Mapping Applications with Collectives over Sub-communicators on Torus Networks

Abhinav Bhatele[†], Todd Gamblin[†], Steven H. Langer[†], Peer-Timo Bremer[†,‡], Erik W. Draeger[†], Bernd Hamann[*], Katherine E. Isaacs[*], Aaditya G. Landge[‡], Joshua A. Levine[‡], Valerio Pascucci[‡], Martin Schulz[†], Charles H. Still[†]

[†]Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, California 94551 USA
[‡]Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, Utah 84112 USA
[*]Department of Computer Science, University of California, Davis, California 95616 USA

***Abstract*— The placement of tasks in a parallel application on specific nodes of a supercomputer can significantly impact performance. Traditionally, this *task mapping* has focused on reducing the distance between communicating tasks on the physical network. This minimizes the number of hops that point-to-point messages travel and thus reduces link sharing between messages and contention. However, for applications that use collectives over sub-communicators, this heuristic may not be optimal. Many collectives can benefit from an increase in bandwidth even at the cost of an increase in hop count, especially when sending large messages. For example, placing communicating tasks in a cube configuration rather than a plane or a line on a torus network increases the number of possible paths messages might take. This increases the available bandwidth which can lead to significant performance gains.**

We have developed *Rubik*, a tool that provides a simple and intuitive interface to create a wide variety of mappings for structured communication patterns. Rubik supports a number of elementary operations such as splits, tilts, or shifts, that can be combined into a large number of unique patterns. Each operation can be applied to disjoint groups of processes involved in collectives to increase the effective bandwidth. We demonstrate the use of Rubik for improving performance of two parallel codes, pF3D and Qbox, which use collectives over sub-communicators.

I. Introduction

The mapping of parallel tasks in an application to the network topology has traditionally been aimed at reducing the distance between communicating tasks to minimize link sharing and congestion [1], [2], [3], [4]. This works well for applications that have point-to-point communication with a small number of neighbors for each task and collectives over global communicators. In these situations, minimizing the number of hops for point-to-point messages is appropriate since collectives over global communicators are, by definition, uninfluenced by the mapping.

The increase in number of nodes and thus network diameters has forced application developers to revisit the collectives deployed in their codes and to restrict them wherever possible to sub-communicators instead of the global communicator. In such cases, the overall performance can be significantly impacted by how these sub-communicators are placed on the network. However, finding an optimal mapping for such groups of collectives can be challenging. For example, creating compact groups reduces the hop count, yet may leave a large number of hardware links on their boundaries unused, limiting the overall bandwidth utilization. Further, the actual performance of a collective is heavily dependent on multiple layers of system software starting with the MPI implementation to the packet routing algorithms. Even though topology aware algorithms have been developed for *implementing* collectives over torus and fat-tree networks [5], [6], we are not aware of a systematic study that deals with mapping applications to improve the performance of collectives over sub-communicators.

This paper presents a preliminary study of the performance of collectives, in particular all-to-alls and broadcasts, over sub-communicators on n-dimensional (nD) torus networks. The underlying premise is that placing communicators such that they span multiple dimensions and utilize the wrap-around torus links can increase the effective bandwidth as well as provide more message routes which can reduce congestion. For example, an all-to-all over eight nodes in a straight line will typically be significantly slower than one over a cube of $2 \times 2 \times 2$ nodes. While existing libraries [3], [4] focus on minimizing the number of hops to reduce latency, we propose a new tool called *Rubik* to maximize bandwidth utilization through the use of links in as many dimensions as possible.

Optimizing the mapping of collectives can be challenging as their performance may depend on several factors: the algorithms used for implementing the collectives in the underlying MPI layer; the protocols used for different message sizes; and the strategies used for routing packets on the hardware. Additionally, most non-trivial mappings will result in a complex interference patterns among the simultaneous collectives which are difficult to predict.

Exploring the entire set of potential mappings is infeasible and developing an automatic optimization tool is non-trivial. Instead, Rubik provides a simple and intuitive interface for developers to use and create a wide variety of mappings aimed at utilizing more hardware links while avoiding excessive latency or congestion. In particular, Rubik provides:

- A simple notation to describe both machine and application topologies as nD Cartesian spaces;
- The ability to hierarchically define and manipulate *groups* of processes involved in some collective;
- A number of dimension-independent operations such as

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

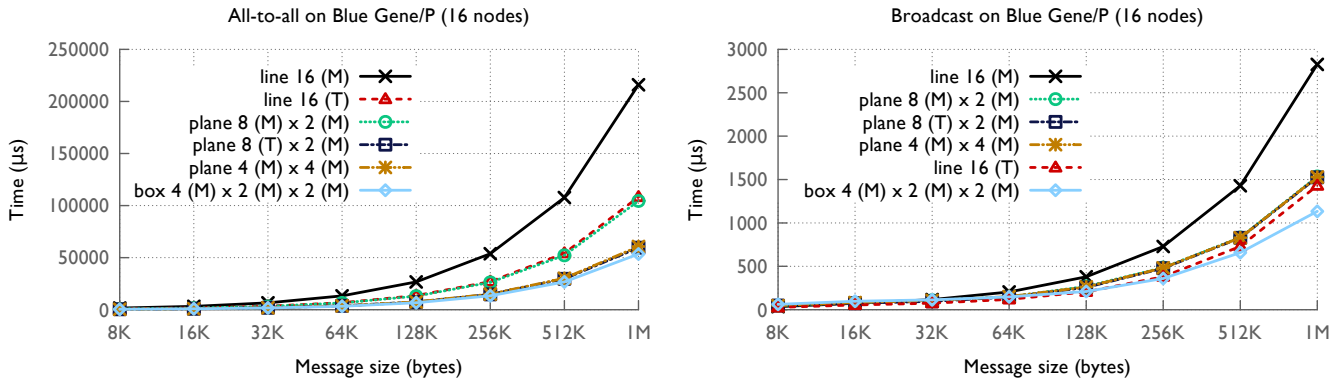


Fig. 1: Performance of all-to-all and broadcast operations with different mappings on Blue Gene/P (Intrepid)

tilt and *mod* which when applied to groups tend to increase the number of links utilized by a collective; and
- An intuitive visualization tool for three-dimensional (3D) machine topologies illustrating the resulting mapping.

We demonstrate the applicability of Rubik using two highly scalable production applications deployed at Lawrence Livermore National Laboratory (LLNL): pF3D, a laser-plasma interaction code and Qbox, a first-principles molecular dynamics application. We show that using Rubik, we can easily create hundreds of mappings with few lines of Python code. Further, we compare the observed network traffic of some of the best and worst mappings to provide initial insights into the causes for the differences in performance.

II. Increasing Link Utilization and Bandwidth

On nD torus networks such as those on the IBM Blue Gene machines, sufficiently large messages may be routed adaptively to minimize hot-spots or congestion on the network. A careful mapping of communicating tasks to the physical network can assist the system software and hardware in achieving this goal. Consider a message sent between two nodes on an nD Cartesian network. Depending on where these tasks are located, there are one, two, or more routes that the message can take. Looking at Fig. 2, we see that if the tasks are placed on a line, there is a single shortest path route between them. If we place the tasks on the diagonally opposite corners of a 2×2 plane, there exist two shortest paths between them, and twice the available bandwidth. Also, when one of these paths is being used by other messages, the other path can be used to avoid congested links.

