2014 IEEE 28th International Parallel & Distributed Processing Symposium

Exploiting Geometric Partitioning in Task Mapping for Parallel Computers

Mehmet Deveci*, Sivasankaran Rajamanickam†, Vitus J. Leung†, Kevin Pedretti†,
Stephen L. Olivier†, David P. Bunde‡, Ümit V. Çatalyürek*, and Karen Devine†
*The Ohio State University, Columbus, Ohio
† Sandia National Laboratories, Albuquerque, New Mexico
‡Knox College, Galesburg, IL

***Abstract*—We present a new method for mapping applications' MPI tasks to cores of a parallel computer such that communication and execution time are reduced. We consider the case of sparse node allocation within a parallel machine, where the nodes assigned to a job are not necessarily located within a contiguous block nor within close proximity to each other in the network. The goal is to assign tasks to cores so that interdependent tasks are performed by "nearby" cores, thus lowering the distance messages must travel, the amount of congestion in the network, and the overall cost of communication. Our new method applies a geometric partitioning algorithm to both the tasks and the processors, and assigns task parts to the corresponding processor parts. We show that, for the structured finite difference mini-app MiniGhost, our mapping method reduced execution time 34% on average on 65,536 cores of a Cray XE6. In a molecular dynamics mini-app, MiniMD, our mapping method reduced communication time by 26% on average on 6144 cores. We also compare our mapping with graph-based mappings from the LibTopoMap library and show that our mappings reduced the communication time on average by 15% in MiniGhost and 10% in MiniMD.**

I. Introduction

Task mapping — the assignment of a parallel application's tasks to the processors of a parallel computer — is increasingly important as the number of processors in new supercomputers grows from $O(100K)$ to $O(1M)$ and beyond. With large-diameter networks in these supercomputers and many users submitting jobs of various sizes, processor allocations (the sets of processors assigned by a job scheduler to parallel jobs) can become more sparse and be spread further across the entire network. As a result, communication messages can travel long routes in the network and network links may become congested by heavy traffic, which makes maintaining scalability in large-scale machines difficult. These effects can be lessened through the use of topology-aware task mapping. Recent experiments have shown that task mapping can significantly impact performance of parallel applications (e.g., [1], [4], [8], [12], [21]); one application exhibited a 1.64X speedup due to improved mapping [16].

An effective mapping of tasks to processors considers both the tasks' communication pattern and the physical network topology to reduce application communication cost. We propose a new task mapping strategy that uses geometric information to represent application tasks and compute resources. We define metrics based on this geometric information to represent the cost of communication between tasks, and use these metrics to evaluate and select effective mappings.

Much research has focused on mapping tasks to block-based allocations, such as those on IBM's BlueGene systems (e.g., [3], [8], [16], [28]). Our focus is on non-contiguous (i.e., sparse) allocations, where nodes from any portion of the machine can be assigned to a job without regard to the allocation's shape or locality. Such allocations are used in many parallel systems (e.g., Cray, clusters). Mapping strategies developed for general allocations can be used automatically for the more restricted case of block allocations.

Most previous non-contiguous approaches have represented tasks' communication patterns and network topologies as graphs; graph algorithms were then applied to find good mappings. Finding optimal topology mappings has been shown to be NP-Complete [19], so heuristics are often used to reduce complexity (e.g., [7], [9], [10], [13], [14], [22]). We, instead, use an inexpensive geometric partitioning algorithm to reorder tasks and processors based on their geometric locality, and use the reordering to map tasks that are "close" to each other geometrically to processors that are "close" to each other in the mesh or torus. Initial experimentation with geometric approaches proved promising [23]; our work improves the geometric strategies, compares them with more sophisticated graph-based methods, demonstrates them for additional applications, and provides software suitable for use in parallel applications.

General-purpose, open-source graph-based mapping algorithms are available. The LibTopoMap library [19] requires as input a task-communication graph describing the amount of communication between tasks, as well as static files describing the network topology. It uses the ParMETIS graph partitioner [20] to divide tasks into n parts, where n is the number of nodes in the allocation, and then applies a graph algorithm (Greedy, Recursive Bisection, Reverse Cuthill-McKee) to map the parts to nodes. The JOSTLE [27] and Scotch [25] libraries combine mapping with load balancing by using recursive bisection of both network-topology and application-data graphs to partition data and map the resulting parts to processors. Like these libraries, our approach is designed for general-purpose use in applications and is available in the Zoltan2 [11] library.

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.15

IEEE computer society

The main contributions of this paper follow.

• We present a new geometric algorithm for task mapping in non-contiguous processor allocations (Section IV).

• We present metrics for evaluating mappings in mesh- and torus-based networks (Sections II and III), and validate these metrics using performance counter information from the Cray Gemini routers (Section V).

• We demonstrate our algorithm in two proxy applications on up to 64K cores, and assess the quality of our mappings with respect to application communication cost and execution time (Section V).

• We compare our geometric mappings to applications' default mappings, application-specific optimizations, and the LibTopoMap graph-based mapping library, showing that our geometric mappings reduce both communication time and communication metrics for the target applications relative to other methods (Section V).

II. Mapping Metrics

We use two primary metrics to represent the network communication: *average hop count* (the average length of paths taken by messages) and *maximum congestion* (the maximum number of messages sent across communication links). In this paper, we assume static routing of messages. Also, we assume that each message is transferred over a single path (i.e., messages are not split and sent through multiple paths), and that all links have the same capacity.

Let $G_t(V_t, E_t)$ be the graph representing task communication, where V_t is the set of tasks, and E_t is the set of edges that represent communication between tasks. If $t_1, t_2 \in V_t$, edge $(t_1, t_2) \in E_t$ if and only if tasks t_1 and t_2 communicate. In the same way, let $G_n(V_n, E_n)$ be the graph representing the network topology. V_n is the set of nodes, and E_n is the set of edges that represent the physical communication links between nodes. If $n_1, n_2 \in V_n$, edge $(n_1, n_2) \in E_n$ if and only if nodes n_1 and n_2 have a connecting link between them. Let Γ be a function for the assignment of tasks to nodes. That is, $n_1 = \Gamma(t_1)$, if t_1 is assigned to a core in node n_1. Using these assumptions, we define *dilation* as follows:

$$dilation(t_1, t_2) = SPL(\Gamma(t_1), \Gamma(t_2), G_n), \quad (1)$$

where SPL is a function that returns the shortest path length between two nodes. The total dilation is

$$Dilation(\Gamma) = \sum_{(t_1,t_2)\in E_t} dilation(t_1, t_2) \quad (2)$$

$Dilation$ is related to the average number of edges traversed by each message (*average hop count*):

$$AverageHopCount(\Gamma) = Dilation(\Gamma)/|E_t| \quad (3)$$

In this paper, we measure the average hop count, and use the terms "hop count" and "average hop count" interchangeably.

