max_I n_I$ elements (which is considerably more than p elements when the n_I's differ significantly), and gather locally into these buckets. For the `MPI_Alltoallv` call, a vector datatype with N_I one-element blocks at a stride of Nn and an extent of one element can then be used, such that a different count for each node will suffice. The all-to-all operation receives into a similar bucket layout, for which a node-local scatter sendtype can be defined. Due to the high space consumption, we have not implemented this solution.

For the first two solutions, a more flexible datatype and/or collective interface could solve the problem in a more scalable manner [11]. A way to express p indexed types in one operation is needed, where each type consists of n blocks of size $b[j], 0 \leq j < n$, and the jth block of type $i, 0 \leq i < p$ starts at displacement $ib[j] + \sum_{k=0}^{j-1} pb[k]$. We would call this type a *stretched vector*.

2.4 Irregular all-to-all

The scheme of Figure 1 can likewise be used for the irregular all-to-all operations but leads to the same problems as discussed above. Since no process has information on the amount of data that other processes will exchange, an additional node-local gather of send and receive counts is necessary for setting up the datatypes for the `MPI_Alltoallw` call. For a hierarchical implementation of `MPI_Alltoallw`, information on the datatypes used by the node-local processes is additionally needed; communicating datatype information is not supported by MPI, so additional datatype marshalling as discussed in, e.g., [5] would be required.

3. EXPERIMENTAL RESULTS

We implemented the hierarchical all-to-all algorithms as described in Section 2.2 (second solution) and Section 2.3. The measurements include all type creation and destruction overheads, but the communicator creation is performed and benchmarked separately. We compare to the native implementations of `MPI_Alltoall` and `MPI_Alltoallv` for our vendor MPI library (our current versions of `mvapich` and `OpenMPI` do not support `MPI_Comm_split_type` in a non-trivial fashion) on a small InfiniBand-based cluster. The cluster has $N = 36$ nodes with $n = 16$ cores per node composed of two 8-core 2.3GHz AMD 6134 Opteron processors/node, and interconnected with a Mellanox MT4036 QDR InfiniBand switch. The total number of cores is $p = 576$.

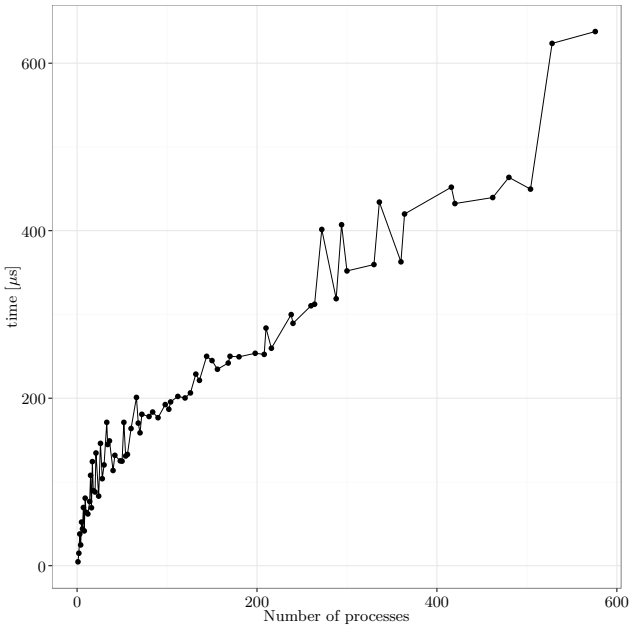

Figure 4: Communicator splitting for creation of `local` and `bridge` communicators, and node-sorted rank map.

3.1 Communicator creation overhead

We first time splitting of `MPI_COMM_WORLD` into node-local shared-memory communicators (`local`) and `bridge` communicator spanning the shared-memory nodes. As Figure 4 shows, time grows sublinearly with the number of MPI processes, except for a rather large jump at $p = 504$. For small all-to-all problems, the split times of $600\mu s$ would add significantly to the running time, therefore creation of the communicators needed for the hierarchical algorithms must be done separately, once and for all. In our implementations, an initialization call caches communicators and node sorted rank map with the user communicator.

