Automatic Topology Mapping of Diverse Large-scale Parallel Applications

Juan J. Galvez
University of Illinois at Urbana-Champaign
jjgalvez@illinois.edu

Nikhil Jain
Lawrence Livermore National Laboratory, CA
nikhil@llnl.gov

Laxmikant V. Kale
University of Illinois at Urbana-Champaign
kale@illinois.edu

ABSTRACT

Topology-aware mapping aims at assigning tasks to processors in a way that minimizes network load, thus reducing the time spent waiting for communication to complete. Many mapping schemes and algorithms have been proposed. Some are application or domain specific, and others require significant effort by developers or users to successfully apply them. Moreover, a task mapping algorithm by itself is not enough to map the diverse set of applications that exist. Applications can have distinct communication patterns, from point-to-point communication with neighbors in a virtual process grid, to irregular point-to-point communication, to different types of collectives with differing group sizes, and any combination of the above. These patterns should be analyzed, and critical patterns extracted and automatically provided to the mapping algorithm, all without specialized user input. To our knowledge, this problem has not been addressed before for the general case.

In this paper, we propose a complete and automatic mapping system that does not require special user involvement, works with any application, and whose mapping performs better than existing schemes, for a wide range of communication patterns and machine topologies. This makes it suitable for online mapping of HPC applications in many different scenarios.

We evaluate our scheme with several applications exhibiting different communication patterns (including collectives) on machines with 3D torus, 5D torus and fat-tree network topologies, and show up to 2.2x performance improvements.

KEYWORDS

Topology aware mapping, Automated mapping, Network topology, Profiling

ACM Reference format:
Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. 2017. Automatic Topology Mapping of Diverse Large-scale Parallel Applications. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079104

This research is part of the Blue Waters sustained-petascale computing project, which is supported by NSF award number OCI 07-25070 and the State of Illinois. Blue Waters is a joint effort of the University of Illinois and NCSA. 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-728887).

1 INTRODUCTION AND MOTIVATION

The ever increasing demand for computational power has resulted in deployment of supercomputers with tens of thousands of nodes. Due to the large number of nodes, scalability of applications is often limited by the communication performance of current generation supercomputers. As computational capacity continues to rapidly increase due to introduction of GPUs and Many-core CPUs, network resources are expected to remain critical in next generation systems.

Network aware mapping of tasks to processors, referred to as *task mapping*, is one of the primary ways of improving the observed communication performance. A good mapping can reduce overall load on the network in two ways: 1) avoiding network communication by co-locating heavily communicating tasks on the same node, and 2) by reducing the average and worst-case loads on links.

Existing mapping strategies are either application-specific [1–3] or can require too much effort on the part of users or developers to adopt. For example, *Rubik* [4] is a manual scheme that assumes structured communication patterns and knowledge of the application's virtual topology, and may require manual calculation of a solution for each topology and shape. General purpose schemes typically provide a generic mapping algorithm [5, 6], but it is up to the user to obtain a suitable communication graph of their application, convert to a format which the implementation understands, and translate the solution to a format which the machine scheduler can use. As a result, there has not been a widespread adoption of topology mapping techniques, even though they have been proven to (sometimes substantially) improve performance.

Automatic task mapping is attractive because it can reap these benefits for any parallel application on any architecture, topology and allocation shape without specialized user knowledge. In this paper we propose a novel mapping scheme named TopoMapping[1], designed to meet the following goals:

- Is automatic and works for any MPI application (no user involvement or input required).
- Produce solutions given any time constraint. Because optimal mapping depends on the particular job allocation, it must be calculated "on-line" right before the application starts; thus it must not increase the overall job execution time and in fact should shorten it.
- Produce solutions at least as good as current best-performing algorithms, including expensive algorithms and those based on graph partitioning.
- Can optimize based on multiple criteria. Existing algorithms typically use a single metric as predictor of application performance. Proposed metrics do not guarantee

[1]TopoMapping is currently available to users on Blue Waters [7] as a system module.

strong correlation, therefore using multiple predictors can improve performance as we will show.

We could not find an existing scheme that satisfies all of these goals. A critical component in such a system is automatic communication graph generation, i.e. to profile the communication pattern of an application and generate a suitable graph for the mapping algorithm. Some parallel applications make heavy use of collectives, and this information has to be included in the communication graph. However, including all patterns is not desirable, because this can produce graphs with high node degree that will slow down most mapping algorithms, or impede the optimization of the patterns that are more critical for the application. To the best of our knowledge this has not been adequately addressed before.

TopoMapping also implements a parallel mapping scheme capable of finding good solutions quickly for a wide range of problems. The scheme is particularly tailored for HPC scenarios. It exploits the high parallelism in a job by running separate instances of a custom randomized mapping algorithm on each processor. The search space depends on the parameters and heuristics chosen for each particular instance of the algorithm, thus allowing for a deep search over a wide search space.

The main contributions of this paper are summarized below:

- Automatic mapping system that adapts to application, topology and shape without specialized user control. An important part of this scheme is selection of the optimal communication graph for the given scenario.
- Parallel mapping scheme that exploits job parallelism and fast mapping algorithm to quickly find good solutions, and selects best solution based on multiple criteria.
- Configurable mapping algorithm whose search space and behavior depends on input parameters and heuristics.
- Evaluation of scheme in terms of hopbytes and congestion, two of the most relevant metrics in literature.
- Evaluation of scheme with several HPC applications, including MILC [8], Qbox [9], and pF3D [10] on CrayXE, BG/Q and fat-tree clusters.

2 RELATED WORK

Many existing mapping schemes are either designed for specific applications, networks, structured graphs, or require some form of manual tuning from the user to optimize the solution for the application of interest [3, 4, 11, 12].

Several general-purpose schemes have also been proposed over the years. In general, they only focus on mapping algorithms, proposed for either generic Quadratic assignment problems (QAP) [13] or specifically for large mapping problems in HPC scenarios. For example, Vogelstein et al. [13] recently proposed the Fast Approximate Quadratic (FAQ) assignment algorithm. They show it to be faster and achieve lower objective values for large part of the QAPLIB benchmark library compared with previous state-of-the-art. In this paper we compare our scheme with FAQ, and show that FAQ is too expensive for many large-scale HPC problems. Agarwal et al. [6] proposed an algorithm designed for mapping and load balancing of parallel applications. The algorithm can also be applied to QAP problems. It has a similar complexity compared to FAQ and we also evaluate it in this paper. In [5] Hoefler et al. propose three general algorithms intended for mapping of large-scale applications. They evaluate the algorithms using a congestion metric, and sparse-matrix vector multiplication code, but don't evaluate the mappings using production HPC applications. We will show that our scheme outperforms these for a wide range of problems.