The congestion metric is defined as follows:

$$Congestion(e) = \sum_{(t_1,t_2)\in E_t} inSP(e, \Gamma(t_1), \Gamma(t_2), G_n), \quad (4)$$

where $inSP$ returns 1 if and only if e is in the shortest path between $\Gamma(t_1)$ and $\Gamma(t_2)$. Otherwise, it returns 0. Therefore, the congestion on a link is the number of messages that go through it. Then the maximum congestion is

$$MaxCongestion(\Gamma) = \max_{e\in E_n}\{Congestion(e)\} \quad (5)$$

Maximum congestion represents the maximum number of messages that go through any link. We refer to maximum congestion as "congestion" in the rest of this paper. Since communication is a real time process and is affected by many outside factors (e.g., network traffic and overhead from competing jobs), theoretical metrics can only approximate actual communication time. Experiments in Section V validate the efficacy of our two metrics.

III. Targeted Computing Environment

Mesh- or torus-based networks are common in parallel computers; for example, Cray's XT and XE computers and IBM's BlueGene computers have torus-based networks. In the Cray XE6's 3D torus, for example, each Gemini router connects to six neighboring routers, two each in the x, y and z dimensions (see Figure 1). Messages' routes between nodes can be represented as a path of "hops" along network links in x, y and z. Differences in bandwidth along the various dimensions can exist depending on the physical connections (e.g., backplane, mezzanine, cable) used in each dimension. However, in this work, we assume uniform bandwidth in all dimensions; accounting for varying bandwidth is reserved for future work.

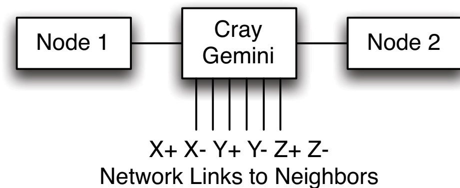


Figure 1. Layout of a Cray Gemini router

In mesh- and torus-based systems, "coordinates" of the routers within the network are often made available by calls to a system library. A router with 3D coordinates (i, j, k) can communicate with a router with coordinates $(i + 1, j + 1, k + 1)$ via a three-hop path, with one hop in each of the x, y and z dimensions. The torus provides wrap-around, so that messages take the shortest path (i.e., proceed in the positive or negative direction) along each dimension.

In the Cray XE6, these coordinates are available from Resiliency Communication Agent (RCA) tool through calls to `rca_get_meshcoord`. Each MPI process can obtain the coordinates of the router to which its compute node is attached. Our task-mapping methods then use dilation (Eqn. 1) between routers within the network as an approximation of communication cost between MPI processes.

Each router in a mesh/torus network is typically connected to one or more multicore compute nodes. In the Cray XE6, for example, each router connects two nodes (hosts). The two Cray platforms we used (DOE's Cielo and NERSC's Hopper) have 16 and 24 cores per node, respectively. Parallel applications can use from one MPI process per node (with threading providing parallelism within the node) to one MPI process per core (with shared-memory message passing within the node). In the latter case, co-locating interdependent MPI processes within a node reduces communication over the network, and, thus, reduces execution time. Our task-mapping experiments address this case, but since our methods address the mapping of MPI processes to compute resources, they can be applied without loss of generality to the multithreaded case as well.

One practical difference between Cray XE and IBM BlueGene systems is the way compute nodes are allocated to jobs. In IBM systems, jobs are given a contiguous "partition" or block of nodes within the network; each dimension of this block must be a power of two. In contrast, on Cray systems, jobs are given non-contiguous node allocations of any size requested by the user. Available nodes are selected according to a space-filling curve algorithm in the ALPS scheduler [2]. Thus, while the scheduler attempts to assign nearby nodes to jobs, no guarantees of locality are provided. As a result, task-mapping algorithms for Cray systems need to accommodate non-block, non-contiguous allocations. While our methods are designed for non-contiguous allocations, they could be applied to contiguous allocations as well.

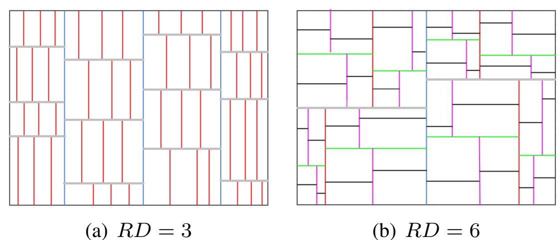
Ideally, closer proximity of router coordinates results in lower communication costs between two nodes. However, congestion caused by communication patterns within an application and by other applications on the system can influence application behavior. For our mapping experiments, we access the Cray XE6 Gemini tile counters to obtain information about input and output message stalls, a measure of congestion within the network [24]. With these counters, we show how improved task mapping reduces network congestion for the application and validate our computed congestion metric (Eqn. 5).

IV. Geometric Task Mapping

Our proposed topology-aware mapping algorithm uses the router coordinates to represent the network topology of the machine. The cost of communication between pairs of cores is approximated by the dilation (Eqn. 1) of their routers' coordinates. Thus, the machine topology is described only by the cores' coordinates, rather than a topology graph in which bandwidth information between every pair of cores must be specified. Each of the application's MPI processes is also represented by a coordinate, corresponding to either the center of the process' application domain or the average coordinate of its application data. For example, in a structured grid-based finite difference application, the center of an MPI process' subgrid can be used as its coordinate. Our algorithm uses a geometric partitioning algorithm to consistently reorder both the MPI processes and the allocated cores; this reordering is used to construct the mapping. In this section, we provide details of our mapping algorithm. We use the term "machine coordinates" to refer to the router coordinates associated with each core, and "task coordinates" to refer to the centroid or averaged coordinates provided by the application's MPI processes.