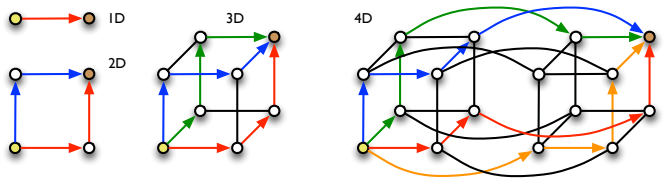


Fig. 2: Disjoint (non-overlapping) paths and "spare" links between a pair of nodes on n-dimensional topologies

Increasing the number of dimensions and placing the nodes at opposite corners of a hypercube results in more paths that can be used to route packets of a message. For the $2 \times 2 \times 2$ box shown in Fig. 2, there are six shortest paths. However, each of these shares one link with another path. Consequently, there are only three disjoint paths emerging from the source node and ending at the destination, but there are three extra links (shown in black) that can be used to split the traffic after the first hop. With a $3 \times 3 \times 3$ box (not shown), the number of disjoint paths stays the same (decided by the number of outgoing links from the source node) but there exist more extra links. For a $2 \times 2 \times 2 \times 2$ hypercube, there are four disjoint paths and 16 extra links. In general, increasing the number of dimensions increases the number of disjoint shortest paths as well as the number of "spare" links while increasing the distance between nodes increases the number of "spare" links.

The total number of links for a mesh with d dimensions and n nodes in each dimension is $n^{d-1}(n-1) \times d$. The number of disjoint paths between a source-destination pair at opposite corners of the mesh is the number of outgoing links, d. The total number of links used in those d routes is $d \times d$. Hence the number of spare intermediate links is: $(n^{d-1}(n-1) \times d) - d^2$. This number increases exponentially with the increase in the number of dimensions. So, as we go to higher dimensional meshes or tori, we drastically increase the number of available paths a message can be routed along.

Another way of increasing the number of routes is adding wrap-around torus links to a mesh which increases the total number of links by a factor of $n/(n-1)$ (number of links on a torus is $n^d \times d$). Having extra links that can be used to route messages increases the available bandwidth and reduces the chances of network congestion. In this paper, we investigate mapping strategies that exploit these three different ways of optimizing the layout of collectives:

- Placing communicating tasks on the corners of a plane/box or mesh of higher dimension instead of a line.
- Increasing the distance between communicating pairs to create more spare links
- Using wraparound torus links as additional routes.

In Fig. 1 and 3, we support our hypothesis by comparing several mappings of two different collective operations on the Blue Gene/P (BG/P) and Blue Gene/Q (BG/Q) architectures. We chose MPI_Alltoall and MPI_Bcast as the two operations

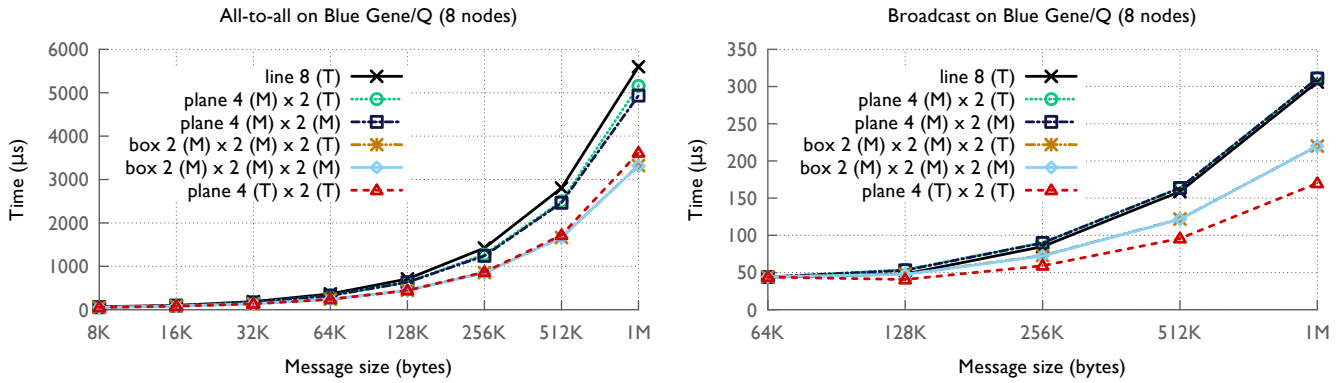


Fig. 3: Performance of all-to-all and broadcast operations with different mappings on Blue Gene/Q (Veas)

because they are heavily used in the application codes used later in the paper. Fig. 1 compares the performance of mapping these collectives over 16 nodes of a 4,096 node BG/P allocation (a $8 \times 16 \times 32$ torus). We can arrange the 16 nodes in different ways: all 16 nodes in a line (mesh or torus), a plane of 8×2 or 4×4 and a box of $4 \times 2 \times 2$.

For the all-to-all plot in Fig. 1, doing the collective over a 16×1 torus instead of a mesh improves the performance by 50% for 1 MB messages. When the all-to-all is done over a 8×2 mesh, we get the same improvement. If we change the longer dimension of this mesh to a torus, we get an additional 22% improvement. Having more routes in a 4×4 plane or a $4 \times 2 \times 2$ box also gives the same performance boost. These results clearly support the strategies outlined above for improving communication performance. The broadcast also achieves similar improvements with a maximum reduction in runtime of 60% for 1 MB messages with the best mapping (compared to 75% for the all-to-all). It is important to note that for the broadcast, a 16×1 torus leads to better improvements than using a plane. Hence, mapping requires a careful consideration of the communication patterns one is trying to optimize and an understanding of how they are implemented in software and the underlying hardware.

Fig. 3 shows similar results for the five-dimensional (5D) BG/Q network using an all-to-all and broadcast over 8 nodes. The size of the allocated partition was 1024 nodes (a $4 \times 4 \times 4 \times 8 \times 2$ torus). The improvements are smaller as compared to BG/P but still significant - 40% reduction in time for the all-to-all and 44% for the broadcast using the best mapping for 1 MB messages. Hence, even on higher dimensional networks that provide more routes, higher bisection bandwidth, and lower latencies, a careful mapping may still improve performance.

III. The Rubik mapping tool

We have developed Rubik, a tool that simplifies the process of creating task mappings for structured applications. Rubik allows an application developer to specify communicating groups of processes in a virtual application topology succinctly and map them onto groups of processors in a physical network topology. Both the application topology and the network topology must be Cartesian, but the dimensionality of either is arbitrary. This allows users to easily map low-dimensional structures such as planes to higher-dimensional structures like cubes to increase the number of links used for routing.

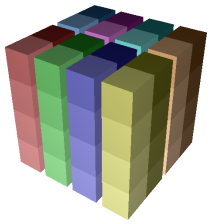
Rubik also provides embedding operations that adjust the way tasks are laid out within groups. These operations are intended to optimize particular types of communication among ranks in a group, either by shifting them to increase the number of available links for communication between processor pairs (as in Fig. 2), or by moving communicating ranks closer together on the Cartesian topology to reduce latency. In conjunction with Rubik's mapping semantics, these operations allow users to create a wide variety of task layouts for structured codes by composing a few fundamental operations, which we describe in the following sections.

