APHiD: Hierarchical Task Placement to Enable a Tapered Fat Tree Topology for Lower Power and Cost in HPC Networks

George Michelogiannakis, Khaled Z. Ibrahim, John Shalf
Lawrence Berkeley National Laboratory

Jeremiah J. Wilke, Samuel Knight, Joseph P. Kenny
Sandia National Labs

February, 2017

Work at Lawrence Berkeley National Laboratory was supported by the Director, Office of Science, of the U.S. De- partment of Energy (DOE) under Contract No. DE- AC02-05CH11231. Work was also supported by the DOE National Nuclear Security Administration (NNSA) Advanced Sim- ulation and Computing program at Sandia National Labs, managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

APHiD: Hierarchical Task Placement to Enable a Tapered Fat Tree Topology for Lower Power and Cost in HPC Networks

George Michelogiannakis, Khaled Z. Ibrahim, John Shalf
Lawrence Berkeley National Laboratory
1 Cyclotron road, Berkeley, CA 94720
{*mihelog,kzibrahim,jshalf*}*@lbl.gov*

Jeremiah J. Wilke, Samuel Knight, Joseph P. Kenny
Sandia National Labs
7011 East Ave, Livermore, CA, 94550,
{*jjwilke,sknigh,jpkenny*}*@sandia.gov*

***Abstract*—The power and procurement cost of bandwidth in system-wide networks has forced a steady drop in the byte/flop ratio. This trend of computation becoming faster relative to the network is expected to hold. In this paper, we explore how cost-oriented task placement enables reducing the cost of system-wide networks by enabling high performance even on tapered topologies where more bandwidth is provisioned at lower levels. We describe APHiD, an efficient hierarchical placement algorithm that uses new techniques to improve the quality of heuristic solutions and reduces the demand on high-level, expensive bandwidth in hierarchical topologies. We apply APHiD to a tapered fat-tree, demonstrating that APHiD maintains application scalability even for severely tapered network configurations. Using simulation, we show that for tapered networks APHiD improves performance by more than 50% over random placement and even 15% in some cases over costlier, state-of-the-art placement algorithms.**

I. INTRODUCTION

Communication has been identified as one of the top ten exascale research challenges in terms of power, performance and procurement cost [1], [2]. To maintain feasibility and remain under a proposed 20MW power cap for the system, both the network procurement and pJ/bit of data movement costs must be controlled [3], [4]. The decreasing memory per core in high performance computing (HPC) also produces more off-chip traffic, further stressing the network. This is already evident in the Blue Gene/Q which provides 3× fewer bytes per flop than the Blue Gene/L, and Cray's XT5 which provides 38× fewer bytes per flop than the XT3 [5]. Simply over-provisioning the network is not sustainable moving forward in terms of power as well as procurement cost [2].

Tapered hierarchical networks have been proposed to decrease procurement and power cost [6]. In particular, higher levels in system-wide networks often require expensive optical cables. Rather than requiring full all-to-all bandwidth, many applications have significant locality in their communication pattern. In tapered networks, more bandwidth is provisioned at lower levels, both saving cost and potentially improving performance if cost savings on top-level switches and cables are converted into extra bandwidth at lower levels.

To ensure high performance on tapered topologies, task placement can maximize the tapering of application traffic, reducing the load on top-level links [7]–[9]. Task placement refers to relocating tasks (e.g., MPI ranks) to different compute nodes to reduce average communication distance and congestion. Calculating an optimal task placement is an NP-complete problem. Practically, task placement must rely on heuristics [8]–[11]. Past work confirms the performance and cost benefits of locality- and network-aware placements on a mesh [12], [13], torus [13], [14], fat tree [14], and dragonfly [15]. In the case of a mesh, the worst-to-best ratio of application performance can reach 4× [14].