3.2 Regular all-to-all

We first investigate whether hierarchical communication makes sense; our assumption is that common MPI libraries employ some form of hierarchical implementation for hierarchical systems (clusters). We compare, on the full cluster with $p = 576$ MPI processes, for a regular problem, the performance of (a) a strongly non-hierarchical implementation of all-to-all in terms of send-receive operations ($p-1$ communication rounds, where rank i receives from rank $(i-r)$ mod p and sends to rank $(i+r)$ mod p in round $r, 0 \leq r < p$, (b) `MPI_Alltoall`, (c) `MPI_Alltoallv`, (d) `MPI_Alltoallw`, and

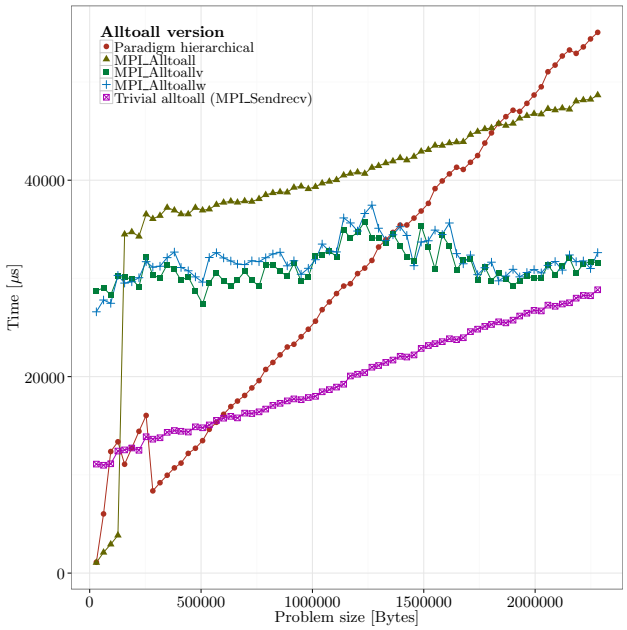

Figure 5: Regular all-to-all communication on regular communicator, five implementations: `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Alltoallw`, p-round send-receive, and our implementation. Communicator is `MPI_COMM_WORLD`.

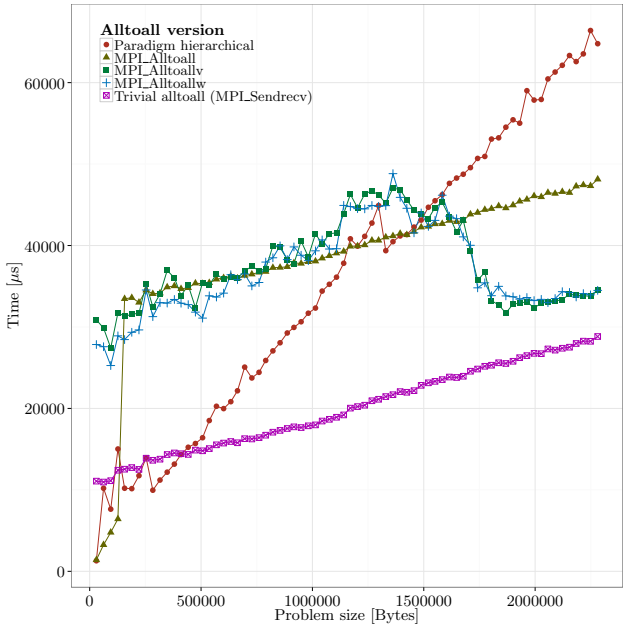

Figure 6: Regular all-to-all communication on regular communicator, five implementations: `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Alltoallw`, p-round send-receive, and our implementation. Communicator is a random permutation of `MPI_COMM_WORLD`.