A. Partition trees

The fundamental data structure in Rubik is the *partition tree*, a hierarchy of nD Cartesian spaces. We use partition trees to specify groups of tasks (or processes) in the parallel application and groups of processors (or nodes) on the network. Nodes of a partition tree represent boxes, where a `box` is an nD Cartesian space. Each element in a box is an object that could be a task or a processor. New boxes are filled by default with objects numbered by rank (much like MPI communicators).

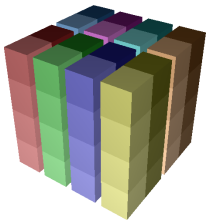
Every partition tree starts with a single root `box` representing the full nD Cartesian space to be partitioned. We construct a box from a list of its dimensions, e.g., a $4 \times 4 \times 4$ 3D application domain. From the root, the tree is subdivided into smaller child boxes representing communication groups (MPI sub-communicators) in the application. Child boxes in a partition tree are disjoint, and the union of any node's child boxes is its own box. Unlike other tools, which are are restricted to two or three dimensions, Rubik's syntax works for any number of dimensions. An arbitrary number of dimensions can be specified when a box is constructed.

Fig. 4 shows the Rubik code to construct a partition tree, with incremental views of the data structure as it is built. On line 1, we construct a $4 \times 4 \times 4$ domain using the `box` command. This creates a one-level tree with a single box of 64 tasks (Fig. 4a). In line 2, we use Rubik's `div` command to



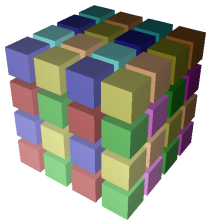
```
app = box([4,4,4])
app.div([2,1,4])
```

(a) `div` operation



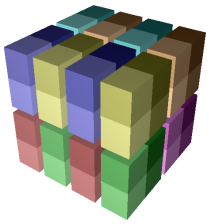
```
app = box([4,4,4])
app.tile([2,4,1])
```

(b) `tile` operation



```
app = box([4,4,4])
app.mod([2,2,2])
```

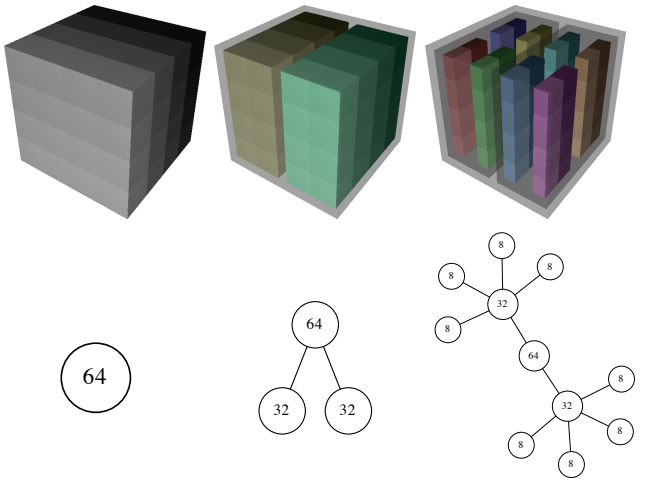
(c) `mod` operation



```
app = box([4,4,4])
app.cut([2,2,2],
        [div,div,mod])
```

(d) `cut` operation

Fig. 5: Partitioning operations in Rubik: `div`, `tile`, `mod` and `cut`



(a) One level (b) Two levels (c) Three levels

```
domain = box([4,4,4])   # a. Create a cube
domain.div([1,1,2])     # b. Divide into halves
for child in domain:
    child.div([2,1,2]) # c. Divide each half into 4
```

Fig. 4: Incremental partition tree construction using the `div` operation

split this tree along the third dimension into two boxes of 32 tasks (Fig. 4b), which fully cover the original box. Lines 3 and 4 loop over the newly created children and further split each child into 4 children of its own, with 8 tasks each (Fig. 4c).

The cubes at the top of Fig. 4 show the Cartesian structure of the tree. Leaf nodes are nested in transparent halos of their parent boxes. Each leaf box is given a unique color, and object numbering (MPI rank) within each leaf box is shown using a color gradient. The lowest rank within a leaf box has the lightest color. The tree diagrams below the cubes show the partition tree structure with boxes shown as nodes and labeled by the number of tasks they contain.

B. Partitioning operations

The `div` operation used in the previous section is one of four operations in Rubik that divide a box into children: `div`, `tile`, `mod` and `cut`. Like the `box` constructor, these operations can be used on an arbitrary number of dimensions.

Div. Section III-A showed two hierarchical `div`s to illustrate the concept of the partition tree and Fig. 5a shows how we can chop a $4 \times 4 \times 4$ cube into 8 leaves of 8 objects each using a single `div`. `div` takes a set of *divisors* $d_0, d_1, ...d_n$ as argument, one for each dimension of the box it divides. It slices the parent box into d_i groups along dimension i, creating $\prod_{i=0}^{n-1} d_i$ child boxes. The child boxes form a $d_0 \times d_1 \times ... \times d_n$ space where the task at position $(x_0, x_1, ..., x_n)$ in the parent box is in the child box with index $(\frac{x_0}{d_0}, \frac{x_1}{d_1}, ..., \frac{x_n}{d_n})$.

Tile. While `div` divides a space into a fixed number number of groups, `tile` divides a space into fixed-size child boxes, or *tiles*. The number of tiles created depends on the size of the box that `tile` is applied to. Arguments to `tile` are tile dimensions rather than divisors. Formally, `tile` on a $D_0 \times D_1 \times ... \times D_n$ space is equivalent to `div` with divisors $\frac{D_0}{d_0}, \frac{D_1}{d_1}, ..., \frac{D_n}{d_n}$. Figs. 5a and 5b show the same boxes created using `div` and `tile`.

Mod. The `mod` operation shown in Fig. 5c is similar to *div* in that it also takes a list of n divisors and creates $\prod_{i=0}^{n-1} d_i$ child boxes. However, `mod`'s child boxes are interleaved, not contiguous. With `mod`, task $(x_0, x_1, ..., x_n)$ will be a member of the child box $((x_0 \bmod d_0), (x_1 \bmod d_1), ..., (x_n \bmod d_n))$.

Cut. The `cut` operation shown in Fig. 5d is a generalization of `div` and `mod`. `cut` takes the same set of divisors as `div` and `mod`, but it also takes a second list that specifies the manner of slicing in each dimension. In the picture, we can clearly see that `cut` creates contiguous slices along dimensions where `div` is specified, but along the third dimension which uses `mod`, the child boxes are interleaved.