Some mapping heuristics are based on graph partitioning. One example is Recursive Bisection in [5] where task and network graphs are recursively partitioned in two at each stage using a graph partitioner like METIS [14] or Zoltan [15]. The Scotch [16] suite implements a similar mapping scheme. We also compare our scheme with the recursive bisection scheme [5] using METIS.

The scheme we propose matches or outperforms existing schemes, addresses the issue of communication graph generation and limitation of single performance metrics, and is automatic, thus facilitating adoption for any MPI application in many different systems.

3 TASK MAPPING PROBLEM

In this paper we use the well-known Task Graph Model (TGM) [5, 6]. Here we describe the model, and explain two important limitations which are frequently ignored by existing schemes that need to be addressed for effective mapping.

3.1 Task Graph Model (TGM)

Let the *task graph* be an undirected graph $G_t = (V_t, E_t)$ representing the communication of an application with $|V_t|$ tasks. There is an edge $e_{ab} \in E_t$ between two tasks $a, b \in V_t$ with weight $w(e)$ if they communicate a total of w bytes in both directions.

Let $\mathcal{N}$ be the set of nodes in the system, and $G_n = (\mathcal{N}, E_n)$ a directed graph representing the network topology. An edge exists between nodes if there is a network link directly connecting them. A node n has a set $P(n)$ of processors (which can be empty for nodes that should not be used for mapping, i.e. nodes not allocated to the current job). The total set of processors is given by $P = \cup_{n \in \mathcal{N}} P(n)$. We assume that $|V_t| = |P|$, i.e. each processor will run one task.

The task mapping problem consists of assigning tasks to processors such that some objective function is minimized. Let $S : V_t \mapsto P$ denote an assignment of tasks to processors. This is a combinatorial optimization problem with $n!$ possible assignments, where n is the number of tasks.

3.2 Optimization objectives

Accurately modeling an application's execution time and solving the mapping problem based on this criteria, particularly for a general-purpose scheme, proves too complicated. Previous work has proposed minimizing indirect objectives which are simpler to model, like hopbytes [6], network congestion or dilation [5]. For these objectives, the problem of task mapping has been shown to be NP-hard. Most formulations have focused on optimizing hopbytes, making the problem equivalent to a Quadratic assignment problem[2].

Given a task mapping S, hopbytes of task t is defined as:

$$\text{hopbytes}(S, t) = \sum_{e_{t,u} \in E_t} w(e) \times \text{distance}(S(t), S(u)) \qquad (1)$$

[2] Furthermore, finding an approximate solution to QAP within some constant factor of the optimal cannot be done in polynomial time unless P=NP [17].

where distance(p, q) is the number of network links between processors p and q. The average hopbytes $\bar{h}$ and maximum hopbytes $\hat{h}$ are defined as:

$$\bar{h} = \frac{\sum_{t \in V_t} \text{hopbytes}(t)}{|V_t|} \tag{2}$$

$$\hat{h} = \max_{t \in V_t}\{\text{hopbytes}(t)\} \tag{3}$$

We will measure congestion in terms of the maximum link load in the system. Let the binary variable l_{tu} indicate if traffic between tasks $t, u \in V_t$ traverses link l, then the maximum link load is defined by[3]:

$$\text{maxload} = \max_{l \in E_n}\{\text{load}(l)\} \tag{4}$$

$$\text{load}(l) = \sum_{e_{tu} \in E_t} w(e) \quad \text{where } l_{tu} = 1 \tag{5}$$

3.3 Limitations of Task Graph Model

TGM has two important limitations which are frequently ignored or not adequately addressed:

3.3.1 Optimization criteria. One obvious limitation is that there is no guarantee that the optimization objectives used in TGM will correlate with application execution time. In fact, correlation may depend on various factors including the specific application, the machine topology, network architecture and shape of the allocation. As such, there are situations when using only one criteria as predictor proves insufficient.

Consider the performance results for MILC [8] on BG/Q with different mapping solutions, shown in Table 1. MILC has heavy point-to-point (p2p) communication pattern which typically accounts for most of the communication time. It also uses a global Allreduce which induces global synchronization and is sensitive to imbalance. Hopbytes is measured only for p2p (expressed as hops per byte in table for legibility).

As we can see, there are solutions with similar mean hopbytes, but different max hopbytes. Mean and max hopbytes correlate very strongly with mean and max p2p time, respectively. For MILC, the imbalance resulting from high max p2p time has a large influence on the global Allreduce, and negatively impacts application execution time. As such, if we want to use hopbytes as predictor for performance of MILC, the average (or equivalently the total hopbytes), which is the widely used predictor when measuring hopbytes, proves inadequate and the maximum must also be considered.

Table 1: Effect of Avg and Max hopbytes on MILC performance on BG/Q (using different mapping solutions)

avg hops per byte	max hops per byte	mean p2p time	max p2p time	avg comm time	max comm time	exec time
0.71	1.14	55.5	76.5	78.0	88.1	618.3
0.73	1.37	56.7	89.5	92.7	106.2	631.3
0.88	4.79	75.0	266.0	256.5	291.3	783.3
1.61	2.02	96.4	113.6	122.8	134.2	646.9

[3]In systems with dynamic routing, this metric will vary dynamically.

Many existing mapping schemes do not consider multi-objective optimization (e.g. [5, 6]).

3.3.2 Task graph. In general, previous work either implicitly assumes that the task graph will include all communication between tasks in the application, or leaves the responsibility of building a suitable graph to others. This, however, is not trivial and imposing it upon users only serves to limit adoption of topology mapping techniques. Many applications have a mix of different communication types which occur in different phases of the application, and congestion can happen at different times. Applications can also employ collectives which involve communication between a large number of tasks[4]. Collectives can account for a significant amount of the run time and should not be ignored. However, including any and all information in the graph is undesirable because:

(1) Collectives can substantially increase the node degree in the task graph, increasing the time to solution of most mapping algorithms, with no guaranteed benefit to mapping quality and application performance. Since we require fast calculation, this aspect must be controlled.
(2) Many collectives that involve all tasks or a significant number of tasks don't benefit from mapping.
(3) Communication volume does not necessarily correlate with communication time, and trying to optimize for everything makes it harder to optimize for the communication types that are more important for the application.

To illustrate point 3, suppose we have an application where the volume of communication in bytes of an Alltoall does not have the same effect on performance as the same volume of communication of p2p exchanges. This can happen for various reasons, e.g. some machines may optimize different types of collectives and communication operations in different ways, or these operations might occur in different stages of an application with possibly different overlap of computation and communication. Since it may not be possible to minimize the hopbytes of both types of communication, we need to select the trade-off that's right for the application. A real-world example of this is shown in Section 4.1.