A. Multi-Jagged (MJ) Algorithm for Geometric Partitioning

Our proposed task mapping algorithm uses a geometric partitioning algorithm, the Multi-dimensional Jagged algorithm (MJ) [15] of the Zoltan2 Toolkit [11], to partition task and machine coordinates. The MJ algorithm partitions a set of coordinates into a desired number of parts (P) in a given number of steps called the *recursion depth* (RD). During each recursion, one-dimensional partitioning is done along a dimension; the dimension is alternated at each recursion. Therefore, MJ is a generalization of the Recursive Coordinate Bisection (RCB) algorithm [6] in which the MJ algorithm has ability to do multisections instead of bisections. Although our implementation of MJ can partition into any number of parts P, we simplify our explanation here by assuming P can be written as $P = \prod_{i=1}^{RD} P_i$. In the first level, MJ partitions the domain into P_1 parts using $P_1 - 1$ cuts in one direction. In the next level, each of the P_1 parts is partitioned separately into P_2 parts using cuts in an orthogonal direction. This recursion continues in each level. Figure 2 shows two 64-way partitions using MJ with $RD = 3$, $P = 4 \times 4 \times 4$ (left), and $RD = 6$, $P = 2 \times 2 \times 2 \times 2 \times 2 \times 2$ (right). When $RD = \lceil \log_2 P \rceil$, MJ is equivalent to RCB (as in Figure 2(b)).



(a) $RD = 3$ (b) $RD = 6$

Figure 2. Partitioning into 64 parts using MJ with different recursion depths. Cutlines in the same level of recursion share the same color.

MJ's complexity depends on P, RD, the number of points n, and the average number of iterations it needed to compute

cutline locations. During partitioning on level i, each point is compared to $\log_2 P_i$ cut lines (using binary search). Thus, MJ's complexity is $O(n \times it \times \sum_{i=1}^{RD} \log_2 P_i)$. When MJ is used as RCB, its complexity is $O(n \times it \times \log_2 P)$.

B. Using MJ for Task Mapping

Although MJ is proposed as a parallel (MPI+OpenMP) algorithm [15], we use it as a sequential algorithm in this context. The size of the partitioning problem is proportional to the number of processors. Since current supercomputers typically have $O(100K)$ processors, the partitioning algorithm would be communication bound if done in parallel; little or no speedup would be obtained by parallelizing this process. Instead, each processor calculates the same mapping independently. A reduceAll operation is performed at the beginning of task mapping to provide all machine and task coordinates to every processor. Then every processor performs the sequential mapping operation and obtains the exact same mapping. We describe in Section IV-C how we exploit parallelism to improve the quality of the mapping with minimal additional cost.

The proposed mapping algorithm is defined as follows: Given $tdim$-dimensional coordinates of the tasks (tc), and $pdim$-dimensional coordinates of the cores (pc), together with the number of tasks (tn) and cores (pn), the algorithm returns a mapping from cores to tasks ($p2t$) (and/or tasks to cores $t2p$). Algorithm 1 gives the description of the task mapping algorithm.

Algorithm 1 Task Mapping Algorithm using MJ

Require: $tc, tdim, tn, pc, pdim, pn, RD$
$minDim \leftarrow \min(tdim,\ pdim)$
$usedNumProcs \leftarrow numParts \leftarrow \min(tn,\ pn)$
if $pn > tn$ **then**
  $procPerm \leftarrow getClosestSubset(pc,\ pdim,\ pn,\ tn)$
else
  $procPerm \leftarrow range(0,\ pn)$
end if
$taskPerm \leftarrow range(0,\ tn)$
$taskParts \leftarrow MJ(tc,\ minDim,\ tn,\ taskPerm,\ numParts,\ RD)$
$procParts \leftarrow MJ(pc,\ minDim,\ usedNumProcs,\ procPerm,\ numParts,\ RD)$
$p2t, t2p \leftarrow getMappingArrays(taskParts,\ procParts,\ taskPerm,\ procPerm,\ tn,\ pn)$

MJ's main purpose in Algorithm 1 is to consistently number the cores and tasks. Function MJ partitions the task and cores into $usedNumProcs$ parts, and assigns a part number to each core and task. Cores and tasks that share the same part number are then mapped to each other by GETMAPPINGARRAYS, and the resulting mappings are stored in $p2t$ and $t2p$.

Since the tasks and cores are partitioned separately, Algorithm 1 ensures consistent part numbering among both MJ calls. First, the minimum dimension is chosen between the tasks and cores. For example, if $pdim = 3$ while $tdim = 2$, one of the cores' coordinates is ignored to ensure that the geometric partitioner follows the same order in both of the partitioning operations. Next, the algorithm can follow different paths depending on the number of coordinates of tasks and cores. There are three possibilities at this step:

1) $tn = pn$**:** A one-to-one mapping between cores and tasks exists. For task t assigned to core p, $t = p2t[p]$ and $p = t2p[t]$.

2) $tn > pn$**:** When there are more tasks than cores, a core is assigned multiple tasks. Both cores and tasks are partitioned into pn parts, with multiple tasks in the each part. The mapping results will be $t \in p2t[p]$ and $p = t2p[t]$.

3) $tn < pn$**:** When there are more cores than tasks, the algorithm does not split a task among multiple cores. Instead, during a preprocessing step, it chooses a subset of tn cores. Then, mapping is performed within this subset as if $tn = pn$. Some cores will be idle, as they are not assigned any tasks. Our implementation uses a modified K-means clustering algorithm [17] to choose the closest subset of cores within the allocation; in this paper, however, this special case is not considered.

