Using Node Information to Implement MPI Cartesian Topologies

William D. Gropp
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

ABSTRACT

The MPI API provides support for Cartesian process topologies, including the option to reorder the processes to achieve better communication performance. But MPI implementations rarely provide anything useful for the reorder option, typically ignoring it. One argument made is that modern interconnects are fast enough that applications are less sensitive to the exact layout of processes onto the system. However, intranode communication performance is much greater than internode communication performance. In this paper, we show a simple approach that takes into account only information about which MPI processes are on the same node to provide a fast and effective implementation of the MPI Cartesian topology. While not optimal, this approach provides a significant improvement over all tested MPI implementations and provides an implementation that may be used as the default in any MPI implementation of `MPI_Cart_create`.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**;

KEYWORDS

Message passing, MPI, Process topology, Cartesian process topology

ACM Reference Format:
William D. Gropp. 2018. Using Node Information to Implement MPI Cartesian Topologies. In *25th European MPI Users' Group Meeting (EuroMPI '18), September 23–26, 2018, Barcelona, Spain.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3236367.3236377

1 INTRODUCTION

In a parallel program, different processes exchange data with each other. A parallel computer provides an interconnection network that moves the data between the processes. For all but the smallest (in terms of number of processing elements) parallel computers, it is not possible to provide the same performance, either in terms of message latency or bandwidth, between all possible pairs of processes. Thus, determining the assignment of processes (and threads) to the hardware can impact communication performance and thus the performance of the parallel application.

The Message Passing Interface has included routines to aid the programmer in defining a *virtual process topology*, that is, an assignment of processes to processors, along with expected (or perhaps preferred) communications between them. In MPI-1, users could define either a *Cartesian* topology or a *Graph* topology. The Graph topology defines a general relationship between processes as a graph. The Cartesian topology defines a regular, n-dimensional mesh of processes, with the option of periodicity in any combination of dimensions. These virtual process topology routines create a new MPI communicator. Additional routines allow the user to find the rank, in this new communicator, of processes relative to the Cartesian process topology. For example, `MPI_Cart_shift` returns the rank of neighboring processes in the coordinate directions for an arbitrary shift; a shift of one gives the immediate neighbors.

In addition to their convenience, these routines provide the option to reorder the process ranks to better match the underlying physical interconnect and hardware. The value of this can be seen in Figure 1, which shows how the common assignment of MPI processes consecutively on a node can lead to significantly greater internode communication than a more optimized layout.

Despite these advantages and a wealth of research on how to implement both the Cartesian and Graph topologies effectively, the MPI process topologies are rarely used and even when used, rarely provide any performance benefit. For example, few benchmarks use any of the MPI process topology routines; Table 1 summarizes some popular benchmarks. For the only benchmark that does use `MPI_Cart_create`, SNAP, the communication demands are limited—the code uses only blocking send and receive, and uses `MPI_Cart_sub` rather than `MPI_Cart_shift` to find the neighboring processes.

This creates a "chicken and egg" situation—benchmarks and users do not make use of the MPI process topology routines, hence MPI implementations do not see a great demand for optimizing them, so users and benchmarks do not see a performance benefit in using them, so they don't use them. While some of these benchmarks would be clearer if they used the Cartesian process topologies rather than manually creating the process decomposition by hand, the lack of a performance benefit discourages the use of these routines.

One reason that implementations have rarely provided an implementation of `MPI_Cart_create` that takes advantage of the "reorder" option is the difficulty in creating an optimal mapping of the processes to processors. The major exceptions were the IBM MPI implementation for the Blue Gene/L and later Blue Gene systems; this system used a 3D mesh (5D for the Blue Gene/Q) with jobs allocated to a block of processes, and the NEC SX systems around 2002 [20]. Other systems, such as the Cray XE6, which use a mesh interconnect, may both include service nodes in the mesh (these do not run user programs) and, for reasons of utilization efficiency, may not allocate a user job to a single block of processes.

EuroMPI '18, September 23–26, 2018, Barcelona, Spain
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6492-8/18/09…$15.00
https://doi.org/10.1145/3236367.3236377