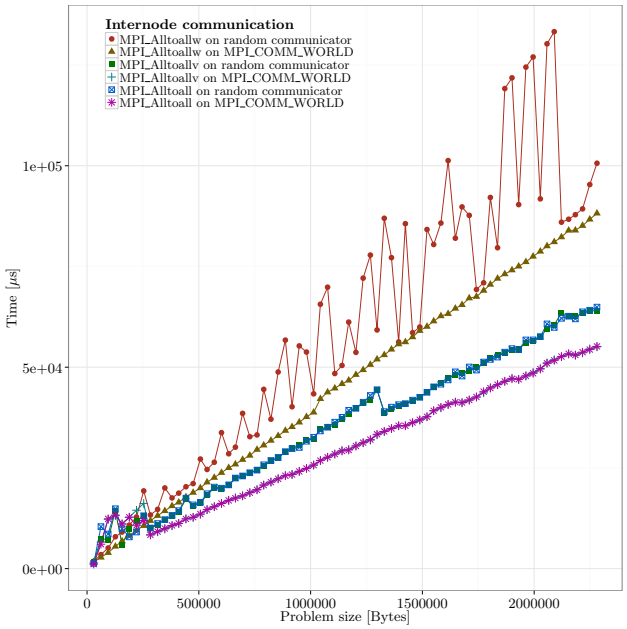

Figure 7: Hierarchical all-to-all implementation with either `MPI_Alltoall`, `MPI_Alltoallv` or `MPI_Alltoallw` for the internode communication.

(e) our implementation. Results, both for `MPI_COMM_WORLD` and a communicator where the ranks have been randomly permuted (using a fixed seed), are shown in Figure 5 and Figure 6. Plots show the mean running times over 100 iterations with outliers removed using Tukey's outlier filter, see, e.g., [3]. For all experiments the total problem size pm is in the range $[0, 2Mi]$ Bytes.

The results are revealing. The trivial send-receive implementation fares remarkably well, compared to the vendor implementations of `MPI_Alltoallv` and `MPI_Alltoallw`. These are obviously not hierarchically implemented. Hierarchical algorithms makes sense: our implementation is better than send-receive for problem sizes up to about 570KBytes. The vendor implementation has a very unfortunate early, implementation shift around 128KBytes, independent of the communicator.

In Figure 7 we investigate the overhead incurred by having to use `MPI_Alltoallw` for the communication between the shared memory nodes. A regular communicator is used, again $p = 576$, and we do this by replacing the `MPI_Alltoall` call in Algorithm 1 by an equivalent `MPI_Alltoallw` call. As can be seen, with the vendor MPI library, the price for using the non-scalable `MPI_Alltoallw` call is high. This operation should be avoided when possible, but as discussed in Section 2.3, for non-regular communicators, this is not possible with current MPI collectives and datatype support.

Finally, we solve regular problems on an irregular communicator, $N = 36$, but alternating $n = 1$ and $n = 16$ over the nodes, for a total of $p = 306$ MPI processes, both for `MPI_COMM_WORLD` and a randomly permuted communicator. We plot our implementation against the vendor MPI library. Results are shown in Figure 8, where the weak performance of the vendor MPI is very conspicuous. These experiment must all be repeated with other MPI libraries.