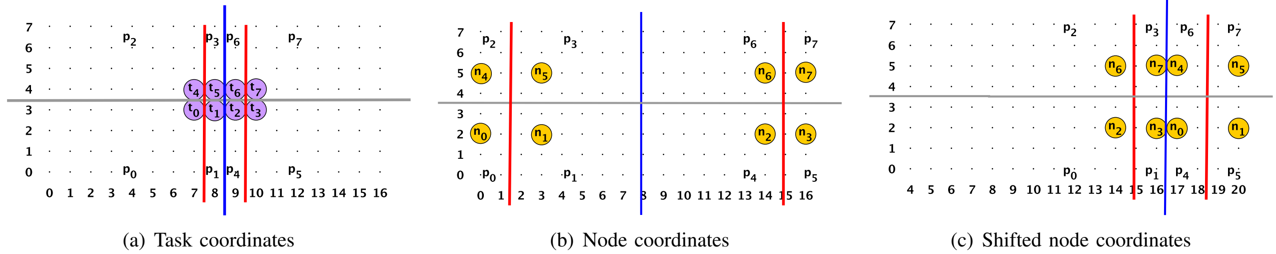
The complexity of Algorithm 1 is dominated by the calls to MJ, since $getMappingArrays$ runs in linear time with respect to tn and pn. Thus, when MJ is used as RCB and $tn = pn$, the overall complexity of the mapping algorithm is $O(tn \times it \times \log_2(tn))$.

C. Improving the quality of the mapping

The ability of our mapping strategy to reduce communication costs depends on the results of the MJ partitioner. In this section, we describe several ways that we can improve the quality of the mapping by modifying the input to MJ. These improvements are computed with very little extra expense, as they are computed in parallel across sets of processors.

Shifting the machine coordinates: The first improvement involves considering the 3D torus interconnection present in many supercomputer networks. Torus networks provide wrap-around communication links in each network dimension that are not reflected in the machine coordinates. Thus, since MJ is not aware of connectivity information, MJ considers nodes at edges of the network coordinates to be far apart, even though there is a one-hop path between them.

In our proposed task-mapping method, we transform the coordinates to account for wrap-around in each dimension. Our shifting strategy applies a one-dimensional operation to each dimension independently. First, we find the shift position – the largest gap in the node coordinates. Then, assuming the largest gap is greater than one, we transform the machine coordinates on one side of the shift position



(a) Task coordinates (b) Node coordinates (c) Shifted node coordinates

Figure 3. An example showing the benefit of shifting node coordinates in torus networks. The numbers of nodes and tasks are equal. Tasks and nodes sharing the same number are mapped to each other (3(a)). Assuming nearest-neighbor communication, the unshifted mapping (3(b)) has average hop count of 3.66 and 3 in the x and y directions, respectively; note that messages between n_1 and n_2 and between n_5 and n_6 require six hops due to wrap-around links. With the node partition obtained after shifting around the wrap-around links (3(c)), the mapping has average hop count of 2 and 3 in x and y.

by adding to them the maximum machine coordinate in that dimension. Ties in the largest gaps are broken using the number of nodes that are on either side of the shift positions. Even though the list of machine coordinates is not sorted for any dimension, gap detection can still be performed in O(pn) time by using a counting sort algorithm.

Figure 3 shows an example of mapping eight tasks onto eight nodes in a 17x8 2D torus topology. In Figure 3(b), the maximum gap along the x dimension is found between n_1 ($x = 3$) and n_2 ($x = 14$) (also between n_5 and n_6). The heuristic shifts the coordinates of all nodes that have $x \leq 3$ by adding the maximum x coordinate (17) to the x coordinates of these nodes. Figure 3(c) shows the updated machine coordinates after this shift operation. Assuming nearest neighbor communication for the tasks in Figure 3(a), the mapping in Figure 3(b) results in average hop counts of 3.66 and 3 in the x and y dimensions, respectively. After shifting, the mapping in Figure 3(c) achieves average hop counts of 2 and 3.

Rotating the machine and task coordinates: The quality of the mapping also depends on the order of the dimensions to which the partitioning is applied (e.g., first partition in x, then y, then z). For example, Figure 4 shows how the quality of the mapping can change by choosing a different order of dimensions in partitioning.

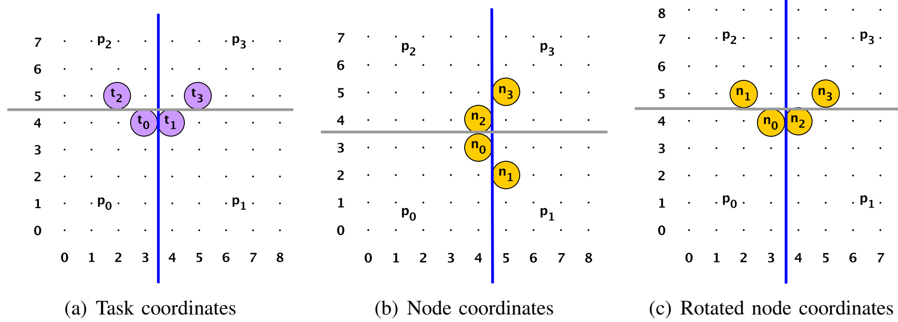
It is difficult to predict which dimension ordering for partitioning will provide the best mapping quality. One could choose a permutation of the dimensions based on the aspect ratios of the machine and task coordinates. The permutation that makes the aspect ratios along dimensions closest can be chosen as the best permutation. However, as our experiments will show, this greedy method fails to find the best permutation in most of the mappings. To overcome this issue, we use a speculative method. Recall that, in Section IV-B, we described sequential task mapping in which every process computed the same mapping. Since there are pn processes, we instead calculate different mappings with different rotations in each process. Then, given the communication pattern of the tasks, each mapping's hop count is computed, and the one with the lowest hop count is chosen. This comparison requires one extra reduceAll and broadcast operation. If the dimensions of the tasks and the machine are $tdim$ and $pdim$, there are $rp = (tdim)! \times (pdim)!$ different rotations. For a 3D torus with 3D task coordinates, $rp = 3! \times 3! = 36$. We group processes into sets of size 36, in which each process calculates a mapping using a different rotation. Each process calculates the quality of its own mapping. Then within each group, the best quality mapping is determined, and is broadcast to the group. When the number of processes is not divisible by rp, the remaining processes are distributed among groups so that as many rotations as possible are calculated within each group.

Similarly, reflections of the node or task coordinates along coordinate axes could be done. The total number of different reflections is 2^{maxDim}, where $maxDim = max(tdim, pdim)$. Again, the processes can be grouped such that each group has 2^{maxDim} processes that each calculate a different mapping. Combined with the rotation operation, the total number of different solutions becomes $2^{maxDim} \times tdim! \times pdim!$, which is 288 for the usual case of a 3D torus with 3D task coordinates. Our implementation includes the rotations described above, but does not yet include reflections.