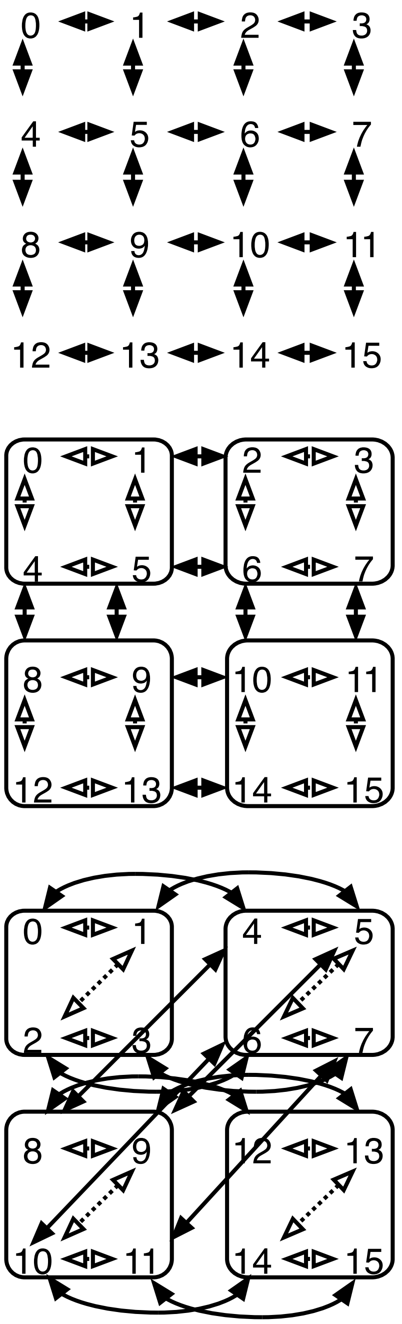

Figure 1: A sample mapping of processes to nodes. The top figure is a natural ordering of processes on a mesh, with arrows for the communication between neighbors. The middle image shows this with nodes containing four processes; the open arrows are for intranode communication. The bottom figure shows the communication in the typical case where MPI processes are assigned consecutively to a node, showing the greater number of internode communications.

Table 1: Summary of the use of `MPI_Cart_create` in several popular benchmarks that could make use of a Cartesian process topology. Note several CORAL benchmarks include code that calls `MPI_Cart_create`, but those are not used in the benchmark. LAMMPS provides an option to use reordering, but this does not appear to be used in any of the benchmarks.

Benchmark	Codes	Uses Cart	With reorder
NAS PB 3.3.1 [14]	All	No	NA
HOMB 1.0 [8]	–	No	NA
POP 2.0 [17]	–	No	NA
HPCC 1.4.1 [9, 11]	All	No	NA
CORAL [2]	HACC	Yes	No
	nekbone	No	NA
	QMCPACK	No	NA
	Snap	Yes	Yes
SPP-2017 [19]	PSDNS	Yes	No
	WRFV3	Yes	No
LAMMPS (16 Mar 2018)[10, 16]	–	Yes	Yes

Given the difficulty in mapping even a regular grid of processes to processors in real systems, most implementations of MPI ignore the "reorder" parameter and make no effort to optimize the placement of processes to processors.

Today, modern networks are quite good, with high bisection bandwidth and low latency. Many networks require only a few "hops" to connect any processor with any other. However, a major bottleneck remains—getting on and off of the processor chip, or more generally the node. This observation has led to a revision of the standard "postal" communication model [4] to include the maximum rate at which all of the processes on a node can move data into or out of the interconnection network.

This observation suggests a simple approach for implementing process reordering in `MPI_Cart_create`: order the processes *within* a node to minimize the number of off-node communications, *ignoring* the rest of the interconnect topology. This is the approach taken in this paper, with the algorithm described in Section 2.1 and results in Section 3, where we also compare with a vendor tool intended to perform the MPI rank mapping outside of the MPI application.

One last note is that the routine `MPI_Dims_create` does not take as input any information about the processes that would be included in a Cartesian topology, other than the total number of processes. Thus, this routine cannot help in informing the user about what choice of dimensions would be a good match for the underlying physical topology. This fact has also been noted by Träff [21].

1.1 Related Work

Almost from the beginning of MPI in 1992, work has been done on implementing the reorder feature of the MPI process topology routines, especially for Cartesian topologies. Most of these have been focused on effective mapping of MPI processes to the interconnect topology rather than considering which processes should be mapped to the same node. The following is a sampling of the work in this area.

The update to the process topology interface introduced in MPI-2.2 is covered in [6]; this focuses mostly on the introduction of the distributed graph topology. The Cartesian process topology routines used in this paper are unchanged since MPI-1.

There are a number of papers that discuss how to fully implement the MPI process topology mechanism. Träff describes an implementation for both Cartesian and graph topologies for the NEC SX-6 [20]. This paper used benchmarks written for the paper and does not include any applications, emphasizing the scarcity of codes that make use of the process topology functions. Hoefler and Snir [7] describe a general approach that is related to bandwidth minimization for sparse matrices, along with software that implements their approach. The software is available but is not packaged in a way that makes it an easy implementation route for the MPI process topology functions. Rashti et al [18] use the graph partitioning code Scotch to implement the MPI process topology functions and demonstrate significant performance improvements, showing good results on the application LAMMPS. However, the examples use only a relatively small number of processes, and the cost depends on the implementation of the Scotch graph partitioning library.

Mercier and Jeannot [13] describe an algorithm that works for the most general MPI process topology routine, `MPI_Dist_graph_create`. They show very good results but note that their algorithm is not scalable. They also focus only on the internal structure of the nodes and ignore the physical network topology.

Ma et al [12] consider locality and topology for intra-node communication; this work does not look at the MPI process placement. However, the considerations in this paper apply to mapping of processes within the node and could further improve the performance of the approach developed in this paper. As this approach relies on non-standard (though widely available) tools, it was not considered for this paper.

Zarrinchian et al [22] seeks to minimize the network contention from processes on the same node competing for the network interface, based on the size of message traffic between processes. While not in the context of MPI, this work is focusing on the same limited resource, the network interface.

Another approach is to use an external tool to determine the MPI process placement, which is then applied when the MPI job is initialized. Several tools have been developed to perform such placements, such as Cray's `pat_report -O mpi_rank_order`, which generates a file that controls the placement of MPI processes for subsequent runs.