4 TOPOMAPPING SCHEME

As we explained at the beginning of the paper, we desire a process that automatically maps any parallel application without specialized user knowledge or control. The main novelty of our scheme is that it can achieve this while addressing the limitations of the TGM. The principal components of our proposed strategy are:

- Automatic application profiling step that generates a suitable task graph.
- Configurable mapping algorithm: takes as input the task graph, network graph, set of heuristics and cost function.
- ParMapper: Parallel mapping calculator that runs multiple instances of the mapping algorithm at the same time and selects the best solution based on multiple criteria.

The TopoMapping scheme is outlined in Fig. 1. It consists of a profiling phase where the best communication graph for the application (referred to as Task_Graph*) is generated. Production runs will use this graph and ParMapper to calculate the best task

[4]Examples are MPI Alltoall, AllReduce, Gather, Scatter.

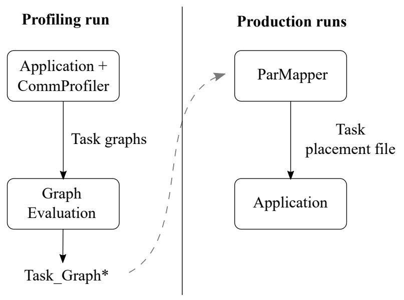

Figure 1: Overview of TopoMapping scheme.

placement for the given job allocation. The result is a map file that is used by the system to launch the application. We explain the process and components in detail below.

4.1 Automatic profiling and task graph generation

As explained in 3.3, a critical step in the mapping scheme is automatic generation of suitable task graphs. This is done in the profiling phase, which is performed once for a given job size. In this phase, the user runs the application (linked with a profiling library called CommProfiler) using the same or similar parameters as would be used in production runs. The duration or number of iterations can be small, as long as communication that occurs in that time is representative of the patterns that occur during a complete run (i.e. it is not important to record the exact volume in bytes between tasks, but rather the connections and weights between them).

4.1.1 CommProfiler. We have developed a PMPI-based profiling library, called CommProfiler, which can be linked with any MPI application to record useful information about the communication behavior of the application. This information includes time spent waiting for communication to complete classified by *type*, and the amount of bytes sent between MPI ranks for each type. Supported types are point-to-point (p2p) and all of the different MPI collectives (Alltoall, Gather, etc).

The weight of a type is the amount of time spent in that operation. We identify the most important types used by the application based on their weight during the profiling run. First, large collectives (global or involving a substantial amount of tasks) are filtered out, because they consist of a significant portion of the nodes in the job, so there is little to no room for improvement through mapping. For the remaining communication types, we select the heaviest ones whose total time is a significant percentage of the total communication time. Let C^* be the most critical types identified in this fashion. For example, if the application spends 85% of its communication time doing point-to-point and Alltoall, $C^* = \{\text{p2p, Alltoall}\}$.

For each type t in C^*, the profiler generates a cg_t task graph containing only communication of that type. Next, it generates two graphs *Sum* and *Weighted-sum*, with aggregated information of the cg_t graphs. The *Sum* graph contains the union of tasks of cg_t graphs, and weights of edges are the sum of weights from the cg_t graphs. The *Weighted-sum* graph also contains the union of tasks from the cg_t graphs, but weights of edges are proportional to the amount of time spent by the type in the profiling run. For example, if p2p represents 60% of the communication time and Alltoall 40%, the edge weights of the former are multiplied by 0.6 while the edge weights of the latter are multiplied by 0.4.

Table 2: Qbox execution time using different task graphs (All-to-all are non-global collectives)

Task graph	avg p2p time	avg a2a time	exec time (s)
Point-to-point (p2p)	7.73	49.87	168.72
All-to-all (a2a)	44.96	22.80	192.54
Sum	21.54	22.62	166.17
Weighted-sum	16.60	23.88	161.49

4.1.2 Task graph evaluation. The information collected by CommProfiler gives some intuition about how the application will respond to mapping based on the identified critical types. However, without specialized knowledge of the application, it is difficult to know a priori which graph is best. In order to select the best graph, we provide an automatic script that tests the graphs by mapping and running the application with each of them (using ParMapper). The graph that leads to the lowest execution time (Task_Graph*) will be the graph used in production runs (see Fig. 1).

We should note that the evaluation is tied to the job allocation where the profiling run is performed. There is no guarantee that the resulting task graph is the best choice for all possible allocations in the machine. However, in most cases the graphs generated by CommProfiler are sufficiently distinct that the result translates to different allocations. Also, the graph evaluation step is optional, but if the user decides to skip it, he must determine which of the graphs generated by CommProfiler to use for production runs.

Table 2 shows the effect of task graph when mapping Qbox [9] on Blue Waters [7]. The identified critical types C^* in this scenario are p2p and non-global Alltoall. We observe that p2p has a greater effect on performance, and also how optimizing for a single type can have the greatest effect on performance of that type, as opposed to optimizing for multiple types. The best graph in this case is *Weighted-sum*, because it is possible to optimize for both p2p and Alltoall while taking into account the greater weight of p2p.

4.2 ParMapper: The parallel map calculator

The ParMapper utility runs inside the job prior to the application, takes as input the task graph obtained in a separate profiling run, and calculates a task mapping optimized for the job (i.e. the specific topology and geometry where the scheduler has placed the job).

ParMapper uses the TopoMgr open-source library [18] that provides an abstract API to access topology information like set of nodes in the job, processors in each node and distance between nodes. TopoMgr supports Cray XT/XE machines and Blue Gene systems. It can easily be extended to other systems (we added support for Catalyst which has a fat-tree network).

ParMapper exploits the parallelism in the job by running a custom greedy randomized mapping algorithm (named GreedyMap) simultaneously on multiple processors. The search space explored by GreedyMap (i.e. the set of solutions it can reach given infinite number of trials) varies depending on the parameters and heuristics chosen. ParMapper can choose different combinations of parameters for each instance, called *configurations*. This will be described in detail later. Once every instance finishes, or the time limit elapses, results are communicated to one processor, who determines the best solution.

Multiple trials of GreedyMap with different search spaces produce a wide spectrum of solutions, which is particularly helpful in ensuring that the scheme is effective for a large number of applications, topologies and allocation shapes. Furthermore, it facilitates choosing solutions based on multiple criteria. This capability is beneficial for task mapping because, like we showed in 3.3, we cannot expect individual predictors by themselves to correlate perfectly with application execution time. The default[5] strategy used by ParMapper to select the best solution is to generate the Pareto frontier [19] based on two metrics: $\bar{h}$ and $\hat{h}$ (see Eqs. 2 and 3). Let $\bar{h}_0$ be the smallest $\bar{h}$ in the Pareto set. ParMapper will select the solution i with smallest $\hat{h}_i$ that satisfies $\bar{h}_i < \alpha \cdot \bar{h}_0$, where $\alpha \geq 1$.