V. Experiments

We tested our geometric mapping methods in two proxy applications: MiniGhost [5] and MiniMD [18]. For each application, we ran weak scaling experiments to evaluate the effect of mapping on communication and execution time. We compared our geometric method with the applications' default task layout and with the graph-based task mapping library LibTopoMap [19]. For MiniGhost, we also compare with an application-specific grouping of tasks for multicore nodes. These mapping methods are described below and summarized in Table I.

- **None:** The application's default mapping of tasks to ranks: task i is performed by rank i.
- **Multicore Grouping (Group):** Tasks reordered into 16-task blocks, with 2x2x4 tasks per block. A block is then assigned to cores within the same node, so that frequently



(a) Task coordinates (b) Node coordinates (c) Rotated node coordinates

Figure 4. An example showing the benefit of rotating the node orientation. Assuming communication is required between only tasks 0 and 2, the unrotated mapping (4(b)) has average hop count of 1 and 1 in the x and y directions, respectively. In the rotated node-partition (4(c)), partitioning is performed in the y dimension first, and then in the x dimension. The mapping obtained after rotation has average hop count of 1 and 0 in x and y.

Method	Abbreviation	Description
No mapping	None	Task i performed by core i.
Multicore Grouping	Group	16-task blocks; 2x2x4 tasks per block.
Geometric	Geom	Geometric with one rotation w.r.t. aspect ratio
Geometric + Rotations	Geom+R	Geometric with 36 rotations
Geometric + Rotations + Coordinate Shift	Geom+R+S	Geometric with 36 rotations and torus-aware shifting
LibTopoMap	TopoMap	Graph-based mapping [19]

Table I
MAPPING METHODS USED IN EXPERIMENTS

communicating tasks are within the same node. However, this reordering does not account for inter-node communication, since it does not use any information about the position of the nodes in the network. Group exploits application-specific knowledge about the finite-difference grid; thus it is available only in MiniGhost.

• **Geometric (Geom):** Geometric mapping [23] with recursion depth $RD = \lceil \log_2 P \rceil$ (i.e., performing bisection at each level). A single rotation is determined at the beginning of the algorithm by using the aspect ratios of the task and machine coordinates. Ties among coordinates along a dimension are broken arbitrarily; if several x coordinates lie along a cut, the choice of coordinates that go to the left of the cut or to the right is arbitrary.

• **Geometric with Rotations (Geom+R):** Geometric (MJ) mapping with recursion depth $RD = \lceil \log_2 P \rceil$; it calculates 36 different solutions according to 36 different rotations, and chooses the one with the lowest hop count metric. Ties among machine coordinates are broken first by coordinates in other dimensions, and then by MPI ranks so that coordinates with lower MPI ranks go to one side and those with higher ranks to the other.

• **Geometric with Rotations and Coordinate Shift (Geom+R+S):** Geom+R mapping with coordinate shifting done as preprocessing to account for torus networks.

• **LibTopoMap (TopoMap):** Graph-based mapping strategies available in the open-source library LibTopoMap [19]. For each experiment, we use the result with the lowest application execution time among LibTopoMap methods Greedy, Recursive Bisection, and Reverse Cuthill-McKee. If LibTopoMap does not find a mapping that is better than the input mapping, it returns the input mapping.

We ran all experiments on the DOE Cielo Cray XE6 at Los Alamos National Laboratory, and the Hopper Cray XE6 at NERSC. On both platforms, we used gcc 4.7.2 compilers and Cray's MPICH2 implementation. Our geometric mapping techniques are implemented in the Zoltan2 library [11].

A. *Mapping in a finite difference application*

We compared the effect of our mapping method in MiniGhost [5], a finite-difference proxy application that implements a finite difference stencil and explicit time-stepping scheme across a three-dimensional uniform grid. Using a seven-point stencil, each task communicates with two neighbors along each dimension; tasks along geometry boundaries communicate with only their neighbors interior to the boundary (i.e., boundary conditions are non-periodic). Each task is assigned a subgrid of the 3D grid based on its task number. The numbers of tasks in each dimension pn_x, pn_y, pn_z (with $pn_x \times pn_y \times pn_z = pn$) are specified by the user. Subgrids of the 3D grid are assigned to tasks by sweeping first in the x direction, then the y direction, then the z direction. Thus, task i shares subgrid boundaries (and, thus, requires communication) with tasks $i+1$ and $i-1$ to its east and west, respectively; with tasks $i+pn_x$ and $i-pn_x$ to its north and south; and with tasks $i+(pn_x)(pn_y)$ and $i-(pn_x)(pn_y)$ to its front and back. In the default MiniGhost configuration, task i is performed by rank i.

As shown in [4], the execution time of MiniGhost with its default mapping does not scale well in weak scaling tests. Our goal is to improve scalability by mapping tasks onto processors so that tasks that share boundaries are placed

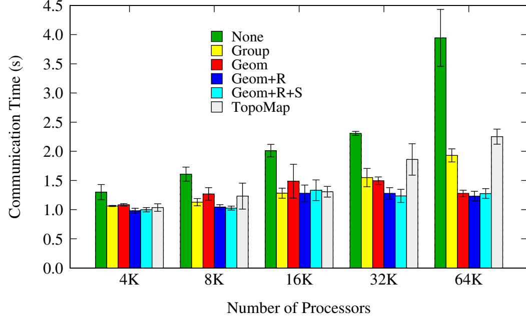
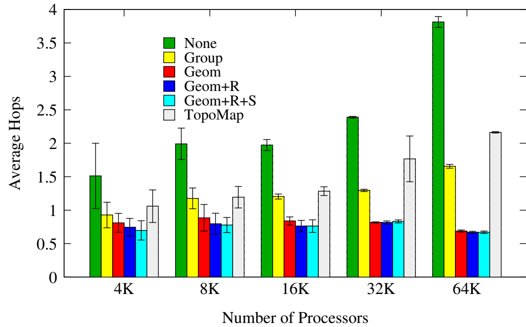
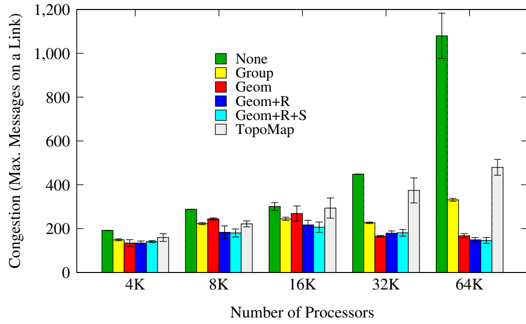