Galver et al [3] describes TopoMapping, a tool that works for arbitrary applications by mapping tasks to processing elements, with results from runs on several systems with both torus and fat tree topologies. Like other such tools, this requires a separate profiling step.

In addition, there is an abundance of tutorials and advice available on "MPI Rank Assignment" or "MPI Rank Mapping," which usually depends on implementation- or system-specific mechanisms to place MPI processes on nodes or on sockets within nodes. The work in this paper relies only on features available within an MPI implementation, and has been implemented using only features in MPI 3.0.

Table 2: Results of using `MPI_Cart_create` on several systems showing that none remap the processes. Two open source MPI implementations are also included, as they are used on a wide variety of systems.

Name	System Type	MPI	Cart Remaps
Blue Waters	Cray XE6/XK7	Cray MPI	No
Theta	Cray XC40 (KNL)	Cray MPI	No
Piz Daint	Cray XC40/XC50	Cray MPI	No
(Open Source)		MPICH 3.2.1	No
(Open Source)		Open MPI 3.1.0	No

1.2 Behavior of Current Implementations

How effective are implementations of `MPI_Cart_create`? To test the implementations, the following tests were run:

(1) Does `MPI_Cart_create` remap MPI processes? This is tested by checking whether the created communicator is congruent in the MPI sense to the input communicator, meaning that their process groups have the same members and ordering.
(2) Given a 2D Cartesian decomposition and communication to the 4 neighbors, how many messages are sent off-node?
(3) Same question as above, but for 3D and the 6 neighbors.

On most systems, `MPI_Cart_create` did not remap the processes — in other words, it was a no-op and provided no benefit to the user. This would be acceptable if the default process mapping was effective for Cartesian process topologies, but Section 3 shows that this is not the case. Table 2 shows the systems tested and that none of them remap `MPI_Cart_create`. The open source versions of MPICH (version 3.2.1) and Open MPI (version 3.1.0) also do not reorder processes in `MPI_Cart_create`. While few if any current systems reorder processes in `MPI_Cart_create`, past systems such as the IBM BlueGene/L and NEC SX series [20] have implemented process reordering in some cases.

To look at the quality of the mapping, we consider the number of communication partners that are either on the same node or a different node. For a simple halo exchange (for a "star" or "plus" stencil), the communication partners are just the processes in the coordinate directions. To consider the number of partners in the two cases, the test program described in Section 3 was used. The results in Table 3 compare the number of on-node and off-node communication partners for both `MPI_Cart_create` and our node-aware replacement. That table shows that the standard implementation of `MPI_Cart_create` has more inter-node communication partners than is necessary, leading to suboptimal communication performance for operations such as a halo exchange.

2 USING NODE INFORMATION

As discussed above, while there has been a great deal of work describing methods that can provide high quality process mappings, these are rarely if ever used in practice by MPI implementations. This may be due to their complexity or cost in time. This raises the question: is there a simple approach that provides much of the benefit of a good process topology map without the complexity and time cost of more complex methods?

One key fact is that because of the rapid increase in the number of cores per processor chip, the amount of interprocessor and internode bandwidth has not kept up with the increase in the number of cores. As a result, the bandwidth per core is limited and the performance of MPI processes no longer follows the classic "postal" model but instead a "max-rate" model [4]. In this model, the communication cost is modeled as

$$T = \alpha + kn/\min(R_N, kR_C),$$

with latency α, the rate that each process can achieve in sending or receiving a message given by R_C, and the maximum rate that data can leave (or enter) the node and enter (respectively exit) the network (the *injection bandwidth*) given by R_N, and there are k simultaneously communicating processes on the node. In the case where R_N is very large, this reduces to the usual model with the inverse bandwidth $\beta = 1/R_C$. But when many processes on a node are communicating at the same time and R_N is not large compared with R_C, the available bandwidth per process is reduced to R_N/k. Thus, reducing the number of off-node communications is critical in achieving good MPI performance.

This suggests that simply arranging the mapping of processes so that the number of off-node communications is reduced will provide a significant benefit, *independent of the network topology*. The network topology does remain important, and an optimal process mapping may provide additional benefit; for example, [15] shows the impact of the reduced bandwidth along one dimension of the Cray Gemini network and the value to applications of taking this into account when assigning processes to nodes.

2.1 Algorithm

The algorithm is simple, exploiting the very regular structure of a Cartesian mesh. Two assumptions are made: (1) the resulting communicator will have the same number of processes per node and (2) in the implementation described, `MPI_Comm_split_type` with a type of `MPI_COMM_TYPE_SHARED` splits the communicator into processes on the same node. A future version of MPI may want to consider a type for splitting based on either nodes or on network interfaces, which is more accurate for this algorithm.

This algorithm has been implemented with the name `MPIX_Nodecart_create`, along with the corresponding Cartesian topology routines, such as `MPIX_Nodecart_shift`. For brevity, we will sometimes refer to the communicator created by this routine as a "nodecart" communicator. We have used the `MPIX_` prefix to conform to the MPI standard requirement that only functions in the standard begin with `MPI_` and used a different name to make it easy to compare with the provided implementation of `MPI_Cart_create`. A 'shim' library was also implemented to redirect the MPI Cartesian routines to use the Nodecart equivalents, making it easy to evaluate the nodecart approach with the few applications that take advantage of process reordering on Cartesian communicators.