C. Mapping

Partition trees in Rubik are used not only to specify groups of tasks in a Cartesian application domain, but also to specify groups of processors on the physical network. The tool is designed to simplify the process of mapping tasks between spaces with potentially different dimensionality. A fundamental example is that of mapping planes to boxes. Scientific applications may perform collective operations within a plane in the application domain, but mapping a plane directly onto a 3D mesh network will not maximize the number of physical links available for communication within the plane. Mapping

```
# Create app partition tree of 27-task planes
app = box([9,3,8])
app.tile([9,3,1])

# Create network partition tree of 27-processor
    cubes
network = box([6,6,6])
network.tile([3,3,3])

network.map(app)  # Map task planes into cubes
```

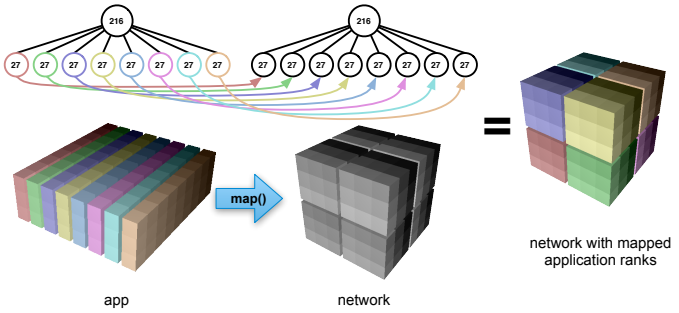
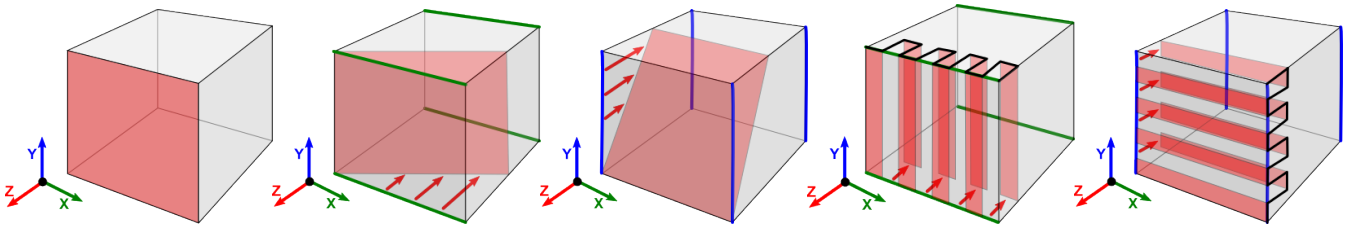


Fig. 6: Mapping a partition tree from the application to network domain using Rubik



(a) `tilt(Z,*,*)` (b) `tilt(Z,X,*)` (c) `tilt(Z,Y,*)` (d) `zigzag(Z,X,*)` (e) `zigzag(Z,X,*)`

Fig. 7: `tilt` and `zigzag` operations for the Z plane of a three-dimensional box (a). The first operand of both operations selects the planes to modify; the second selects the directions along which increasing (`tilt`) or alternating (`zigzag`) shifts are applied.

the plane's tasks to a higher dimensional space allows more bandwidth to be exploited. Rubik makes this easy by facilitating mapping for arbitrary number of dimensions.

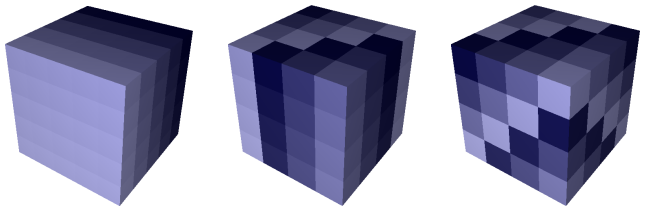
Fig. 6 shows two boxes of 216 objects subdivided into eight 27-object groups. The first box's children are planes, and the second box's children are cubes. Regardless of the particular structure, the number of leaves in the two partition trees is the same and each is of the same size. Such trees are considered *compatible*. Two compatible trees can be `mapped` by performing a simple breadth-first traversal of their leaves and pairing off successive child boxes. The arrows in the figure show these pairings for the example. For each pair, we take the tasks in the child boxes in the application domain and copy them into the corresponding boxes in the network domain.

The Rubik `map` operation reduces the burden of mapping multi-dimensional spaces by allowing the user to think only in terms of group sizes. The particular shapes of groups are specified separately using the simple partitioning operations discussed above. All that is required for a `map` is tree compatibility. Indeed, despite their drastically different shapes, `map` can be applied to any two partition trees shown in Fig. 5, because their leaves are all the same size. They could also be mapped to the tree in Fig. 4c, even though it has more levels. These partitions could even be mapped to four- or five-dimensional boxes, as long as their leaves are compatible.

D. Permuting operations

By default, the Rubik `map` operation copies ranks between Cartesian spaces in scan-line order, with the highest-indexed dimension varying fastest. While this is an intuitive default order, a user may want to permute ranks within groups to target bandwidth or latency optimizations. Rubik has several operations that allow tasks to be permuted to exploit properties of the physical network: `tilt`, `zigzag`, and `zorder`.

Tilt. The `tilt` operation can increase the number of links available for messaging on nD Cartesian networks. Fig. 7b and 7c show illustrations of two tilts applied to a 3D box. Conceptually, `tilt(op1,op2,op3)` selects one hyperplane (denoted by `op1`) and a direction (`op2`) along which an increasing number (`op3`) of shifts are applied normal to the direction of the hyperplane. Shifts are applied in a circular fashion to all parallel hyperplanes resulting in a permutation that "tilts" each hyperplane. Fig. 8 shows multiple, successive applications of the tilt operation to a $4 \times 4 \times 4$ box. On the left is an untilted box, with tasks colored by identity (MPI rank) from lightest to darkest. In the center, we see the same tasks after permutation by one tilt, and on the right is the same box after two tilts have been applied.



```
Z, Y, X = 0, 1, 2  # Assign names to dimensions
net = box([4,4,4]) # Create a box
net.tilt(Z, X, 1)  # Tilt Z (XY) planes along X
net.tilt(X, Y, 1)  # Tilt X (YZ) planes along Y
```

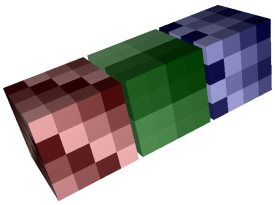
Fig. 8: Untilted, once-tilted, and twice-tilted 3D boxes

`tilt` uses the insights described in Section II to increase the available links for communication between neighbor tasks

in the manner described in Fig. 2. For example, by shifting hyperplanes normal to X, we add links in the X dimension that neighbors can use for communication. Successive tilts in additional dimensions, as shown on the right in Fig. 8, add links in more dimensions. The higher the dimension of the network the more independent `tilts` can be performed and the more links can be exploited.

Zigzag. The `zigzag` operation is similar to the tilt operation in that it shifts hyperplanes along a dimension. However, rather than shifting each successive plane by an increasing amount, `zigzag` only shifts alternating segments by a constant amount. This targets bandwidth in effectively the same way that `tilt` does, by adding links along the permuted dimension. However, `zigzag` has better latency properties than `tilt` since tasks stay closer to their starting point after a `zigzag` than they would with `tilt`. Figs. 7d and 7e show illustrations of two `zigzag` operations applied to a 3D box.