Figure 5. Maximum communication time in weak scaling experiments with MiniGhost



(a) Average Hop Count



(b) Maximum Congestion

Figure 6. Average Hop Count (a) and Maximum Congestion (b) for weak scaling experiments with MiniGhost

"near" each other in the allocation. We ran weak-scaling experiments with 4096–65,536 processors (256–4096 nodes) of Cielo. Each task owned a $60 \times 60 \times 60$-cell subgrid; we ran the simulation for 20 timesteps with 40 variables per grid point. For each experiment, we obtained a node allocation of the requested size, and ran all mapping methods within that allocation. We repeated each experiment five times with different allocations, and averaged the results across the five instances; error bars in figures show the standard deviation from the averages.

Figure 5 shows the maximum communication time (across processors) for weak scaling experiments with MiniGhost. With MiniGhost's default mapping (None), communication time increases dramatically as the number of processors is increased. MiniGhost's Group method controls the growth in communication costs, but consistent with results in [4], costs increase at the highest processor counts. Geometric methods Geom+R and Geom+R+S provide the lowest communication costs, and, as desired for weak scaling, the communication costs remain nearly constant as the number of processors increases. Above 16K processors, denser node allocations ($> 20\%$ of the total machine) help the geometric methods further reduce average hop count and congestion. TopoMap also reduced communication relative to MiniGhost's default mapping, but was unable to reduce communication as much as Group, Geom+R and Geom+R+S. Table II shows the total execution time and the percentage of total execution time that was spent in communication. These results show the importance of topology-aware mapping in general, as all mapping strategies maintained scalable communication better than the default layout. Geom+R and Geom+R+S provided the most consistent performance, maintaining communication as 13–14% of total execution time.

Figure 6 shows the calculated quality metrics: average $x+y+z$ hops (Eqn. 3) and maximum congestion (Eqn. 5). As the number of processors increases, the average hop count, congestion and communication cost all follow the same upward trend for the default MiniGhost mapping. Since Group does not account for inter-node communication, its average hop count increases with the number of nodes. TopoMap's average hop count and max congestion also increase as we scale to larger number of processors. Average hop count for the geometric mappings Geom, Geom+R and Geom+R+S, however, remains nearly unchanged as we use more processors, suggesting greater scalability using the geometric mappings. Congestion is also low for the geometric mappings, resulting in lower communication cost.

Among the geometric methods, Geom+R and Geom+R+S provided the best mappings. The benefit of coordinate shifting in Geom+R+S is small for these experiments. Geom+R+S usually produces hop counts no greater than Geom+R, and when the allocation allows, can reduce the hop count relative to Geom+R. Thus, the averaged values are very similar. Geom+R+S has slightly greater variation in hop counts for individual experiments, since for some allocations, it can further reduce the hop count via shifting.

The effect of communication between two nodes connected by the same Gemini router is not accounted for in the average hop metric, since both nodes in this case have the same machine coordinate. But this communication can impact overall communication costs. In Figure 7, we show the percentage of messages that go between the two nodes in a Gemini router (i.e., the percentage of communication that is intra-Gemini communication) for each method. Arbitrarily breaking ties among machine coordinates in Geom leads to high intra-Gemini communication; tasks with the same

	None		Group		Geom		Geom+R		Geom+R+S		TopoMap	
	Total	% Comm	Total	% Comm	Total	% Comm	Total	% Comm	Total	% Comm	Total	% Comm
4K	6.07	16.2%	5.70	13.9%	5.76	15.4%	5.75	13.6%	5.77	13.9%	5.74	14.2%
8K	6.40	18.3%	5.91	14.3%	6.05	15.7%	5.80	13.8%	5.78	13.8%	5.97	14.5%
16K	7.16	16.9%	6.14	14.2%	6.36	15.4%	6.22	14.2%	6.23	14.2%	6.15	14.6%
32K	7.60	19.0%	6.94	13.0%	7.29	13.8%	6.41	12.9%	6.38	13.0%	6.82	14.1%
64K	9.57	24.4%	8.31	12.2%	7.53	13.2%	6.29	13.4%	6.29	13.4%	8.26	13.8%

Table II

TOTAL EXECUTION TIME AND PERCENTAGE OF THAT TIME SPENT IN COMMUNICATION IN WEAK SCALING EXPERIMENTS WITH MINIGHOST.

machine coordinates are placed in either node without regard to their positions. Since intra-Gemini communication is more expensive than intra-node communication, reducing the amount of intra-Gemini communication (through grouping as in Group or better tie breaking as in Geom+R and Geom+R+S) can reduce overall communication costs. Interestingly, the very low intra-Gemini communication for the default mapping ("None") on 64K processors is due to the experiment's $32 \times 64 \times 32$-task configuration. By default, tasks are ordered by first sweeping in the x direction. Thus, with 32 tasks in the x direction, the first 16 tasks are given to one node attached to a Gemini, and the next 16 are given to its other node. Thus, only two tasks share an intra-Gemini boundary (the two in the middle of each x sweep), keeping intra-Gemini communication very low. Other configurations (e.g., $16 \times 64 \times 64$ tasks) would not have this happy benefit. These results show the importance of minimizing both intra-Gemini communication and hop count.

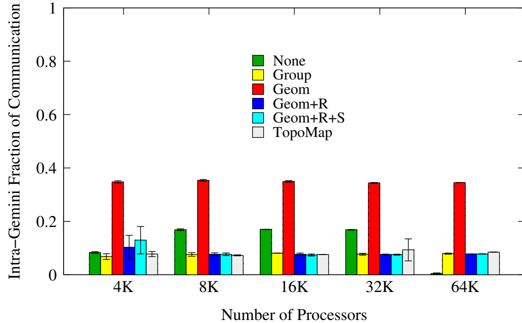


Figure 7. Intra-Gemini communication: the fraction of total communication between processors in different nodes sharing the same machine coordinate. This communication is not reflected in the hop count metric.