(1) Identify the nodes, and create a communicator consisting of one process from each node. This communicator is called the `leadercomm`. Finding the nodes is easy and can be accomplished with `MPI_Comm_split_type`, and is shown in Step 1 in Figure 2 (under the assumption above that this splits by node. Note that the MPI implementation almost certainly knows which processes are on the same node, even if there is no corresponding split type). An MPI implementation may already have this information internally and not need to perform any communication. Creating the `leadercomm` is also easy, using `MPI_Comm_split`, and is shown in Step 2 in Figure 2.

(2) The size of `leadercomm` is the number of nodes; each process is now informed of the number of nodes in Step 3 in Figure 2.

(3) With a known number of nodes, and under the assumption (which should be confirmed by the implementation) that there are the same number of processes involved from each node, it is easy to create a two-level decomposition of the mesh—one for the processes on a node, and one for the nodes themselves. This is shown in Step 4 in Figure 2.

While we would like to take advantage of `MPI_Dims_create` to create the dimensions array for the node (the "intradims"), unfortunately we cannot. To see this, consider a 3D mesh of 576 processes on a system with nodes of 36 processes (thus 16 nodes). `MPI_Dims_create` will give 9, 8, 8 as the dimensions of a 3D mesh with 576 processes. But the 3D dimensions of 36 given by `MPI_Dims_create` are 4, 3, 3. There is no arrangement of these two sets of dimensions that will give a 3D array of 16 nodes. Instead, given 9, 8, 8 as the dimensions of the process array, the best sizes for the processes on the node are 9, 2, 2, which is unfortunately (but unavoidably) somewhat unbalanced.

This illustrates another way in which the design of the `MPI_Dims_create` API does not meet the needs of MPI programmers. In this case, a better overall set of dimensions might be 12, 8, 6, with the node dimensions of 6, 2, 3.

Instead, the approach taken is to factor the number of processes on each node and find a good decomposition of the dimensions for the processes on the node subject to the constraint that this decomposition works with the overall dimensions in `dims`:

```
Step 1: Factor node size into primes
Step 2: Set intradims[i] to 1 and
        remaindims[i] to dims[i]
Step 3: Take the largest prime factor fac
        find j such that
            fac divides remaindims[j] and
            remaindims[j] is larger than
            remaindims[k] for any k such
            that fac also divides remaindims[k]
Step 4: Set intradims[j] to intradims[j]*fac and
        remainsdims[j] to remaindims[j]/fac
Step 5: Repeat from Step 3 until all factors are
        distributed
```

(4) Given the information on the mesh of processes within a node, and the mesh of nodes, the coordinates within both of these meshes can be determined. This is a local operation that involves no communication, and is shown in Step 5 in Figure 2.

(5) Finally, given the coordinates of each process, and following the rules for the ranks of processes in a Cartesian communicator, the rank of each calling process within the new communicator can be determined. Using this rank in `MPI_Comm_split` creates a new communicator whose processes are reordered so that processes with a node are likely to be close together in the Cartesian communicator. This is shown in Step 6 in Figure 2.

(6) Not shown in Figure 2 are the steps needed to save information about the Cartesian topology that are needed to implement the routines that extract data from that communicator, e.g., the data needed to implement `MPI_Cart_shift` or `MPI_Cart_get`.

2.2 Optimizing the algorithm

Once the node and leader communicators are formed and the information on them is distributed to all processes in the original communicator, the construction of a new Cartesian communicator requires only a single `MPI_Comm_split` operation. In fact, it is possible to construct the full group locally on each process using `MPI_Group_incl`, then use `MPI_Comm_create_group` to create the communicator. On some MPI implementations, that may also be a local operation; on others, it may internally require only a single `MPI_Allreduce` operation.

The implementation saves information on the node and leader communicator in an attribute attached to the input communicator. Thus, subsequent uses of `MPIX_Nodecart_create` use that information and do not need to repeat the construction of these communicators. An MPI implementation may benefit from creating these communicators at initialization, which would simplify many operations that seek to be node-aware.

This approach can be applied to a multilevel hierarchy. For example, many compute nodes contain multiple processor chips; data communication performance between chips is usually lower than within a chip. If information is available that identifies which processor chip is running a process, and assuming that processes are not moved from one chip to another, this approach can ensure that the processes are mapped to the chips and nodes in such a way to reduce the internode and interchip communication. As many tutorials and tools for MPI process placement deal with this very issue, properly implementing `MPI_Cart_create` to take just the basics of the node topology into account would make MPI programs achieve better performance without requiring the use of nonstandard, non-portable, and volatile external tools and specifications (volatile in the sense that as nonstandard, there is no guarantee that these will not change in incompatible ways over time, causing programs to suddenly run less efficiently without any indication to the victimized user).