4.3 TopoMapping run time

Let M be the time to run ParMapper. M is user-configurable. We recommend 1 minute for task graphs with low node degree (like MILC), and 3-10 minutes for others (containing large collectives, like Qbox). We use these time limits in Section 6.

For *profiling* runs, the time required by the scheme is $T = P(n + 1) + nM$ where n is number of graphs generated by CommProfiler, and P the time to run the application in profiling mode (for each graph, we need to calculate a mapping and run the application in order to evaluate it). For example, in our experiments, average execution time P with Qbox was 2.5 minutes, $n = 4$ (as seen in Table 2), and we chose $M = 5$, making T roughly 32.5 minutes. For MILC, $n = 1$ (p2p graph), $M = 1$, $P = 5$, thus $T = 11$.

For *production* runs, with the best graph already known, the only requirement is to run ParMapper (scheme run time is thus M).

4.4 GreedyMap mapping algorithm

We now explain the GreedyMap algorithm used as part of our scheme. One of the most important design constraints for GreedyMap is low complexity, because we require finding solutions given any possible time constraint by the user. This precludes using more complex algorithms, including existing greedy algorithms [5, 6]. Lower complexity heuristics translate to reduced search, risking lower quality solutions. To compensate for this, we exploit the parallelism in the job by running multiple instances of GreedyMap simultaneously using different *configurations* (i.e different set of parameters and heuristics). Note that a configuration doesn't simply change the starting point of GreedyMap as in multi-start procedures like GRASP [20]. Here it alters the heuristics used for the search. As we will see, this allows achieving solutions similar in quality to $O(n^3)$ algorithms in less time.

GreedyMap is shown in Algorithm 1. It takes as input the set of nodes $\mathcal{N}$ in the job allocation, the task graph G_t, and a *configuration*. It iterates sequentially over the tasks in T, and in each step places the task in the best free processor. Cost is estimated using a function that measures the cost of the partial solution. Cost functions include measurement of hopbytes, max hopbytes and max link load.

Algorithm 1 GreedyMap

```
Input: $\mathcal{N}$, $G_t$, $T$, $\Delta$, packNodeFirst, cost_function
Output: Task mapping $S : V_t \mapsto P$
1:  freeProcessors($n$) ← number of available processors in $n$ $\forall n \in \mathcal{N}$
2:  $S \leftarrow \emptyset$ // Initialize empty solution
3:  $n_l \leftarrow$ node of $p_0$ // last used node
4:  for $t \in T$ do
5:      if packNodeFirst and freeProcessors($n_l$) > 0 then
6:          $\mathcal{K} \leftarrow$ {free processor in $n_l$}
7:      else
8:          $\mathcal{K} \leftarrow$ processors with same lowest cost in $\Delta$ closest nodes to $n_l$
9:      end if
10:     $p \leftarrow$ random processor from $\mathcal{K}$
11:     $S(t) \leftarrow p$ // assign task $t$ to processor $p$
12:     $n_l \leftarrow$ node of $p$
13:     freeProcessors($n_l$) ← freeProcessors($n_l$) −1
14: end for
```

When a task has been placed in a node, the parameter 'packNodeFirst', if TRUE, forces subsequent tasks to be assigned to that node until it is filled. This configuration notably reduces the complexity of the algorithm and works well in combination with other heuristics as we will analyze later. If packNodeFirst=FALSE or current node has been filled, GreedyMap examines the Δ closest nodes, and selects a *random* processor from the set with equal lowest cost.

4.4.1 GreedyMap complexity analysis. The complexity of the algorithm depends on the configuration used. Let $N = |\mathcal{N}|$ be the number of nodes in the allocation, and $T = |V_t|$ the number of tasks. Note that nodes contain multiple processors and one task is assigned to each processor, so $N < T$.

In the slowest configuration (packNodeFirst=FALSE), Δ nodes are explored in each iteration, and the cost function is applied for the current task (which requires traversing the task's neighbors in the task graph G_t). The complexity of GreedyMap in this case is thus $O(T\Delta d)$ where d is the average node degree in G_t. Δ is at most N, and some configurations use a reduced value like $\sqrt{N}$.

We should note that $d \ll T$ in general. Useful graphs tend to be sparse. As explained in 4.1, we ignore collectives involving a significant number of tasks when generating task graphs.

If packNodeFirst=TRUE, the complexity is $O(T + N\Delta d)$, which is considerably less than the previous configuration, particularly with "fat" nodes (with many processors)[6].

As we can see, the complexity depends on the configuration used. ParMapper can return, if necessary, a solution as soon as one of the configurations is finished.

4.5 GreedyMap configurations and heuristics

A configuration of GreedyMap is given by the tuple $(T, \Delta, \text{packNodeFirst}, \text{cost\_function}, \text{Torus\_Wrap})$ where T is the task list. Torus_Wrap is only applicable for torus topologies and, if true, allows wrapping through torus edges when exploring nodes.

[5]Hop-bytes is faster to calculate than congestion and available on many machines. A congestion metric, on the other hand, requires knowledge of routes which in systems with dynamic routing like BG/Q is not known prior to running the application.

[6]On Blue Waters [7], one compute thread runs this configuration with MILC [8] communication graph (65k tasks mapped to 4096 nodes) in roughly 0.5 seconds.

The total number of configurations is system-dependent but is less than 100. ParMapper will run every possible configuration in parallel, and a given configuration will also run on multiple processors to exploit randomness of GreedyMap. We will analyze the effect of different heuristics in Section 5.4.

The task list determines the order in which GreedyMap processes the tasks. For many combinatorial problems, selection order is known to be a powerful way of achieving good approximation ratios with greedy algorithms[7]. This order has a major impact with GreedyMap, and different orders yield different results depending on the task graph and network topology. GreedyMap supports the following lists, and custom ones may be provided by users:

(1) OO: tasks in original (application) order.
(2) BFS (Breadth-first search): Starting from one of the tasks, tasks are inserted in the list by performing a breadth-first traversal of the task graph (this is equivalent to doing a bandwidth reduction [22] on the task graph).
(3) BFS-DFS (breadth-first and depth-first search combination): Paths in the task graph are explored in a depth-first manner. When the depth-first search cannot progress, breadth-first search is used to pick the next task.
(4) GPART (min edge-cut): Task graph is partitioned into a set of N (the number of allocated nodes) partitions that minimize edge-cut, using partitioners like METIS [14]. The task list is then composed by placing tasks that are in the same partition into consecutive positions in the list.