As a final step, we leveraged the Cray Gemini's performance counters to measure network congestion empirically [24]. Our model calculates network congestion assuming that all messages are transferred simultaneously. In reality, the message traffic is spread over time and messages interleave with one another. We measured this real-time behavior using the Gemini's per-link stall cycle counters, which increment whenever a message can not move towards its destination due to network congestion. For each Gemini being used by an experiment, we captured the stall counters for each of the Gemini's seven network links (XYZ links plus host link). We then calculated summary statistics such as minimum/maximum/average number of stalls encountered over all links, over only host links, over only X links, etc. We omit a full analysis due to space, but in general the empirical measurements closely match the predictions of our model. Table III lists the Pearson correlation coefficients comparing modeled and measured values over all 605 experiments performed for this work. The column labeled "Measured Max Stalls" corresponds to the link with the highest network stall count for each experiment (i.e., the link with the most congestion). This metric is found to have the best correlation to maximum communication time, and correlates well with our model's calculated maximum congestion metric. The modeled maximum congestion metric correlates slightly less well to the measured run times, possibly due to interference from other jobs running in the system and the heterogeneous link speeds in the network, which we do not account for. Finally, the average hop count metric is slightly less correlated to the measured run times than the congestion metrics. The empirical data suggest that our congestion metrics are accurate for MiniGhost and that the maximum congestion metric should be preferred over the average hop count metric.

	Modeled Avg Hop Count	Modeled Max Congestion	Measured Max Stalls
Max Comm Time	.835	.915	.936
Total Time	.760	.872	.886

Table III

CORRELATION COEFFICIENTS COMPARING MEASURED RUN TIMES WITH COMPUTED METRICS AND NETWORK COUNTER DATA. A COEFFICIENT OF ONE INDICATES PERFECT LINEAR CORRELATION.

Overall, our geometric mapping methods reduced the total execution time by 5-34% relative to the default MiniGhost mapping, and 0-17% relative to the application-specific Group mapping available in MiniGhost. The largest reductions were seen at the highest processor counts, reflecting the importance of mapping as the number of cores in parallel computers increases.

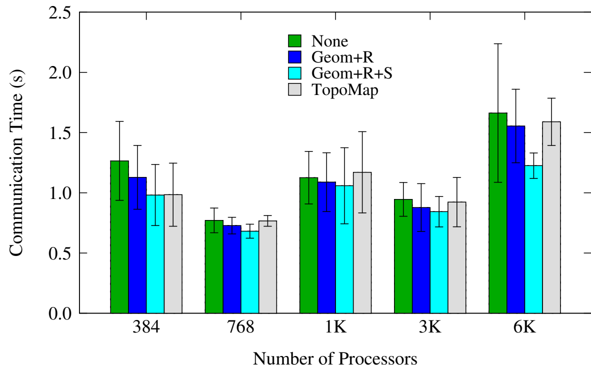


Figure 8. Maximum communication time in weak scaling experiments with MiniMD

B. *Mapping in a molecular dynamics application*

MiniMD, part of Mantevo [18], is an application proxy for parallel molecular dynamics (MD) simulations. It implements several algorithms that are typical in MD codes and emulates the key performance characteristics of larger molecular dynamics codes like LAMMPS [26]. MiniMD essentially solves Newton's equation on N particles. In our experiments, we used the MiniMD algorithm that uses the Lennard-Jones potential for force calculation. MiniMD assigns each processor a fixed spatial dimension in 3D by splitting the spatial dimension into small boxes. At each timestep, each processor computes forces and updates the positions of atoms within its box. The processor communicates with its nearest neighbors in order to compute the forces of the atoms in its box. As atoms move they are reassigned to different processors. Compared to MiniGhost, an important difference in MiniMD's communication pattern is that, based on the cutoff distance used to calculate the force, a processor might need to communicate to more than one processor in a direction when the box sizes are smaller than the cut-off. However, no processor communicates to its corner neighbors or to its neighbors that are more than one hop away. Instead it communicates to its six nearest neighbors multiple times to achieve the same purpose.

MiniMD reorders the MPI ranks into 3D processor grid and assigns the boxes to the ranks based on the rank's position in the logical processor grid. It uses the Cartesian topology interfaces in MPI (MPI_Cart_Create, MPI_Cart_Shift and MPI_Cart_get) to do the reordering. While theoretically MPI can reorder the ranks based on the topology, optimizing for nearest-neighbor communication, we do not observe that in practice. The MPI implementation usually returns the ordering we call "None." We replace the mapping of the tasks to the processor grid by using two of our best algorithms (Geom+R, Geom+R+S) and TopoMap. Our scaling studies for MiniMD were run on Hopper, the Cray XE6 at NERSC. We do a weak scaling study from 384 cores up to 6144 cores. The number of atoms increases from 415K to 6.7M. We use three repetitions for our experiments. Figure 8 shows the maximum communication time of the MiniMD runs in our weak-scaling study. Our geometric algorithms reduce the communication time when compared with the mapping provided by the MPI implementation on all processor counts. Compared to the default MPI mapping, Geom+R+S algorithm reduced the communication time of MiniMD by 6% to 27%. Unlike in MiniGhost, in MiniMD, Geom+R+S does better than Geom+R in larger core counts. At 384 cores, TopoMap ties Geom+R+S in terms of the average communication time. However, Geom+R+S mapping results in reduction in communication time over mappings from TopoMap at all core counts beyond 384. For MiniMD the reduction in communication time by using Geom+R+S over TopoMap range from 1% (384 cores) to 23% (6K cores). More notably, Geom+R+S mappings do better than TopoMap as the core counts increase. We do not present the metrics for the MiniMD runs here due to lack of space. However, as one would expect, we observed reductions in average hop count over no mapping.