3 EXPERIMENTS

To test the effectiveness of `MPIX_Nodecart_create`, tests were run that performed an neighbor exchange in the coordinate directions for 2D and 3D Cartesian meshes. This code is called `ncartperf`, and is included in the implementation. A sketch of the core of the code is shown in Figure 3.

```
Input: oldcomm : communicator of processes
       dims    : dimensions of virtual Cartesian mesh
       numdims : number of dimensions
Output: nodecartcomm : communicator with processes
                       reordered in a Cartesian mesh
Local Variables:
  nodecomm : communicator of processes on the same node
  leadercomm : communicator of leaders on each node
  intradims  : Processor mesh for processes on a node
  interdims  : Processor mesh of the nodes (1 entry per
               node)
  intracoords: Coordinate of this process on node in
               mesh of intradims
  intercoords: Coordinate of this node in mesh of
               interdims
  coords     : Coordinate of this process in mesh (as
               reordered)
Step 1: Find nodes
   MPI_Comm_split_type(oldcomm, MPI_COMM_TYPE_SHARED,
                       rank, MPI_INFO_NULL, &nodecomm)
Step 2: Assemble comm of node leaders
   MPI_Comm_rank(nodecomm, &nrank)
   color = MPI_UNDEFINED
   if (nrank == 0) color = 0
  MPI_Comm_split(oldcomm, color, oldrank, &leadercomm)
Step 3: Inform all processes of the number of nodes
   if (color == 0) MPI_Comm_size(leadercomm, &nnodes)
   MPI_Bcast(&nnodes, 1, MPI_INT, 0, nodecomm)
Step 4: Find 2-level decomposition of dimensions
   MPI_Comm_size(nodecomm, &nsize)
   Confirm that nsize is the same for all processes in
       oldcomm (required for algorithm)
   Distribute factors of nodesize to create intradims,
       attempting to make intradims "square", with
        intradims[i] * interdims[i] == dims[i],
   and such that sum(interdims) is small
Step 5: Find coordinates in virtual mesh
   From rank in leadercomm, compute coordinates of
       node in mesh of size interdims
   From rank in nodecomm, compute coordinates of
       process in mesh of size intradims
   coords[i] = intracoords[i] +
             intercoords[i] * intradims[i]
Step 6: Find rank of process in output commmunicator
   based on coordinates in row-major order,
   and create communicator
   rr = coords[0]
   for (i=1; i<numdims; i++)
       rr = rr * dims[i] + coords[i]
   MPI_Comm_split(oldcomm, 0, rr, &nodecartcomm)
```

Figure 2: Node-aware algorithm for creating Cartesian communicators. This algorithm requires each node to have the same number of MPI processes in `oldcomm`.

```
// Create the communicator (similar code used
// for the nodecart communicator)
MPI_Comm_size(incomm, &ssize)
MPI_Dims_create(ssize, cartdim, dims)
MPI_Cart_create(incomm, cartdim, dims, periods,
                TRUE, &cartcomm)
for (i=0; i<cartdim; i++)
    MPI_Cart_shift(cartcomm, i, 1,
            &cartranks[2*i], &cartranks[2*i+1])

// Time a halo exchange
MPI_Barrier(cartcomm)
t0 = MPI_Wtime()
for (i=0; i<nr; i++)
     MPI_Irecv(bufptrs[i], msgsize, MPI_DOUBLE,
              ranks[i], 0, cartcomm, &rq[i])
for (i=0; i<nr; i++)
     MPI_Isend(bufptrs[nr+i], msgsize, MPI_DOUBLE,
              ranks[i], 0, cartcomm, &rq[nr+i])
MPI_Waitall(2*nr, rq, MPI_STATUSES_IGNORE)
MPI_Barrier(cartcomm)
t  = MPI_Wtime()-t0
```

Figure 3: Sketch of the test performed by the program `ncartperf`. Code used to collect multiple test results and report the results are not shown.

Tests were run on three systems. First, Blue Waters, a Cray XE6/XK7 at the University of Illinois at Urbana-Champaign with 22,636 XE6 nodes and 4228 XK7 nodes, connected in a 3D mesh with the Gemini interconnect. Each XE6 node has 2 AMD Interlagos processors each with 8 core modules (or 16 cores) for a total of 16 core modules or 32 cores per node. For these experiments, only the XE6 nodes were used. For most floating-point intensive applications, 16 threads per XE node is often recommended, and was used in the tests. Second, Theta, an Cray XC40 at Argonne National Laboratory with 4392 nodes of Intel KNL processors, with an Aries interconnect with a Dragonfly configuration. Third, Piz Daint, a Cray XC50/XC40 at the Swiss National Supercomputing Center with 5320 XC50 nodes and 1813 XC40 nodes, also with an Aries interconnect in a Dragonfly configuration.

Each test consisted of 32 iterations of a stencil sweep. Each test was run twenty times, with the first run ignored (to avoid cold-start issues) and the minimum and average times over the ten tests were computed. Rates were computed based on the minimum time, as this usually provides the most reproducible value [5]. The code `ncartperf`, included in the implementation of the nodecart communicator creation, was used for the tests.

As Figures 4–9 show, the nodecart communicator provides substantially better communication performance for the halo exchange for all three systems.