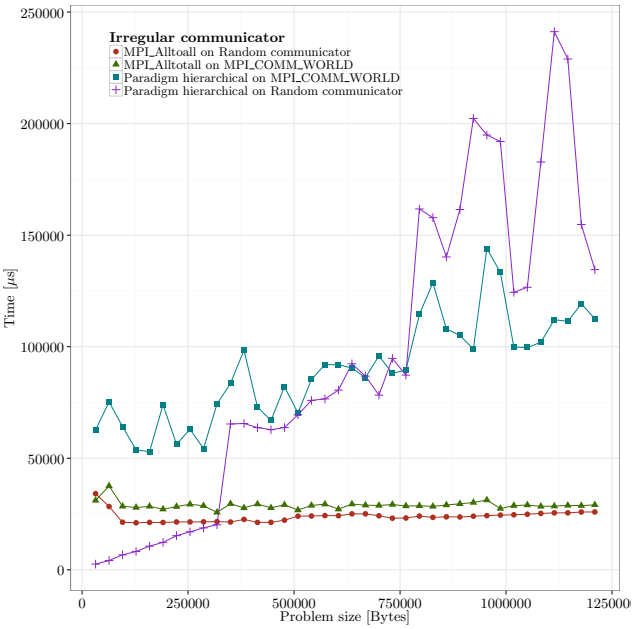

Figure 8: Regular all-to-all problem on irregular communicator, even numbered nodes with $n_I = 16$ processes, odd numbered nodes with $n_I = 1$ processes. Communicator is either `MPI_COMM_WORLD` or a random permutation. Our hierarchical implementation is compared against the vendor `MPI_Alltoall`.

4. CONCLUDING REMARKS

The purpose of this paper was to discuss the use of MPI datatype and collective operations facilities for implementing hierarchical, application-specific collective operations in a portable and efficient manner. For regular problems on regular communicators, we presented two scalable implementations that use only scalable datatype constructors (vector and resize) and the `MPI_Alltoall` operation. Only for the process-local ordering of data elements into node-sorted order a possibly space-consuming indexed-block datatype is needed. Unfortunately, this implementation does not carry over to irregular communicators. Here the fully general, but non-scalable `MPI_Alltoallw` operation is needed, although all data elements for all processes have the same type (signature), and an `MPI_Alltoallv` operation should have been sufficient. These observations might be stimulating for the design of more flexible collective and datatype interfaces [11]. We did not discuss (nor measure) the overhead in creating (and freeing) the datatypes, nor means for amortizing such overheads, for instance by the means for persistent collective interfaces; this and other uses of datatypes for the implementation of complex algorithms are discussed in [13]. Likewise, we only focused on an algorithm schema for small all-to-all problems; as problem size grows, other, possibly pipelined algorithm schemes are needed, and portable, user-level implementations of such schemes pose other challenges (that we may discuss in a follow-up paper).

Our experimental results were partly surprising, revealing some unfortunate implementation choices in the vendor MPI library that we used, but otherwise confirming both that hierarchy sensitive implementations are needed, be it for MPI collectives as defined in the standard, or for other application-specific operations, and that the MPI datatype functionality can be useful without incurring unacceptable overheads.

5. REFERENCES

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(1):45–60, 2011.

[2] W. D. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance expectations and guidelines for MPI derived datatypes: a first analysis. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2011.

[3] J. Hedderich and L. Sachs. *Angewandte Statistik*. Springer, 14 edition, 2012.

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

[5] D. Kimpe, D. Goodell, and R. B. Ross. MPI datatype marshalling: A case study in datatype equivalence. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 17th European PVM/MPI Users' Group Meeting*, volume 6305 of *Lecture Notes in Computer Science*, pages 82–91. Springer, 2010.

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

[7] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*, September 21st 2012. `www.mpi-forum.org`.

[8] H. Ritzdorf and J. L. Träff. Collective operations in NEC's high-performance MPI libraries. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, page 100, 2006.

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

[10] J. L. Träff. Relationships between regular and irregular collective communication operations on clustered multiprocessors. *Parallel Processing Letters*, 19(1):85–96, 2009.

[11] J. L. Träff. Alternative, uniformly expressive and more scalable interfaces for collective communication in MPI. *Parallel Computing*, 38(1–2):26–36, 2012.

[12] J. L. Träff. Optimal MPI datatype normalization for vector and index-block types. In *Recent Advances in Message Passing Interface. 21st European MPI Users' Group Meeting*, 2014.

[13] J. L. Träff, A. Rougier, and S. Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *28th ACM International Conference on Supercomputing (ICS)*, pages 135–144, 2014.