To provide a practical, scalable placement heuristic for tapered topologies, we propose APHiD - **A**lgorithm for **P**lacing processes **Hi**erarchically on **D**-ary trees. APHiD takes as input an application traffic matrix and creates the placement for arbitrary hierarchical topologies by embedding them in a d-ary tree. Fat tree topologies map most naturally to the algorithm, but other topologies like dragonfly [16] or stacked HyperX [17] are also hierarchical and have natural tree embeddings. APHiD is based on hierarchical algorithms proven to produce placements optimal for communication locality [18]. Instead of optimal graph partitioning at every step (an NP-complete problem), APHiD approximates those methods through heuristic graph partitioning algorithms. Relative to other state-of-the-art methods, APHiD reduces computational complexity by $O(E^{k-1})$ where E is the size of the communication graph and k is the tree radix. APHiD can also be implemented using parallel libraries like ParMETIS for scalability.

Overall, APHiD aims to provide more "leaf" bandwidth on leaf switches in the topology near the compute nodes where applications need it. Our work provides two major contributions. First, we introduce APHiD and how it can be practically applied to a set of HPC applications, producing placements competitive with or better than state-of-the-art but with lower computational cost. Second, we describe an implementation of a 3-level fat-tree topology with tunable tapering. Through simulation, we demonstrate how placement algorithms and tapered topologies can be combined to implement scalable HPC networks with greatly reduced power and procurement costs.

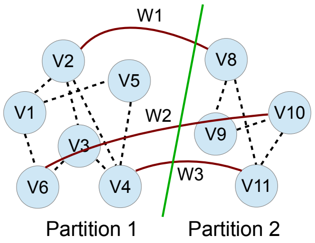


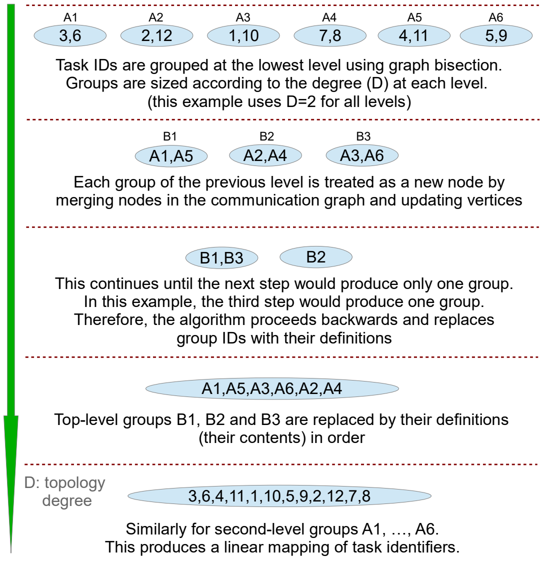
Figure 1. Graph partitioning is the problem of dividing vertices into groups (in this example two) so that the sum of the weights $W1+W2+W3+\ldots$ of the inter-partition edges is the minimum across all grouping combinations.

II. APHiD: Hierarchical Graph Partitioning and Mapping to Topologies

A. Task Placement Algorithm

We formulate our problem as a graph $G = (V, E)$ where vertices (V) represent computation tasks and directed edges (E) represent communication [18]. Each edge is weighted to represent the number of bytes sent. Using this notation, our optimization target of reducing traffic in higher levels of a hierarchical network becomes a graph partitioning problem [8], [18] where the goal is to group nodes such that the total weight of the edges crossing group boundaries is minimized [8], [18], as shown in Figure 1. To map to a fixed network topology, the algorithm is constrained to find partitions of specific sizes. For balanced networks this means partitions of equal sizes. There are two challenges to this simplistic view [8], [18]. First, calculating a provably optimal graph partition is an NP-complete problem. Second, a simple graph partition does not readily map to a multi-level hierarchical topology such as a fat tree [19].