3.1 Stencil Computation Benchmark

To test the potential impact on applications, a sample code that tests various communication approaches for a stencil application to a 2D mesh was modified to use both the MPI Cartesian process

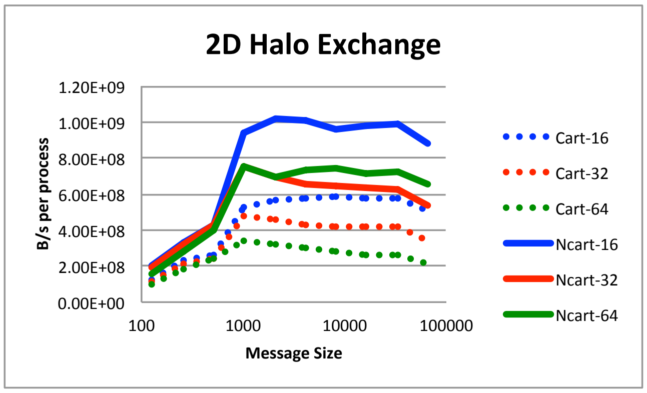

Figure 4: Results for simple halo exchange for a 2D mesh on upto 4K processes on Blue Waters. "Cart-16" is with a communicator from `MPI_Cart_create` on a 16×16 process mesh; Ncart is with the nodecart communicator described in this paper. In all plots, the results using "Cart" are dotted lines and with "Ncart" are solid lines.

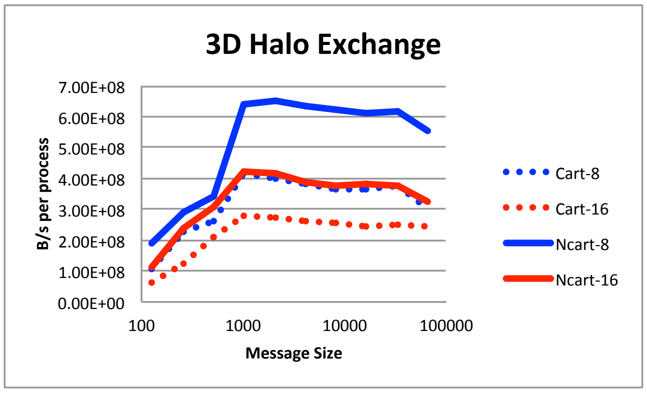

Figure 5: Results for simple halo exchange for a 3D mesh on upto 4K processes on Blue Waters. The legend is the same as Figure 4, except the process mesh is 3D as in $8 \times 8 \times 8$.

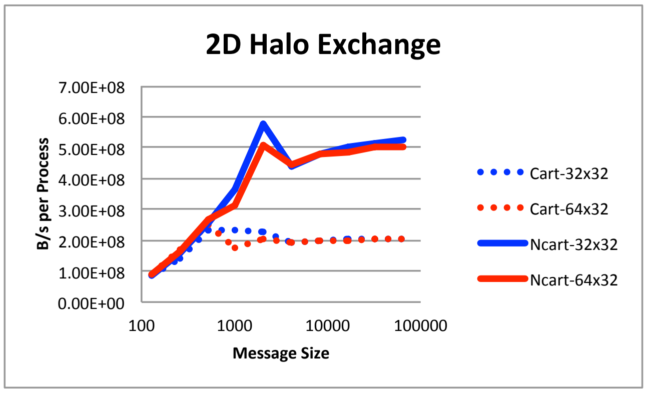

Figure 6: Results for simple halo exchange for a 2D mesh on upto 64K processes on Theta.

Table 3: Number of on-node and off-node communication partners for each process for a shift of 1 in each coordinate direction for communicators created with `MPI_Cart_create` and `MPIX_Nodecart_create`, where each node has 16 processes.

Type	Dim	Size	On-node			Off-node		
			min	max	avg	min	max	avg
Cart	2	128×128	1	2	1.88	2	3	2.12
Nodecart	2	128×128	2	4	3	0	2	1
Cart	3	$32 \times 32 \times 16$	1	2	1.88	4	5	4.12
Nodecart	3	$32 \times 32 \times 16$	3	4	3.5	2	3	2.5

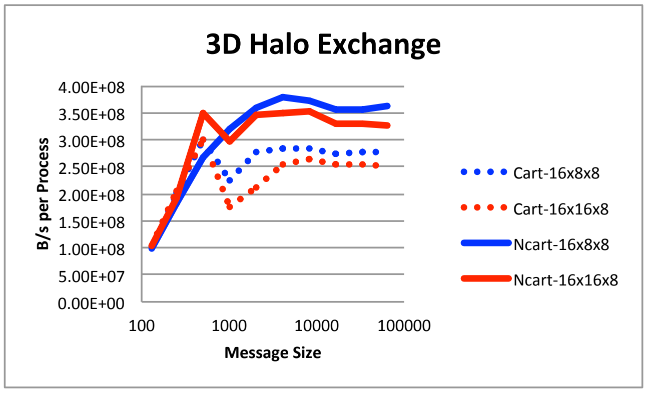

Figure 7: Results for simple halo exchange for a 3D mesh on Theta.

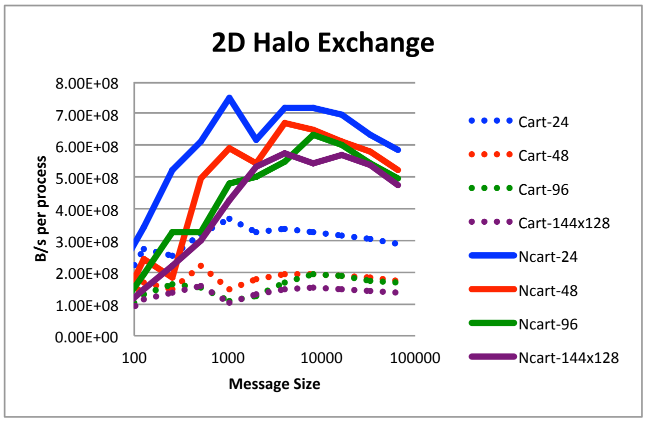

Figure 8: Results for simple halo exchange for a 2D mesh on upto 18,432 processes on Piz Daint, with the same meaning for the legend as in Figure 4.