VI. Conclusion

We have proposed a new topology-aware task mapping method that uses multijagged geometric partitioning (MJ) to reorder the given task and processor coordinates in a way that assigns communicating tasks to "nearby" processors. This method is designed to be effective on mesh and torus-based networks with non-contiguous node allocations, such as the Cray XE6, but extends naturally to block allocations as in IBM BlueGene systems. We also have proposed several improvements (e.g., multiple rotations, coordinate shifting) that improve geometric mappings relative to a baseline geometric method. We compared our method with the applications' default mapping, as well as application-specific mappings and graph-based mappings from LibTopoMap. to improve the quality of the mapping. We showed that our geometric mappings reduced application execution time up to 34% on 64K cores relative to the default mapping for the MiniGhost finite difference proxy application, and up to 23% on 6K cores for the MiniMD molecular dynamics proxy application. We correlated communication time in MiniGhost with computed metrics (average hops, congestion) and validated these metrics with congestion information obtained from the Gemini routers' counters.

As future work, our mapping methods will be extended to accommodate non-uniform bandwidths in the dimensions of the torus networks. We will also investigate the effect of geometric mapping on unstructured applications. And we will experiment with the processor-subset selection via k-means clustering mentioned in Section IV to provide effective mappings when there are fewer tasks than processors. Our test application will be a multigrid-based linear algebra algorithm, in which coarse matrices may be too small to efficiently utilize all cores needed for the fine matrices; in

such cases, we will use our modified k-means clustering algorithm to select a subset of cores to use.

Acknowledgment

We thank Richard Barrett, Erik Boman, Jim Brandt, Ann Gentile, Torsten Hoefler, Steve Plimpton, Christian Trott, and Courtenay Vaughan for helpful discussions. This work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Sandia is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Topology-aware mappings for large-scale eigenvalue problems," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 830–842.

[2] C. Albing, N. Troullier, S. Whalen, R. Olson, and J. Glensk, "Topology, bandwidth and performance: A new approach in linear orderings for application placement in a 3D torus," in *Proc Cray User Group (CUG)*, 2011.

[3] G. Almasi, S. Chatterjee, A. Gara, J. Gunnels, M. Gupta, A. Henning, J. Moreira, and B. Walkup, "Unlocking the performance of the BlueGene/L supercomputer," in *Proc 2004 ACM/IEEE Conf Supercomputing*, 2004, p. 57.

[4] R. Barrett, C. Vaughan, S. Hammond, and D. Roweth, "Reducing the Bulk of the Bulk Synchronous Parallel Model," *Parallel Process Lett*, vol. 23, no. 4, 2013.

[5] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2012-10431, 2012.

[6] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans Comput*, vol. C36, no. 5, pp. 570–580, 1987.

[7] A. Bhatele, G. Gupta, L. Kale, and I.-H. Chung, "Automated mapping of regular communication graphs on mesh interconnects," in *Proc Intl Conf High Performance Computing (HiPC)*, 2010.

[8] A. Bhatele, L. V. Kale, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *Proc 23rd Intl Conf Supercomputing*. ACM, 2009, pp. 110–116.

[9] S. H. Bokhari, "On the mapping problem," *IEEE Trans Comput*, vol. 100, no. 3, pp. 207–214, 1981.

[10] S. W. Bollinger and S. F. Midkiff, "Heuristic technique for processor and link assignment in multicomputers," *IEEE Trans Comput*, vol. 40, no. 3, pp. 325–333, 1991.

[11] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and U. Catalyurek, "Zoltan2: Next-generation combinatorial toolkit." Sandia National Laboratories, Tech. Rep. SAND2012-9373C, 2012.

[12] W. M. Brown, T. D. Nguyen, M. Fuentes-Cabrera, J. D. Fowlkes, P. D. Rack, M. Berger, and A. S. Bland, "An evaluation of molecular dynamics performance on the hybrid Cray XK6 supercomputer," in *Proc Intl Conf Computational Science (ICCS)*, 2012.

[13] T. Chockalingam and S. Arunkumar, "Genetic algorithm based heuristics for the mapping problem," *Computers and Operations Research*, vol. 22, no. 1, pp. 55–64, 1995.

[14] I.-H. Chung, C.-R. Lee, J. Zhou, and Y.-C. Chung, "Hierarchical mapping for HPC applications," in *Proc Workshop Large-Scale Parallel Processing*, 2011, pp. 1810–1818.

[15] M. Deveci, U. V. Catalyurek, S. Rajamanickam, and K. D. Devine, "Multi-Jagged: A scalable multi-section based spatial partitioning algorithm." Sandia National Laboratories, Tech. Rep. SAND2012-10318C, 2012.

[16] F. Gygi, E. W. Draeger, M. Schulz, B. de Supinski, J. Gunnels, V. Austel, J. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz, "Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform," in *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.

[17] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *J Roy Stat Soc C Appl Stat*, vol. 28, no. 1, pp. 100–108, 1979.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2009-5574, 2009.

[19] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proc 25th Intl Conf Supercomputing*. ACM, 2011, pp. 75–84.

[20] G. Karypis and V. Kumar, "ParMETIS: Parallel graph partitioning and sparse matrix ordering library," Dept. Computer Science, University of Minnesota, Tech. Rep. 97-060, 1997.

[21] H. Kikuchi, B. Karki, and S. Saini, "Topology-aware parallel molecular dynamics simulation algorithm," in *Proc Intl Conf Parallel & Distributed Proc Tech & Applications*, 2006.

[22] S.-Y. Lee and J. Aggarwal, "A mapping strategy for parallel processing," *IEEE Trans Comput*, vol. 100, no. 4, pp. 433–442, 1987.

[23] V. J. Leung, D. Bunde, J. Ebbers, S. Feer, N. Price, Z. Rhodes, and M. Swank, "Task mapping stencil computations for non-contiguous allocations," in *Proc 19th Symp Principals & Practice of Parallel Prog (PPoPP)*. ACM SIGPLAN, 2014.

[24] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and K. S. Hemmert, "Using the Cray Gemini performance counters," in *Proc Cray User Group (CUG)*, 2013.

[25] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking*. Springer, 1996, pp. 493–498.

[26] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J Comput Phys*, vol. 117, no. 1, pp. 1–19, 1995.

[27] C. Walshaw and M. Cross, "Multilevel mesh partitioning for heterogeneous communication networks," *Future Generation Comp Syst*, vol. 17, no. 5, pp. 601–623, 2001.

[28] H. Yu, I.-H. Chung, and J. Moreira, "Topology mapping for Blue Gene/L supercomputer," in *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.