Zorder. Z-ordering is a space-filling curve that maps a multi-dimensional space to a linear curve while partially preserving multi-dimensional locality. Space-filling curves have been used heavily in the mapping literature for latency optimizations [**?**], [**?**], [**?**], [7]. Rubik provides a `zorder` permutation operation for this purpose, as well. Like other operations in Rubik, our `zorder` operation can scale to an arbitrary number of dimensions. Rubik dynamically constructs the necessary bit filters to translate high-dimensional Z codes, and `zorder` can be called on any type of box.



```
1 Z, Y, X = 0, 1, 2
2 net = box([12,4,4])
3 net.div([3,1,1])
4 net[0,0,0].tilt(Z,X,1)
5 net[0,0,0].tilt(X,Y,1)
6 net[1,0,0].zorder()
7 net[2,0,0].zigzag(Z,X,1)
8 net[2,0,0].zigzag(X,Y,1)
```

Fig. 9: `tilt`, `zorder`, and `zigzag` operations applied to sub-boxes

Hierarchical permutation. Rubik permutations can be applied to boxes at any level of a partition tree. Fig. 9 shows a 192-task partition tree, `net`. The tree has three $4 \times 4 \times 4$ children. Here, we apply a different permutation (`tilt`, `zorder`, or `zigzag`) to each child. Accessing children is simple: Rubik's partitioning operations define a $d_0 \times d_1 \times ... \times d_n$ subspace and each element can be accessed via the corresponding subscript into this space. In the example, one could call any of these operations on `net`, and they would apply to the *entire* box instead of a subgroup.

E. Writing map files

Once Rubik has mapped a set of tasks to a network decomposition, it can write out map files suitable for use on a number of high performance computing systems. In this paper, we make use of map files for IBM Blue Gene series machines, but map files for, e.g., Cray's XT machines or Fujitsu's K supercomputer could be easily added.

F. Dimensionality-independent operations

As described above, all of Rubik's operations are dimensionality-independent. They can apply to arbitrary number of dimensions, and the same operations that are used on a 3D BG/P or Cray XT torus could be used on a 5D BG/Q torus, or on a 6-dimensional tofu network of the K supercomputer. Each operation is designed so that it can be applied in a lower-dimensional space that is easy to visualize, then scaled up in a regular fashion to higher dimensional spaces. The visualization tool used to generate the figures in this paper takes the same Python scripts as input that are used to generate the mappings. Developers can view their mapping operations as they work, and this allows them to reason intuitively about the effects of Rubik mappings in higher dimensions.

Rubik does not discover optimal network mappings automatically, nor is it intended to do so. It allows developers to leverage *a priori* knowledge of the application and architecture and target latency and/or bandwidth optimizations as the case maybe. It also provides a framework within which application developers can make reasonable mapping choices to quickly and intuitively embed their applications in higher-dimensional Cartesian spaces.

Many topology mapping tools have been developed in prior work, including ones that optimize for inter-process latency in structured codes on Cartesian networks [5] and Infiniband networks [6]. Other mapping tools allow users to change process mappings globally on 3D Cartesian networks [3], [4]. To the best of our knowledge, none of these tools scale to an arbitrary number of dimensions, nor do any of them allow application developers to describe hierarchical mappings among groups of processes representing sub-communicators.

IV. Mapping a laser-plasma interaction code

pF3D [8], [9] is a multi-physics code used to study laser plasma-interactions in experiments conducted at the National Ignition Facility (NIF) at LLNL. It is used to understand the measurements of scattered light in NIF experiments and also to predict the amount of scattering in proposed designs.

A. Communication structure

pF3D operates on a 3D process grid whose Z-direction is aligned with the laser beam. The simulation has three distinct phases and hence communication patterns: wave propagation and coupling; advecting light and solving the hydrodynamic equations. Wave propagation and coupling involves two-dimensional (2D) Fast Fourier Transforms (FFTs) in planes orthogonal to the laser, i.e. the XY-planes. More specifically, these are solved through one-dimensional line FFTs along the X and Y directions through MPI_Alltoall calls.

The advection phase consists of MPI_Send and MPI_Recv calls between consecutive planes in the Z-direction and the hydrodynamic phase consists of near-neighbor data exchange in the positive and negative X, Y and Z directions. The latter happens less frequently than the light cycles (wave propagation and advection). Given this structure, pF3D's natural domain decomposition is to have n_z planes which are split further

into n_x columns and n_y rows resulting in $n_x \times n_y \times n_z$ sub-domains. Within each plane, rows and columns are arranged into sub-communicators for the all-to-all's discussed above. For the test problem used in this paper, pF3D uses $n_x = 16$, $n_y = 8$ and n_z is calculated according to the number of processors available. In particular, for weak scaling the mesh is refined along the Z-direction, adding more XY planes and thus using more processors.

Table I lists the percentage of time spent in the top three MPI routines in pF3D when running on 2,048 and 16,384 cores of BG/P. A significant amount of the time is spent in MPI_Send (communication between adjacent XY planes) and in MPI_Alltoall over X and Y sub-communicators. The point-to-point messages are 320 and 480 KB in size whereas the all-to-all messages are 20 KB in size. Therefore, if we can map the XY planes such that we optimize the point-to-point sends between the planes while simultaneously improving the collective communication for the X and Y FFTs, we can expect performance improvements.

	2048 cores		16384 cores	
MPI call	**Total %**	**MPI %**	**Total %**	**MPI %**
Send	4.90	28.45	23.10	57.21
Alltoall	8.10	46.94	7.30	18.07
Barrier	2.78	16.10	8.13	20.15

TABLE I: Breakdown of the time spent in different MPI calls for pF3D running on 2,048 and 16,384 cores of Blue Gene/P (for the TXYZ mapping)

B. Baseline performance

To establish a baseline performance, we ran pF3D with the default mapping on BG/P. The default mapping, referred to as TXYZ, takes the MPI processes in rank order and assigns them to cores within a node first (the T dimension), then moving along the X direction of the torus, then Y, and finally the Z direction. The times spent in computation and communication are shown as a stacked bar chart in Fig. 10.

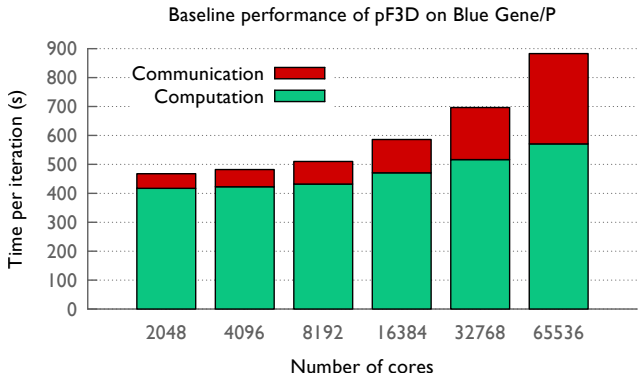


Fig. 10: Weak scaling performance of pF3D on Blue Gene/P for the default TXYZ mapping

The trend suggests that as more processors are used, communication takes up an increasing fraction of the total runtime, culminating in 35% for 65,536 cores. For applications with near-neighbor communication, the TXYZ mapping typically represents a decent default as processes that are close in MPI rank space are generally placed close on the torus network. Further, since both the application domain as well as the mapping are XYZ-ordered, while not optimal, it is a scalable mapping. However, considering the results of Section II and the large message sizes of pF3D's point-to-point communications, a slightly higher latency in exchange for more effective bandwidth may be beneficial. In the next section, we explore mappings that aim at further improving the performance.