The first three lists can be generated in $O(T)$, while the last depends on the complexity of the graph partitioning algorithm.

5 EVALUATION OF TASK MAPPING SCHEMES

In this section we evaluate the TopoMapping scheme using the hopbytes and congestion metrics and compare with many previously proposed algorithms and common heuristics. We evaluate using regular and irregular task graphs, under a variety of network topologies (3D and 5D torus, and fat-tree). In Section 6, we show performance of production applications with TopoMapping.

5.1 Algorithms

We compare TopoMapping with the following algorithms:

- TopoLB by Agarwal et al. [6] of $O(n^3)$ complexity.
- Fast Approximate Quadratic [13] of $O(n^3)$ complexity.
- Algorithms by Hoefler et al. [5]: Greedy, Recursive Bisection and Graph similarity.
- List mapping variations consisting of assigning task i in a list of tasks to processor i in a list of processors. We use the first three task lists from Section 4.5. To form the lists of processors, we use planar order and Hilbert curve (based on geometrical coordinates of processors). The complexity of these schemes is $O(n)$.

Together with ParMapper, they represent a total of twelve different schemes. We have implemented them in C++ according to their published descriptions, except FAQ for which we use the authors' MATLAB implementation[8]. In this section, the *default* scheme refers to the assignment of tasks in application order to a list of processors generated in planar order. RCM refers to the graph similarity algorithm of [5].

[7]For example, for the job scheduling problem to minimize makespan, a simple greedy algorithm is at worst 4/3-optimal, by simply ordering jobs by weight [21].
[8]Available at https://github.com/jovo/FastApproximateQAP

For each solution, we measure the hopbytes (Eq. 1), congestion (Eq. 4) and time to obtain the solution. Results shown are normalized to the lowest observed value, where 1 represents the best value obtained for that metric. For ParMapper, we run *all* of its configurations in parallel, and choose the best solution according to Section 4.2 (i.e. it considers both average and maximum hopbytes).

To measure congestion as defined in Eq. 4, we need to know the routes between nodes. For the experiments in this section we use a heuristic that selects, for each pair of communicating tasks, the currently least loaded shortest path and updates the load of links accordingly. It is not guaranteed to be optimal but produces less congestion than a random shortest path strategy.

We will report the time taken by FAQ, TopoLB and Recursive Bisection when explaining the results. ParMapper is run with fixed durations shown next to its name (10, 20 and 60 s). For the rest of heuristics the calculation time is only a few seconds and so will not be mentioned.

Because TopoLB and FAQ have $O(n^3)$ complexity and run slowly on most of the problems tested, we partition the task graph to size N where N is the number of nodes, and use the algorithm to map the N coarsened tasks to nodes (i.e. does not map tasks to processors). We only show results with TopoLB and FAQ for cases where we could obtain solutions in under one hour.

Results are shown in Figs. 2-5. The shape of the topology is indicated at the top of each figure, the last number being the number of processors per node.

5.2 Regular communication graphs

5.2.1 MILC. MILC [8] is a widely used application for studying quantum chromodynamics (QCD). It simulates four dimensional SU(3) lattice gauge theory by defining quark fields on a 4D grid of space time points. These points are divided among MPI processes, which are also arranged in a virtual four dimensional grid. Most of the communication in MILC is near-neighbor in which every MPI process exchanges data with its eight neighbors in the 4D grid.

We use the MILC task graph obtained from CommProfiler (containing p2p communication, i.e. MPI_Send calls and its variants). The results are shown in Fig. 2. As we can see, ParMapper finds the best solution in terms of hopbytes in all cases. The max link load of the solution found by ParMapper is in many cases the smallest or at worst 24% higher. Note that differences in max link load between ParMapper schemes are due to the fact that ParMapper is trying to minimize hopbytes, not max link load. Minor differences in hopbytes are due to random aspect of GreedyMap.

Other algorithms that perform well for MILC are FAQ and TopoLB. TopoLB takes more than 3 minutes to run for problem sizes with 65,536 tasks and 2,048 nodes, while FAQ takes almost 17 minutes for the largest problem. Recursive Bisection performs well in some cases but not in others (e.g. for 3D torus of shape 16x2x16, hopbytes is 46% higher than the best and max link load is 65% higher), and runs in less than 30 seconds. The performance of Greedy is not very consistent, with its solution being more than two times worse than the best one in some cases.

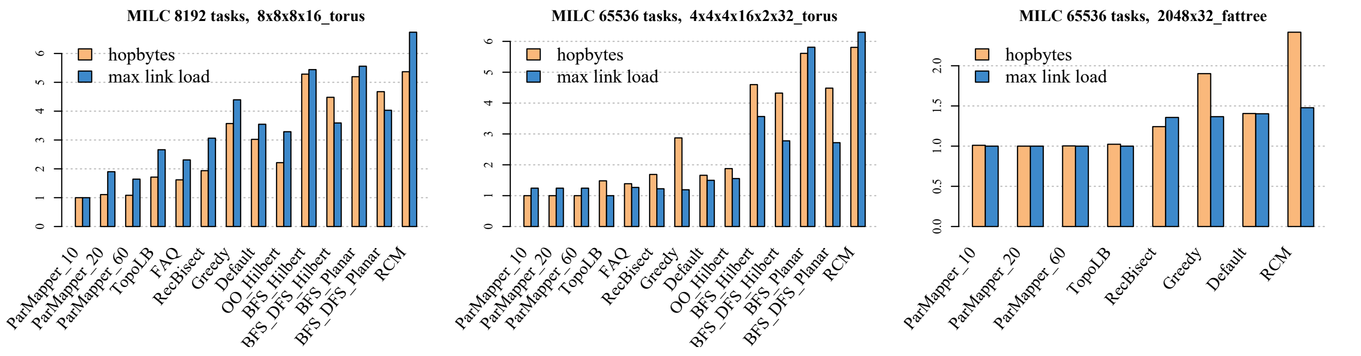

Figure 2: Mapping results with MILC task graph. Left axis results normalized (one represents best value found).