We therefore formulate APHiD as a hierarchical algorithm [8], [18], [20]. The input to APHiD is a communication graph (equivalent to a traffic matrix), and the output is an association of task identifiers to network endpoints (task placement). The first version of APHiD begins at the lowest level and uses graph partitioning to form groups each containing D nodes, where D is the degree of the topology at the lowest level. Then it advances to the second level of the topology, but treats groups formed in the previous step as single nodes. Therefore, group A may contain tasks 1, 2, 6, and 10, but that group will still be treated as a single node in the second level. A second communication graph is generated that then disregards traffic within groups. Except for group size, the degree D also implicitly defines how many levels of hierarchy the algorithm will execute as the integer ceiling of $\log_D(Nodes)$, assuming a constant D in all levels. After APHiD reaches the top level of the topology, it proceeds backwards and unfolds the selections it created at every level with lower-level node identifiers to produce a final linear mapping of task identifiers, shown in Figure 3. This is the bottom–up approach of APHiD, because it begins from the lowest topology level. An illustration of the bottom–up APHiD is provided in Figure 2.



Task IDs are grouped at the lowest level using graph bisection.
Groups are sized according to the degree (D) at each level.
(this example uses D=2 for all levels)

Each group of the previous level is treated as a new node by merging nodes in the communication graph and updating vertices

This continues until the next step would produce only one group.
In this example, the third step would produce one group.
Therefore, the algorithm proceeds backwards and replaces group IDs with their definitions

Top-level groups B1, B2 and B3 are replaced by their definitions (their contents) in order

Similarly for second-level groups A1, …, A6.
This produces a linear mapping of task identifiers.

Figure 2. The bottom–up version of APHiD. Each level forms $Tasks/D$ groups of size D. $D = 2$ in all levels in this example.

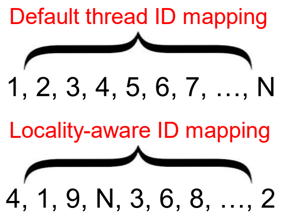

Default thread ID mapping
1, 2, 3, 4, 5, 6, 7, …, N
Locality-aware ID mapping
4, 1, 9, N, 3, 6, 8, …, 2

Figure 3. APHiD produces a linear mapping of task IDs. In APHiD's locality-aware mapping, highly-communicating tasks are mapped adjacent to each other. In this example, task 4 is placed on network endpoint 1, task 1 on endpoint 2, etc.

The above approach has been proven to produce optimal placements, contingent to each graph partition problem producing optimal partitions [18]. Unfortunately, producing an optimal partition is an NP-complete problem. Therefore, we rely on heuristic algorithms such as the recursive bisection graph [21] and K-way partitioning [22] algorithms. We find that the recursive bisection algorithm performs better than the K-way partitioning algorithm for partitioning our applications' traffic graphs in four groups or more. In other sets of graphs, K-way has been shown to outperform recursive bisection [23].

If we were to apply the bottom–up APHiD in applications with 10,000 tasks and $D = 4$, the first partitioning step would produce poor results because the algorithm would have to create 2,500 groups of four tasks in the first level. That many groups overwhelm the heuristic algorithm. We therefore employ a top–down version of APHiD for the purpose of keeping the graph partitioning problem tractable at each step. The first step is to form just D_n groups at the top level and proceed downwards, where D_n is the degree (branching) of the topology at the n^{th} level. In contrast to the bottom–up version, the top–down version uses D to define