C. Mapping on 2,048 cores

Based on the understanding of the communication structure of pF3D, one can use Rubik to generate mappings aimed at optimizing both its point-to-point and collective communication. Here, we use mappings for 2048 cores (512 nodes) as an example to explain the process of using Rubik as well as to explore why certain mappings perform better than others. At 2048 cores, the BG/P partition is a $8 \times 8 \times 8$ torus with four cores per node and the pF3D process grid is $16 \times 8 \times 16$. Following the discussion above, the goal is to place all MPI processes within a pF3D plane close on the network. The corresponding Rubik code (below) first tiles the application domain (line 2) into 16×8 planes and the torus into $8 \times 8 \times 2$ slabs (line 5) as shown below. In the rest of the paper, we refer to this basic mapping as *tiled*. Subsequently, we tilt the planes along the X (line 8) and Y (line 9) directions. These mappings are referred to as *tiltX* and *tiltXY* respectively.

```
1  app = box([16, 8, 16])
2  app.tile([16, 8, 1])
3
4  torus = box([8, 8, 8, 4])
5  torus.tile([8, 8, 2, 1])
6
7  torus.map(app)
8  torus.tilt(Z, X, 1) # tilt XY planes along X
9  torus.tilt(Z, Y, 1) # tilt XY planes along Y
10
11 torus.write_map_file(f)
```

Fig. 11 shows the reduction in the time spent in the top four MPI routines using each of the optimized mappings – XYZT, tile, tiltX and tiltXY. The XYZT mapping reduces the time spent in MPI_Sends significantly because compared to the TXYZ mapping, there is less contention for links during message exchanges between pF3D planes. In the TXYZ mapping, four cores on each node and also nodes with the same X coordinate contend for Y direction links. This is avoided in the XYZT mapping by spreading each pF3D plane to two torus planes and hence using more links (in Z) for the inter-plane communication. In the tiled mapping, four adjacent pF3D planes are placed on the four cores of each node of two adjacent XY-planes of the torus network. As shown in Fig. 11, this provides a good and scalable mapping which outperforms the XYZT mapping also. Inter-plane communication is now confined within a node to the extent possible.

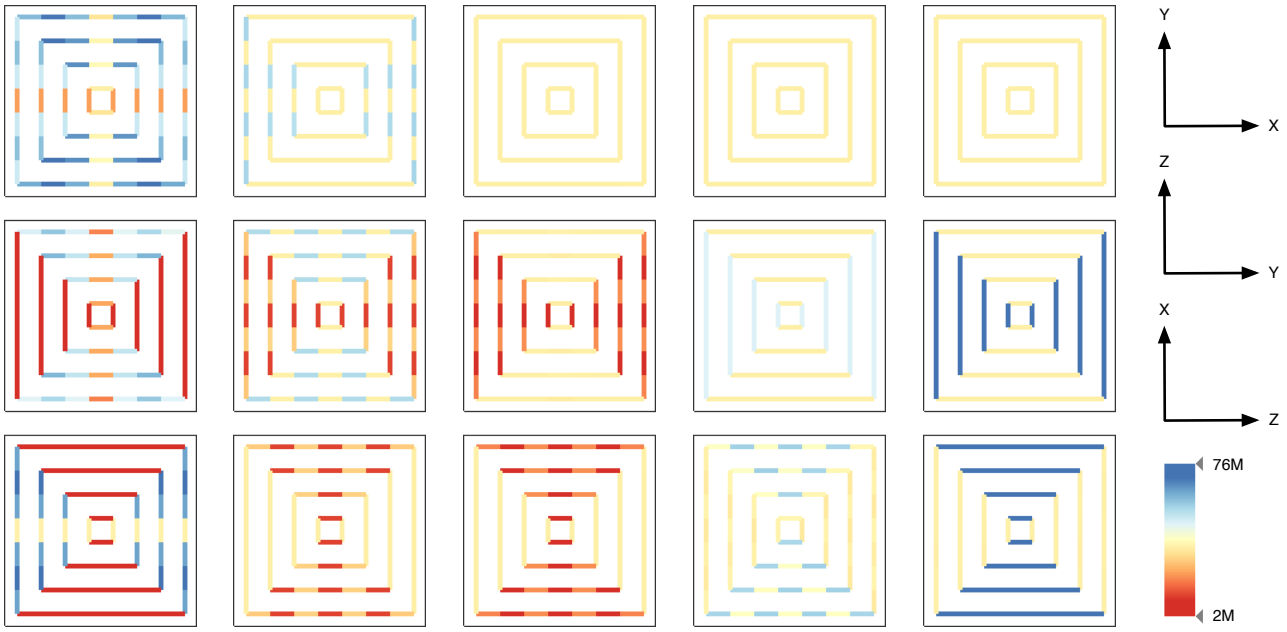


Fig. 12: Two-dimensional projections of the 3D torus network. Each column displays the network traffic along the three directions, X, Y and Z for five mappings of pF3D on 512 nodes: TXYZ, XYZT, tile, tiltX, tiltXY

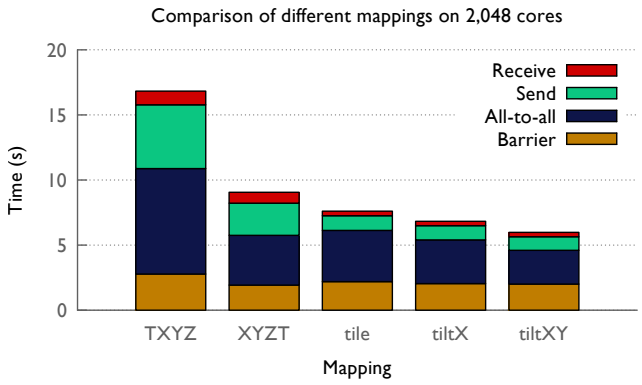


Fig. 11: Time spent in different MPI calls for five different mappings of a $16 \times 8 \times 16$ pF3D grid to 512 nodes (2,048 cores) of Blue Gene/P

In the XYZT and tiled mappings, each all-to-all within the pF3D planes uses only X or Y direction links and some Z links on the network. To increase the number of potential routes for the all-to-all sub-communicators, we therefore apply either one tilt in X (referred to as *tiltX*) or a tilt in both X and Y (referred to as *tiltXY*). Both mappings make links in the Z direction of the torus available to the all-to-alls. The twice-tilted tiltXY mapping reduces the time spent in both send-receives and all-to-alls (by optimizing the intra- and inter-plane communication). At 2048 cores, the communication is only 10% of the total execution time, hence the overall performance improvements are not as significant. The iteration time for the five mappings are 467.76, 429.22, 422.38, 420.580 and 417.095 seconds respectively.

To better understand the impact of mapping and routing on the performance, we collected network counter data for all links of the torus for the five mappings described above (see Fig. 12). We use a novel projection of the 3D network topology provided by Boxfish, an integrated performance analysis and visualization tool we have developed [**?**]. Each image of Fig. 12 shows all network links along two torus dimensions aggregated into bundles along the third dimension.