topology and the "nodecart" process topology. This code, named `stencil`, has been used for tutorials on Advanced MPI, offered at the annual SC conference for the last several years; the code is based on code originally written by Torsten Hoefler. The code is relatively straightforward and the communication is similar to that shown in Figure 3 (in fact, the code implements a number of different communication methods; the use of nonblocking send and receive with `MPI_Waitall` is among the best performing and is used for these results). A simple 5-point approximation to the

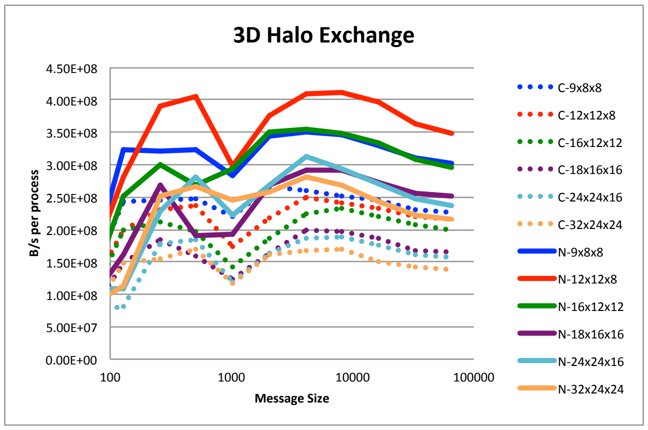

Figure 9: Results for simple halo exchange for a 3D mesh on upto 18,432 processes on Piz Daint. "C-nnxmmxpp" denotes an MPI Cartesian communicator on an nn × mm × pp mesh. "N-. . ." the same, but for the nodecart communicator described in this paper.

Table 4: Results for 2D stencil computation on 4096 processes on Blue Waters. Columns show computational mesh size ($n \times n$) for the global mesh, computation rate in GFLOP/s, and the ratio of rates, showing a modest advantage for the NodeCart communicator.

n	Cart	NodeCart	Ratio
256	30.1	32.4	1.08
512	118	121	1.03
1024	432	491	1.14
2048	1550	1670	1.08
4096	3170	3370	1.11
8192	4400	4710	1.07
16384	5960	6170	1.04

Poisson operator is computed. Results for runs on Blue Waters with 4096 processes are shown in Table 4. These show a small but consistent performance improvement, even for this example. For computational mesh sizes larger than $16K \times 16K$, the computational time dominates, and the difference between the two approaches disappears.

Table 5: Bandwidth in MB/s per process for 3D Halo Exchange on Blue Waters. "World" is `MPI_COMM_WORLD`; "Nodecart" is the implementation of Cartesian topology remapping described in this paper. Results are for the default mapping and for the mapping recommended by `pat_report`. There are 16 processes on each node and 64 nodes.

	Rank Mapped		Default	
n	World	Nodecart	World	Nodecart
128	45	136	98	138
256	132	242	224	266
512	152	294	260	318
1024	205	487	489	616
2048	193	486	483	616
4096	185	483	477	606
8192	179	471	469	584
16384	172	449	456	579
32768	168	442	461	580
65536	148	378	398	508

3.2 Comparison to Rank Mapping

An alternative to using the MPI topology routines is to use a tool that traces the MPI communication in a sample run, then produces a mapping of the MPI ranks in `MPI_COMM_WORLD` to specific processing elements. For example, Cray provides `pat_build` that creates an instrumented executable and `pat_report` that produces a performance analysis and a file that specifies a rank mapping. This rank mapping can then be used in subsequent runs by setting the environment variable `MPICH_RANK_REORDER_METHOD` to `3`.

This approach has both advantages and disadvantages. The advantage is that it requires no changes to the application and can improve the performance of the application. The disadvantages include that it (a) depends on a static communication pattern, (b) is not portable to another system, (c) requires a separate run to collect information, (d) does not take advantage of known structure of the communication, and (e) is made independent of the placement of processes on the physical hardware.

To compare the use of a rank mapping tool with our implementation of `MPI_Cart_create`, a comparison was run, using the `ncartperf` program, for a 3D halo exchange with 1024 processes on 64 nodes of Blue Waters, following the process documented at [1]. The results are shown in Table 5. The rank mapping was computed by running the halo exchange only with `MPI_COMM_WORLD`. These show that for this code, the rank mapping tool does not improve performance and our node-based implementation of `MPI_Cart_create` provides better performance in both the default mapping and the mapping provided by the Cray tool `pat_report`.

These results are a bit surprising. A closer examination shows that in the case where `MPI_COMM_WORLD` is remapped, the average number of on-node communication partners per process is only 0.08 and the average number of off-node partners is 5.92. For the unmapped case, it is 2.75 for on-node and 3.25 for off-node. This compares with the "nodecart" communicator, which in both cases (that is, independent of the original mapping of processes to cores and nodes) has an average of 3.5 on-node partners and 2.5 off-node partners. One hypothesis is that the rank assignment tool is optimizing for the network, not the placement on the nodes.