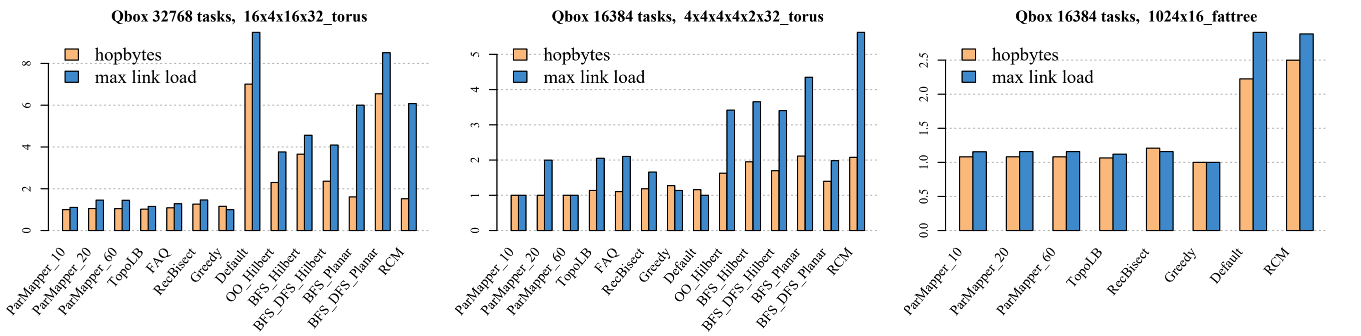

Figure 3: Mapping results with Qbox task graph (composed of point-to-point and Alltoall).

5.2.2 *Qbox.* Qbox [9] is a scalable implementation of first-principles molecular dynamics based on the plane-wave pseudopotential formalism. MPI processes are arranged in a 2D grid, with different electronic states divided among the columns. FFT operations within electronic states result in Alltoall among MPI processes that are part of each column. Large sized broadcast and reductions are performed among MPI processes that are part of each row.

The task graph for Qbox is generated by CommProfiler and contains point-to-point and Alltoall communication. Fig. 3 shows that ParMapper is the best performing in terms of both metrics in all cases. TopoLB closely approximates the best solution, particularly in terms of hopbytes, but took 23 minutes to run for the scenario with 32k tasks and 2,048 nodes. FAQ performs similarly to TopoLB, but we could only run it for the smaller problem sizes (1024 nodes or less, taking up to 130 seconds). Greedy, which didn't perform consistently for MILC, performs well for Qbox, although max load is 65% worse than the best in one of the cases. Recursive Bisection performs slightly worse than Greedy, and the max load in the second scenario (5D torus with 16k tasks) is 66% worse than the best. Recursive Bisection took at most 18 seconds to run.

5.3 Irregular communication graphs

To capture the characteristics of irregular applications, we use the real-world matrices *F1* and *scircuit* from the University of Florida Sparse Matrix Collection (a similar analysis of mapping algorithms was conducted in [5]). We focus on large problem sizes from now on, and avoid TopoLB and FAQ based solutions for large scenarios.

5.3.1 *F1.* In Fig. 4, we can see that OO_Hilbert is the best performing strategy in general for this problem, but ParMapper is close to it. Also, ParMapper finds the best solution in fat-tree (note that Hilbert curve doesn't apply in this topology).

TopoLB performs very well in the case where we used it (with a time to solution of almost 3 minutes), but we didn't test it in the other cases due to the large problem size. The solution found by Greedy is more than 2 times worse than the best in some cases. The hopbytes of Recursive Bisection are within 30% of the best solution, but the max load is more than 2 times worse in some cases. Recursive Bisection took up to 77 seconds (5D torus scenario).

5.3.2 *scircuit.* This graph is highly irregular, with a few tasks having many more neighbors than the rest, turning them into severe bottlenecks if not mapped correctly. Results are shown in Fig. 5. As we can see, many of the solutions are very far from the best observed value (y axis is logarithmic scale), with solutions up to 200 times worse than the best.

In terms of congestion, ParMapper finds the best solution or closely approximates it. OO_Hilbert performs best in general for torus topologies, while ParMapper finds the best solution in fat-tree. Solutions found by Greedy and Recursive Bisection are far from the best value, particularly with congestion, which suggests that the critical tasks were not mapped correctly. Recursive Bisection took up to 65 seconds to calculate a solution.

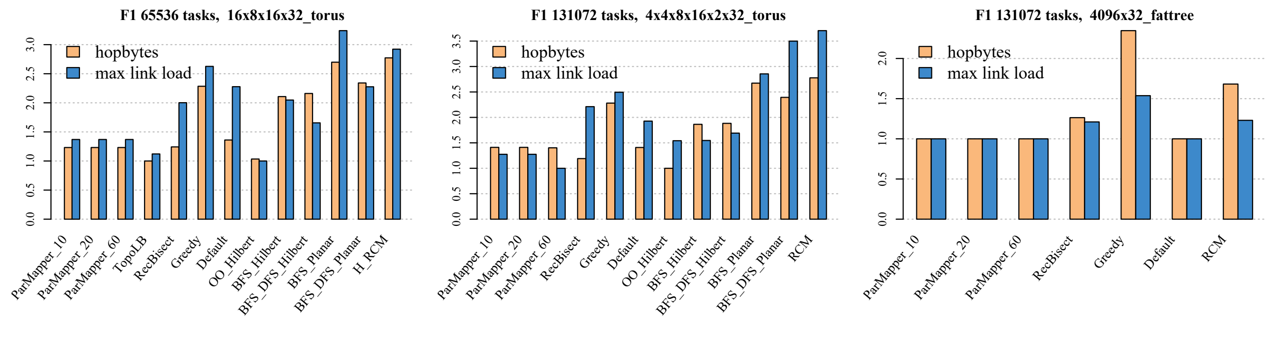

Figure 4: Mapping results with graph based on *F1* matrix from UFL sparse matrix collection.

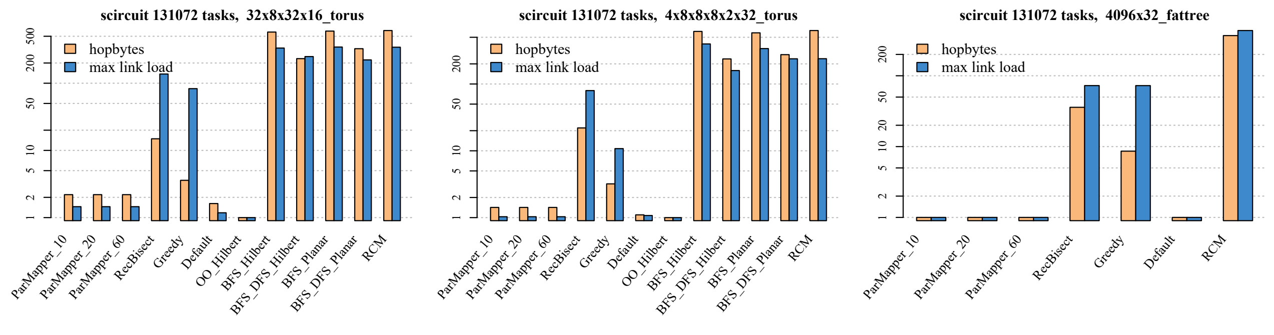

Figure 5: Mapping results with graph based on *scircuit* matrix from UFL sparse matrix collection. Y-axis is log-scale.