It is easy to see that the first three mappings lead to under-utilization of the Z links while the X and Y links are heavily used. Another noticeable pattern is that the first three mappings lead to uneven distribution of traffic on links in a particular direction. This is less noticeable for the tiltX mapping even though there does exist some unevenness in the Z direction. The tiltXY mapping is able to homogenize the traffic for any given direction. Even though this mapping seems to now over-utilize Z links (compared to tiltX) it improves performance.

D. Mapping on 8,192 cores

Rubik facilitates the process of generating mappings for structured communication patterns. Each mapping can be generated using a few lines of Python code and they can be scaled up easily to larger number of processors or higher dimensions. In the process of writing this paper, we generated more than two hundred mappings for pF3D using Rubik and tested all of them on BG/P. Such an extensive exploration would have been infeasible with mappings created by hand. Generating efficient mappings by hand can be a significant effort in terms of the time spent in designing the strategy, writing a program that creates the mapping and debugging and verifying that the logic is correct. Also, extending a mapping for a 3D torus to a 5D torus can be non-trivial. Typically, application developers will stop after finding the first custom mapping that is ”good enough” because it is difficult and time-consuming to generate mappings.

In Fig. 13, we present the results of twenty-five different mappings that were used on 8,192 cores. The pF3D mesh

# Cores	Application	Torus	Best mapping
2048	$16 \times 8 \times 16$	$8 \times 8 \times 8 \times 4$	torus.tile([8, 8, 2, 1]) tilt(Z, X, 1) tilt(Z, Y, 1)
4096	$16 \times 8 \times 32$	$8 \times 8 \times 16 \times 4$	torus.tile([1, 8, 16, 1]) tilt(X, Y, 1) tilt(X, Z, 1)
8192	$16 \times 8 \times 64$	$8 \times 8 \times 32 \times 4$	torus.tile([8, 8, 2, 1]) tilt(X, Y, 1)
16384	$16 \times 8 \times 128$	$8 \times 16 \times 32 \times 4$	torus.tile([8, 16, 1, 1])
32768	$16 \times 8 \times 256$	$8 \times 32 \times 32 \times 4$	torus.tile([2, 8, 8, 1]) tilt(X, Y, 1) tilt(X, Z, 1)
65536	$16 \times 8 \times 512$	$16 \times 32 \times 32 \times 4$	torus.tile([1, 8, 16, 1]) tilt(X, Y, 1) tilt(X, Z, 1)

TABLE II: Rubik operations corresponding to the best mappings (out of the ones tried) for running pF3D on different core counts

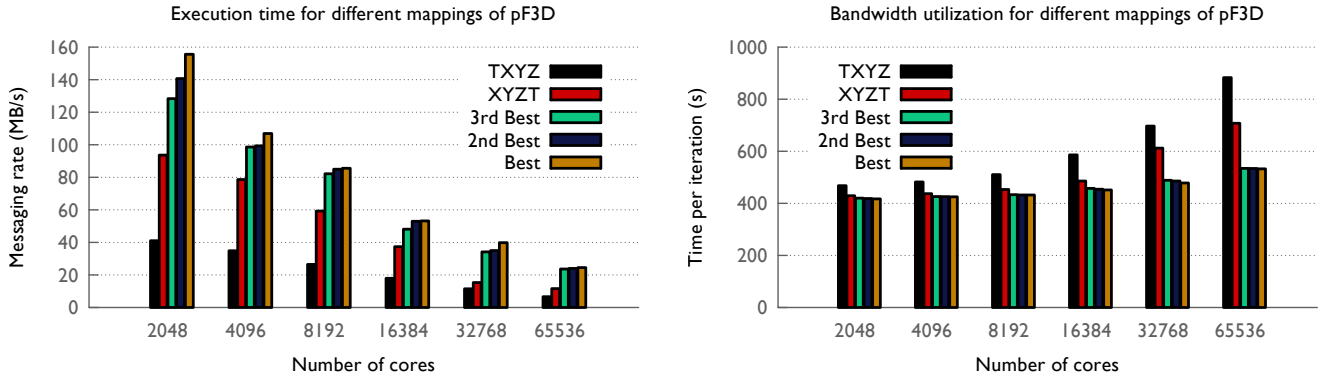


Fig. 14: Weak scaling performance of pF3D for the two default mappings as well as the three best mappings for each core count.

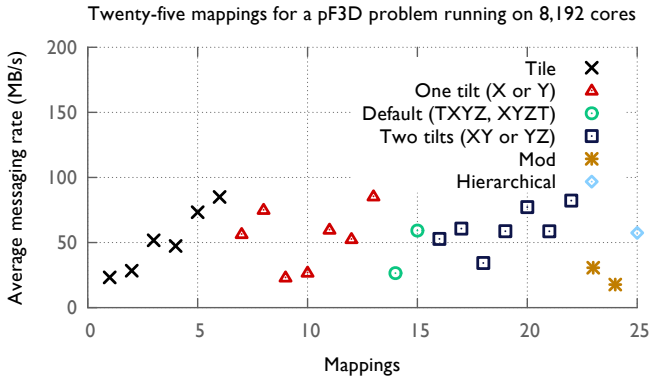


Fig. 13: The ease of generating mappings using Rubik enables us to try a lot of runs guided by our knowledge of the application

at this scale is $16 \times 8 \times 64$ and the network topology is $8 \times 8 \times 32 \times 4$. The mappings have been grouped by the different operations that Rubik supports: tile, mod, tilt, and hierarchical permutations. Hierarchical permutations refers to the use of operations on child boxes instead of the entire domain. The data plotted is the average messaging rate for one iteration of pF3D (a higher messaging rate is better as it reflects a higher bandwidth utilization). This plot shows that several Rubik operations can help achieve a higher bandwidth utilization than that for the XYZT mapping. Further, the fastest messaging rate on 8,192 cores is nearly four times higher than the slowest, with the default mappings somewhere in between these two extremes. So, a good mapping can improve performance but a bad mapping can also make it significantly worse. We discuss the scaling performance of pF3D with different mappings in the next section.

E. Weak scaling performance

On average we explored twenty-five different mappings for every core count. Interestingly, in all our tests the mapping that leads to the best performance is unique for each core count and does not perform as well at another scale. Table II lists the operations performed that give the best mapping among the ones we tried for each core count.

Fig. 14 (left) plots the messaging rates in MB/s achieved for one iteration of pF3D for the two default and three best mappings (a higher messaging rate is better). Even though the messaging rates continually decrease as we run on more cores, the top mappings are able to significantly improve the bandwidth utilization in comparison to the default one. This translates into non-trivial performance increases as shown in Fig. 14 (right). The default TXYZ and XYZT mappings do not scale well. Since this is a weak scaling problem, ideally the bars should be of the same height at different core counts. The TXYZ mapping loses 89% efficiency going from 2,048 to 65,536 cores. The best mapping loses only 28% parallel efficiency at scale and is better than the TXYZ mapping by 40% at 65,536 cores.

V. Mapping a first-principles MD code

Qbox [10] is a first-principles molecular dynamics code used to study the electronic structures of molecular systems to understand their material properties. It is a computationally intensive code and strong scaling of molecular systems leads to significant communication.