4 CONCLUSION

In this paper we showed that a simple implementation of the MPI Cartesian process topology that uses only information about which processes are on the same node can provide significant benefits in terms of interprocess communication performance. This implementation is simple, fast, and robust, and can be added to both open source MPI implementations and used as the starting point for platform-specific MPI implementations.

This approach can be further refined, for example, by taking advantage of the intranode interconnect topology. Extending this to the general case of graph process topologies is more difficult; the simplicity of this approach exploits the implicit information about the process topology that is present in a Cartesian mesh. However, an approach that simply looks to partition a graph into domains with minimum surface to volume ration may be easier to implement and to scale than more complex (though more optimal) mappings.

What if the user prefers to use external rank mapping tools when they are available? This is easy, because of the design of the MPI Topology API. The application need only provide an option to select the value of the `reorder` parameter to the `MPI_Cart_create` routine. By default, this should always be true, permitting the MPI implementation to find a good placement of processes, whether it is using the simple mechanism outlined in this paper or a more powerful (but complex and perhaps more computationally expensive) graph partitioner. But by setting this one parameter to false, any of the external rank reordering tools can be used with the application.

With this algorithm, MPI users can now expect and should demand an `MPI_Cart_create` that helps them assign their processes to the nodes of their parallel computer.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07–25070) and the state of Illinois. The author is grateful to Rajeev Thakur for runs on the Theta system at Argonne National Laboratory and to Torsten Hoefler and Timo Schneider for runs on the Piz Daint system at the Swiss National Supercomputing Center. The author also thanks the reviewers for their careful reading and helpful suggestions for improving the presentation.

REFERENCES

[1] Blue Waters Project 2018. Topology Considerations. https://bluewaters.ncsa.illinois.edu/topology-considerations. (2018).
[2] CORAL 2014. CORAL Collaboration Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/. (2014).
[3] Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. 2017. Automatic Topology Mapping of Diverse Large-scale Parallel Applications. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 17, 10 pages. https://doi.org/10.1145/3079079.3079104
[4] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919
[5] William D. Gropp and Ewing Lusk. 1999. Reproducible Measurements of MPI Performance Characteristics. In *Recent Advances in Parallel Virtual Machine and*

Message Passing Interface (Lecture Notes in Computer Science), Jack Dongarra, Emilio Luque, and Tomàs Margalef (Eds.), Vol. 1697. Springer Verlag, 11–18. 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[6] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff. 2010. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience* 23, 4 (Aug. 2010), 293–310.

[7] T. Hoefler and M. Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–85.

[8] HOMB 2009. Hybrid OpenMP MPI Benchmark 1.0. https://sourceforge.net/projects/homb/. (May 2009).

[9] HPCC 2003. HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/. (2003).

[10] LAMMPS [n. d.]. LAMMPS Benchmarks. http://lammps.sandia.gov/bench.html. ([n. d.]).

[11] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. 2005. *Introduction to the HPC Challenge Benchmark Suite*. Technical Report LBNL-57493. Lawrence Berkeley National Laboratory. https://www.osti.gov/servlets/purl/860347.

[12] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack J. Dongarra. 2010. Locality and Topology Aware Intra-node Communication Among Multicore CPUs. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*. Springer-Verlag, Berlin, Heidelberg, 265–274. http://dl.acm.org/citation.cfm?id=1894122.1894158

[13] Guillaume Mercier and Emmanuel Jeannot. 2011. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *Recent Advances in the Message Passing Interface*, Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–49.

[14] NAS 2016. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html. (2016).

[15] Antonio J. Peña, Ralf G. Correa Carvalho, James Dinan, Pavan Balaji, Rajeev Thakur, and William Gropp. 2013. Analysis of topology-dependent MPI performance on Gemini networks. In *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*, Jack Dongarra, Javier García Blas, and Jesús Carretero (Eds.). ACM, 61–66. http://dl.acm.org/citation.cfm?id=2488551

[16] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (March 1995), 1–19. https://doi.org/10.1006/jcph.1995.1039

[17] POP. 2003. *Parallel Ocean Program (POP) User Guide Version 2.0*. Technical Report LACC 99-18. Los Alamos National Laboratory.

[18] Mohammad J. Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. 2011. Multi-core and Network Aware MPI Topology Functions. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings (Lecture Notes in Computer Science)*, Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra (Eds.), Vol. 6960. Springer, 50–60. http://dx.doi.org/10.1007/978-3-642-24449-0

[19] SPP 2017. SPP-2017 Instructions and Reporting. (2017). https://bluewaters.ncsa.illinois.edu/spp-methodology.

[20] Jesper Larsson Träff. 2002. Implementing the MPI Process Topology Mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=762761.762767

[21] Jesper Larsson Träff and Felix Donatus Lübbe. 2015. Specification Guideline Violations by MPI_Dims_create. In *Proceedings of the 22Nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 19, 2 pages. https://doi.org/10.1145/2802658.2802677

[22] Ghobad Zarrinchian, Mohsen Soryani, and Morteza Analoui. 2012. A New Process Placement Algorithm in Multi-core Clusters Aimed to Reducing Network Interface Contention. In *Advances in Computer Science, Engineering & Applications*, David C. Wyld, Jan Zizka, and Dhinaharan Nagamalai (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1041–1050.