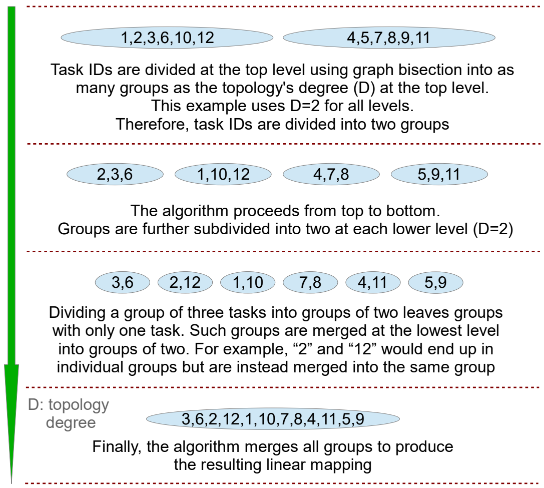

1,2,3,6,10,12
4,5,7,8,9,11
Task IDs are divided at the top level using graph bisection into as many groups as the topology's degree (D) at the top level. This example uses D=2 for all levels. Therefore, task IDs are divided into two groups
2,3,6
1,10,12
4,7,8
5,9,11
The algorithm proceeds from top to bottom. Groups are further subdivided into two at each lower level (D=2)
3,6
2,12
1,10
7,8
4,11
5,9
Dividing a group of three tasks into groups of two leaves groups with only one task. Such groups are merged at the lowest level into groups of two. For example, "2" and "12" would end up in individual groups but are instead merged into the same group
D: topology degree
3,6,2,12,1,10,7,8,4,11,5,9
Finally, the algorithm merges all groups to produce the resulting linear mapping

Figure 4. The top–down version of APHiD starts from the top level and divides tasks into D groups in each level ($D = 2$ in this example). Each group contains $Tasks/D$ tasks, in contrast to $Tasks/D$ groups of D tasks for the bottom–up version. It then proceeds to lower levels in a similar fashion, to produce the final linear mapping.

the number of groups instead of the group size. Thus, groups now contain $Tasks/D$ tasks each. In the next step, each of those D_n groups is further partitioned into D_{n-1} subgroups for $D_n \times D_{n-1}$ groups in total. This proceeds until the lowest level of the topology. This algorithm is illustrated in Figure 4. The bottom–up algorithm keeps group sizes constant, while the top-down keeps the number of groups constant.

Comparing Figure 2 to Figure 4, notice that the final linear mapping identifies pairs of tasks in the same manner (e.g., 3 and 6 are adjacent to each other in both versions). However, some pairs are in different locations, such as (2,12). In top–down APHiD, the first step of dividing tasks into D large groups at the top level of the network is more coarse-grained. Therefore, the top-level step could have easily placed group (2,12) in the other top-level group, but it made little difference for the partitioning quality at the top level. Later steps cannot change choices made in earlier stages. In contrast, the bottom–up APHiD first pairs 2 and 12 together, and then is better able to group that pair with another pair it heavily communicates with. However, as previously discussed, the first step in the bottom–up version is stressful to the heuristic graph partitioning, typically resulting in suboptimal partitions. The top–down APHiD always forms D groups at each level, avoiding this problem.

B. *Pruning*

APHiD improves the quality of heuristic placements by pruning low-weight edges in the communication graph. In an initial step, APHiD detects the maximum amount of bytes exchanged between any two tasks (the maximum weight in the communication graph). APHiD then disregards edges lower than a certain percentage of that maximum. This step helps the greedy decisions of heuristic graph partitioning algorithms by reducing the exploration space. Pruning may not always be favorable if short messages play a critical role in performance, though for our bandwidth tapering study APHiD shows good improvements for most applications. Future extensions to APHiD could also incorporate the number of messages (rather than just byte totals) in the edge weights. In Section V-D, we show how pruning improves the quality of placements and show that a 5% pruning factor is a good tradeoff choice.

C. *Mapping to Different Topologies*

APHiD applies to any hierarchical topology, including a fat tree [19] that we use in our experiments. APHiD should be configured to match the number of levels and degree of each level of the topology. Other popular topologies such as dragonfly can be modelled as a two-level topology (extending our model to more levels of a dragonfly is straightforward), where the number of nodes per group is the degree at the lowest level and the number of groups the degree at the top level. In the fat tree, the linear mapping of task identifiers (e.g., MPI ranks) produced by APHiD dictates what leaf each rank is placed at. APHiD applies to other popular topologies as well, such as the flattened butterfly where each dimension can be treated as a level of a hierarchical topology [24].