A. Communication structure

Qbox uses a 2D process grid for its data distribution and communication. 3D FFTs are done within each column of the

grid implemented as three 1D FFTs and two transposes. The other major communication phase is matrix multiplication over the entire 2D grid that involves calls to ScaLAPACK [**?**]. Table III shows a breakdown of the times spent in different MPI routines for two systems of 512 and 1728 atoms respectively at 2,048 cores. MPI_Alltoallv is a major portion of the MPI time, especially for the larger system. Also, in contrast to pF3D, a larger fraction of the communication time in Qbox is spent in collective operations.

	512 atoms		1728 atoms	
MPI call	**Total %**	**MPI %**	**Total %**	**MPI %**
Alltoallv	8.85	14.87	27.55	41.12
Recv	19.89	33.43	9.00	13.45
Reduce	1.82	3.05	8.51	12.72
Allreduce	6.04	10.14	6.78	10.12
Send	8.40	14.12	-	-

TABLE III: Breakdown of the time spent in different MPI calls for Qbox running on 2,048 cores of Blue Gene/P (for the TXYZ mapping)

B. Performance improvements

We tried mappings for two molecular systems of 512 and 1728 atoms on 2,048 cores of Blue Gene/P. Code for an example mapping that interleaves columns in Qbox on the same torus planes using the mod operation is shown below:

```
app = box([256, 8])
app.tile([256, 1])

torus = box([8, 8, 8, 4])
torus.cut([2, 1, 4, 1], [mod, div, div, div])

torus.map(app)
torus.write_map_file(f)
```

Table IV shows the performance improvements in the overall execution time of Qbox using different mappings. Similar to pF3d, the XYZT mapping improves the performance over the default. On 2,048 cores, for the smaller system, the best performance is achieved using a tiled and tilted mapping (tiltY). We first tile the Qbox 2D grid into process columns and then map each column to some planes of the torus. We then tilt the YZ planes along the Y direction. However, the same mapping performs poorly for the larger system where it is the interleaved mapping (mod) that gives the best performance.

System	**TXYZ**	**XYZT**	**tiltY**	**tiltYZ**	**mod**
512 atoms	13.01	10.39	7.81	9.63	9.63
1728 atoms	11.73	10.52	14.29	9.89	9.83

TABLE IV: Total execution time for a Qbox run of five iterations (in seconds) for different mappings

VI. Related work

The problem of mapping a task graph on to a network graph has been studied for some time [1]. There are several algorithms, heuristics and mapping frameworks to map a communication graph to an interconnect topology graph [2], [3], [4], [11], [12], [13], [14]. Most of this body of work has focused on reducing the latency of message exchanges by bringing communicating tasks closer. On torus networks with adaptive routing, it can be efficient to place communicating tasks apart to increase the potential routes between them. As far as we know, this paper presents the first systematic study and a tool for mapping sub-communicators to torus networks with an emphasis on maximizing bandwidth utilization.

Work that comes close to this paper is the development of topology aware algorithms for implementing collectives in MPI implementations. Faraj *et al.* present rectangular algorithms that exploit the cuboidal nature of 3D tori to optimize broadcasts and all-reduce [5]. Kandalla *et al.* do a similar study with scatter and gather collectives on large-scale Infiniband networks (fat-tree topology) [6]. Solomonik *et al.* present the mapping of 2D matrix multiplication and LU factorization to exploit the existence of rectangular collectives on Blue Gene machines [15]. In contrast, our paper breaks the regularity of the default mappings of structured communication patterns with operations such as tilt and zigzag with a hope of achieving higher bandwidth.

VII. Lessons learned

We presented benchmarking results for all-to-all and broadcast operations on Blue Gene/P and Blue Gene/Q that suggest that making more links available to the system software and hardware to route packets can optimize performance. This is especially true for large messages where the spare links and extra bandwidth can be useful in reducing congestion. Mappings that try to achieve this can lead to longer latencies and increased link sharing. Hence, we need to consider the trade-offs between increasing bandwidth utilization and longer latencies and increased congestion carefully.

The mappings for pF3D and Qbox that we present in the paper support our initial hypothesis that the following mapping techniques can improve performance:

- Placing communicating tasks on the corners of a plane/ box or mesh of higher dimension instead of a line.
- Increasing the distance between communicating pairs to create more spare links
- Using wraparound torus links as additional routes.

Even though topology aware task mapping has been studied extensively, little attention has been paid to systematically placing closely-linked groups of tasks. Applications that perform collective operations over sub-communicators fall under this category. We have developed *Rubik*, a tool that provides a simple and intuitive interface to create a wide variety of mappings for structured communication patterns. Rubik supports a number of elementary operations such as splits, tilts, or shifts, that can be combined into a large number of unique patterns. Each mapping can be applied to disjoint groups of processes involved in collectives to increase the effective bandwidth.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-556491). Accordingly, the U.S. government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. government purposes.

Neither the U.S. government nor Lawrence Livermore National Security, LLC (LLNS), nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, or process disclosed, or represents that its use would not infringe privately owned rights. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. government or LLNS, and shall not be used for advertising or product endorsement purposes.

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] Shahid H. Bokhari, "On the Mapping Problem," *IEEE Trans. Computers*, vol. 30, no. 3, pp. 207–214, 1981.

[2] Soo-Young Lee and J. K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 433–442, 1987.

[3] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, and R. Walkup, "Optimizing task layout on the Blue Gene/L supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 489–500, 2005.

[4] H. Yu, I.-H. Chung, and J. Moreira, "Topology mapping for Blue Gene/L supercomputer," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 116.

[5] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations," in *Proceedings of the 17th IEEE Symposium on High Performance Interconnects*, ser. HOTI '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 63–72.

[6] K. C. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda, "Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather," in *International Parallel and Distributed Processing Symposium*, 2010, pp. 1–8.

[7] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, http://hdl.handle.net/2142/16578.

[8] C. H. Still, R. L. Berger, A. B. Langdon, D. E. Hinkel, L. J. Suter, and E. A. Williams, "Filamentation and forward brillouin scatter of entire smoothed and aberrated laser beams," *Physics of Plasmas*, vol. 7, no. 5, p. 2023, 2000.

[9] R. L. Berger, B. F. Lasinski, A. B. Langdon, T. B. Kaiser, B. B. Afeyan, B. I. Cohen, C. H. Still, and E. A. Williams, "Influence of spatial and temporal laser beam smoothing on stimulated brillouin scattering in filamentary laser light," *Phys. Rev. Lett.*, vol. 75, no. 6, pp. 1078–1081, Aug 1995.

[10] F. Gygi, "Qbox open source code project," http://eslab.ucdavis.edu/, University of California, Davis.

[11] Francine Berman and Lawrence Snyder, "On mapping parallel algorithms into parallel architectures," *Journal of Parallel and Distributed Computing*, vol. 4, no. 5, pp. 439–458, 1987.

[12] S. Wayne Bollinger and Scott F. Midkiff, "Processor and Link Assignment in Multicomputers Using Simulated Annealing," in *ICPP (1)*, 1988, pp. 1–7.

[13] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1408–1424, 1987.

[14] George Almasi and Siddhartha Chatterjee and Alan Gara and John Gunnels and Manish Gupta and Amy Henning and Jose E. Moreira and Bob Walkup, "Unlocking the Performance of the Blue Gene/L Supercomputer," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 57.

[15] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011.