5.4 Analysis of heuristics

In the wide range of problems tested, the schemes that perform best in general are ParMapper, TopoLB and FAQ. TopoLB and FAQ are largely equivalent in terms of quality but are both too expensive to be used for large problems. Of the remaining schemes, Recursive Bisection, Greedy, OO_Hilbert and default can obtain good results in some cases, but not in every scenario.

In general, the heuristic that has the largest influence on solution quality with ParMapper is task selection order. All of the orderings considered are heuristics aimed at grouping together tasks that communicate between themselves. There is not a single strategy which is best for every scenario.

With MILC and scircuit, the best task list is OO. Often, the application order of tasks contains information about the structure of the problem domain, and mapping this structure to processors in a way that minimizes cost obtains best results. Note that this is also the reason why other strategies that use OO sometimes perform well. However, their strategy of selecting processors is not optimal for every geometry. Default, for example, selects processors in the order given by the machine. packNodeFirst=True and OO usually work well together, because packNodeFirst helps to preserve the structure given by the application.

OO does not always produce best results, however. On torus, GPART sometimes performs better with Qbox, while BFS is always best in F1. On fat-trees, GPART is usually the best for every test case. The main reason for this is that existing graph partitioning algorithms are effective at minimizing communication between partitions. GreedyMap will roughly assign each partition to a node which, by the fact that there are very small variations in distance between nodes in fat-trees, leads to good solutions. It is interesting to note that GreedyMap finds the best solution on fat-trees with packNodeFirst=False, which means it does not necessarily preserve the exact group structure given by GPART.

In general, the purpose of Δ is to control calculation time at the expense of solution quality, although sometimes lower Δ can result in better solutions by keeping the partial solution of GreedyMap constrained to the current processor space being explored, which keeps indirectly related tasks close together.

We should also note that we have not used congestion metric as cost function in these experiments, but ParMapper was still able to find solutions with low congestion, suggesting that multi-criteria optimization (based on average and max hopbytes) also helps minimize the congestion metric.

6 APPLICATION PERFORMANCE

In this section we show results running production HPC applications with our automatic mapping scheme on several systems. Each system has its own default mapping strategy to place tasks. Depending on the application's task graph and other factors like the shape of the allocation, default mapping may or may not perform well. We conduct these experiments on Blue Waters [7] (3D torus topology), Blue Gene/Q [23] (5D torus) and Catalyst (fat-tree). Note that while Blue Waters has a 3D torus topology, it may allocate non-contiguous torus partitions, where non-compute nodes are interspersed in the allocation. TopoMapping works well in this situation because it can map to any arbitrary topology. Due to space

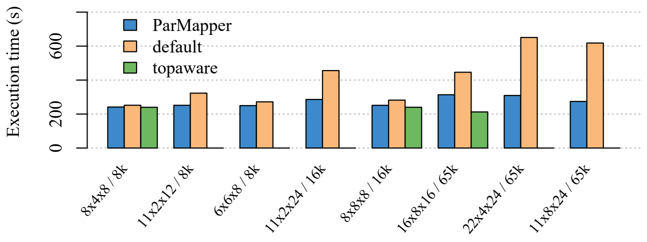

Figure 6: MILC execution time on Blue Waters with TopoMapping, topaware and default mapping. Bottom axis shows allocation shape and number of tasks.

restrictions we will show a subset of representative results that cover a wide range of applications and systems.

6.1 MILC

For MILC, CommProfiler automatically determines the best task graph to be p2p because (disregarding the global Allreduce) it accounts for almost all the communication time.

6.1.1 Blue Waters. We compare TopoMapping with the system mapping and with a mapping tool called *topaware* available on Blue Waters. This tool has been used in the past for MILC production runs on this system. Note that topaware is not an automatic tool: it needs to know how the application partitions its virtual topology, assumes regular communication within this grid and it cannot run on any arbitrary allocation shape returned by the scheduler.

Fig. 6 shows results for various allocation shapes and sizes. As we can see, topaware performs best, but it only runs on a limited number of shapes. Execution time using ParMapper is very close to topaware in most cases. ParMapper shows very good weak scaling performance regardless of number of tasks or allocation shape. The performance of system mapping depends notably on the shape of the allocation. MILC runs up to 60% faster with ParMapper than default with 16k ranks, and up to 2.2x faster with 65k ranks.

6.1.2 BlueGene/Q (Mira and Vesta). MILC has high computation to communication ratio due to the slow speed of cores; with TopoMapping, communication time is as low as 7% of the total run time in our results. With system mapping, it is 20% in the worst case. Results are shown in Fig. 7. ParMapper shows good weak scaling performance. The system mapping scheme is highly dependent on allocation shape. With 32k tasks in shape 4x4x4x8x2, it performs badly and ParMapper improves performance by 16%, while they perform similarly in the other shape. For 65k tasks, system scheme performs well in the shape we obtained, and improvement with ParMapper is smaller (5.4%). Note that ParMapper significantly improves the communication time of MILC (as seen in Fig. 7 (b)).

6.1.3 Catalyst. For many cases, we find that our scheme results in unexpectedly high reduction in communication time and total execution time. For example, we observe 88% and 76% lower total execution time for runs performed on 1024 and 512 processes, respectively. To understand the reasons for these high improvements, we analyzed the default mapping on Catalyst and found that MPI ranks are assigned to cores using a cyclic scheme, i.e. rank 0 is run on core 0 of node 0, rank 1 is run on core 0 of node 1, and

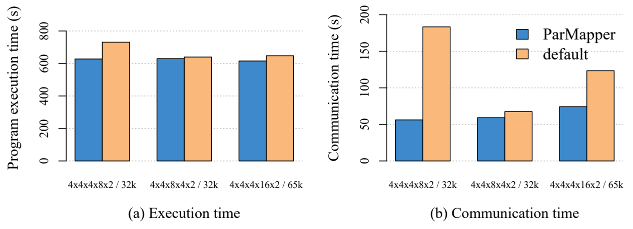

Figure 7: MILC execution time and communication time on Blue Gene/Q with TopoMapping and default mapping. Running on 32 processors per node.

so on, which is not good for an application like MILC that does near-neighbor communication. Thus, to be fair, we also compare TopoMapping results with runs that perform block mapping. In this scenario, we find that TopoMapping reduces the communication time by up to 34% while the total execution time is reduced by 14%.

6.2 Qbox

6.2.1 Blue Waters. As with MILC, the performance of the system mapping varies with allocation shape. CommProfiler determines the best graph to be sum-weighted of p2p and subcommunicator Alltoalls, which are the types where Qbox spends most of its communication time. Results are shown in Fig. 8 (a) (strong-scaled). We observe a performance increase with ParMapper of 45% with 16k ranks, and 70% with 32k ranks.

6.2.2 Catalyst. On this system, TopoMapping improves performance of Qbox by 17% with 512 processes and 28% with 1024 processes compared to the system mapping. It is interesting to note that CommProfiler also includes subcommunicator Allreduce in the task graph (whereas it did not on Blue Waters). The difference in topology (fat-tree vs torus) has an effect on the performance of the Allreduce, and CommProfiler automatically detects this.

6.3 PSDNS-CCD3D

PSDNS-CCD3D is an in-development research code for the simulation of turbulent flows. One of the its main components is the combined compact difference (CCD) [24] kernel. In these experiments we map the CCD kernel on Blue Waters, using test cases provided by authors. Most of the communication time is due to p2p and Alltoall, but CommProfiler determines the best graph to be Alltoall-only, as it allows better optimization of this collective and leads to lower execution time. Results are shown in Fig. 8 (b). TopoMapping shows better weak-scaling performance and improves the run time of the kernel by 16.6% compared to the system map with 32k tasks, and by 22.6% with 65k tasks.

6.4 pF3D

pF3D [10] is a scalable multi-physics code used for simulating laser-plasma interactions in experiments conducted at the National Ignition Facility (NIF) at LLNL. A 3D Cartesian grid is used for decomposing pF3D's domain among MPI processes. The communication performed by pF3D consist of Alltoall operations within subcommunicators defined along X and Y axes in every Z plane, and point-to-point communication across Z planes.

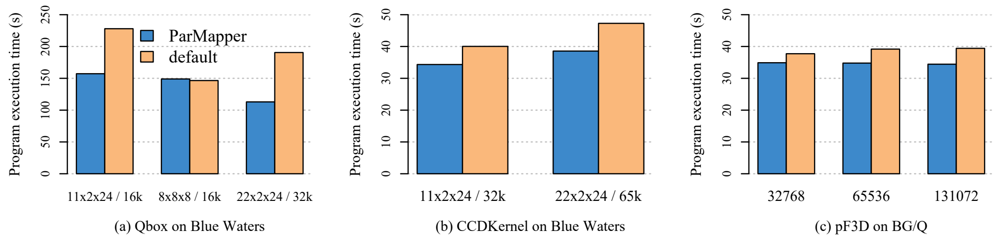

Figure 8: Execution time of Qbox, CCD kernel and pF3D with TopoMapping on different machines.

CommProfiler determines the best graph to be Alltoall-only, for the same reason as with PSDNS-CCD3D. Fig. 8 (c) shows that ParMapper reduces the total execution time of pF3D execution on a Blue Gene/Q by up to 14% when weak scaled. For pF3D, roughly 27% of time is spent in communication with default mapping. With ParMapper, the fraction of time spent in communication reduces to 18%, with a 44% drop in absolute communication time.

7 CONCLUSION AND FUTURE WORK

Task mapping is an effective way of improving the execution time of HPC applications on many systems. Improvement depends on several factors like the application, network topology and job size. In this paper, we proposed a novel task mapping system which is automatic, avoids limitations of existing schemes, and performs better in general. We evaluated our scheme with diverse applications on many systems, using no special knowledge of the applications or domain-specific techniques. Results show typical improvements in execution time ranging from 15% to 54%.

As future work, we would like to integrate the scheme into runtime systems like Charm++ [25] so that mapping is dynamically optimized for different phases of the application by migrating tasks between processors during application execution. This can also be combined with load balancing capabilities to dynamically balance workload between processors.

REFERENCES

[1] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology Mapping for Blue Gene/L Supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006.

[2] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. Mcmanus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980, pages 121–126. Springer, 1995.

[3] Abhinav Bhatele and Laxmikant V. Kale. Application-specific Topology-aware Mapping for Three Dimensional Topologies. In *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS '08)*, April 2008.

[4] 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, and Charles H. Still. Mapping Applications with Collectives over Sub-communicators on Torus Networks. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 97:1–97:11, Los Alamitos, CA, USA, 2012.

[5] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 75–84, NY, USA, 2011. ACM.

[6] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kalé. Topology-aware Task Mapping for Reducing Communication Contention on Large Parallel Machines. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 145–145, Washington, DC. IEEE Computer Society.

[7] National Center for Supercomputing Applications. Blue waters project. http://www.ncsa.illinois.edu/enabling/bluewaters.

[8] MILC Collaboration. MIMD Lattice Computation (MILC) Collaboration Home Page. http://www.physics.indiana.edu/~sg/milc.html.

[9] Francois Gygi, Erik W. Draeger, Martin Schulz, Bronis R. de Supinski, John A. Gunnels, Vernon Austel, James C. Sexton, Franz Franchetti, Stefan Kral, Christoph W. Ueberhuber, and Juergen Lorenz. Large-scale Electronic Structure Calculations of high-Z Metals on the BlueGene/L Platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[10] 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*, 7(5):2023, 2000.

[11] Mehmet Deveci, Sivasankaran Rajamanickam, Vitus J. Leung, Kevin Pedretti, Stephen L. Olivier, David P. Bunde, Umit V. Çatalyürek, and Karen Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 27–36, Washington, DC, USA, 2014.

[12] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. Design of a scalable infiniband topology service to enable network-topology-aware placement of processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 70:1–70:12, Los Alamitos, CA, USA, 2012.

[13] J. T. Vogelstein, J. M. Conroy, V. Lyzinski, L. J. Podrazik, S. G. Kratzer, E. T. Harley, D. E. Fishkind, R. J. Vogelstein, and C. E. Priebe. Fast Approximate Quadratic Programming for Graph Matching. *PLoS ONE*, 10(4), 2015.

[14] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[15] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52:133–152, 2005.

[16] Cédric Chevalier, François Pellegrini, Inria Futurs, and Université Bordeaux I. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *In Proceedings of Euro-Par 2006, LNCS 4128:243–252*, pages 243–252, 2006.

[17] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, July 1976.

[18] Abhinav Bhatele, Eric Bohm, and Laxmikant V. Kale. Optimizing communication for charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 23(2):211–222, 2011.

[19] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *The VLDB Journal*, 16(1):5–28, January 2007.

[20] Paola Festa and Mauricio G.C. Resende. *Grasp: An Annotated Bibliography*, pages 325–367. Springer US, Boston, MA, 2002.

[21] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, April 1977.

[22] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[23] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10. ACM, 2011.

[24] T. Gotoh, S. Hatanaka, and H. Miura. Spectral compact difference hybrid computation of passive scalar in isotropic turbulence. *J. Comput. Phys.*, 231(21):7398–7414, August 2012.

[25] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.