In the case where a system's job scheduler returns a fragmented allocation, APHiD may have to map to a potentially irregular topology (such as an unbalanced tree) since some compute resources may be reserved for other applications [5]. Since APHiD in the end computes a linear mapping of task identifiers, we can map this list to any topology, including irregular ones, with an extra step.

D. *Usability*

APHiD can be used as an *a priori* tool by obtaining the communication graph using static analysis [25] or by providing the programmer a framework for specifying it [26], [27]. To influence task placement, some MPI implementations provide an API to specify the communication graph [28] as well as placement of MPI ranks through text files [20], [29]. The system can record an application's traffic such that when the same application launches again we can use an improved placement, instead of executing APHiD for every single launch. APHiD can also be used for data placement decisions of application runtimes, such as adaptive mesh refinement in Boxlib [30].

When considering systems with multiple applications executing in parallel, APHiD can simply merge all application traffic matrices and thus treat them as one. Tasks from different applications will be placed far apart because they do not exchange messages. This method has more optimization opportunities that simply optimizing each application's placement within its allocated nodes, but system software must provide support for inter-application task placement. In addition, we can apply APHiD during application execution in individual phases (e.g., between barriers) and rely on task migration [7], [31]. However, this decision has to weigh the expected cost of migrating tasks against the rate of change of the traffic pattern. Even dynamic applications that do change communication patterns typically change traffic slowly [32], which does not warrant frequent placement steps. For now, though, we focus our current analysis methodology on initial placement of tasks.

E. Collective Operations

By default, collective operations are included in the communication graph by breaking down each collective operation to point-to-point messages [33], [34]. Some MPI implementations optimize their collective operations based on a default placement or without regard to placement [35]. However, underlying collective algorithms in an MPI implementation may be made more efficient for a given placement and topology (as is allowed by the MPI standard [33], [36]). Here APHiD takes the collective implementations as fixed and tries to optimize task placement for them. A more intelligent strategy may be for the collective implementation to adapt to a fixed task placement.

For some applications, the dominant collective is an MPI_Allreduce of a single double-precision scalar, which carries latency costs but effectively no bandwidth cost. For MiniDFT, many subcommunicator MPI_Alltoallv collectives as part of an FFT contribute significantly to overall traffic. For GTC, many charge reductions via MPI_Allreduce with large buffers (>8KB) also contribute significantly.

III. Tapered Fat-tree Topology

We apply placement algorithms in the context of a 3-level tapered fat-tree topology. Our proposed tapered 3-level tree is shown in Figure 5. In general, the 3-level fat tree consists of three switch types:

- Leaf: The first row of switches connected directly to the compute nodes.
- Aggregation: The second row of switches aggregating traffic from leaf switches to form disjoint subtrees.
- Core: The final row of switches connecting the disjoint subtrees formed by the aggregation switches.

For current HPC systems, a common interconnect uses 48-port Aries routers [37]. For exascale systems, a 64-port router is still feasible and cost-effective as an 8x8 tiled architecture. We assume 64-port routers in the current work, but the general methodology could also be applied to higher or lower radix routers.

The tapering in the tree refers to the bisection bandwidth between two levels of the tree. In an untapered tree, all levels have equal bisection bandwidths resulting in full all-to-all bandwidth. For a 64-port router, this means allocating 32 ports to "up" traffic and 32 ports to "down" traffic. The leaf and aggregation levels have an equal number of switches in the untapered version.

	Untapered	Tapered	0.5-Core	0.25-Core
Subtrees	8	16	16	16
Leaf	256(32)	352(22)	352(22)	352(22)
Agg	256(32)	160(10)	160(10)	160(10)
Core	128	50	25	13
Nodes	4096(16)	4224(12)	4224(12)	4224(12)

Table I
Total no. switches/nodes in fat-tree configurations having 4K nodes. Parentheses show number of switches per subtree and number of nodes per leaf switch.

	Untapered	Tapered	0.5-Core	0.25-Core
Node-Leaf	8192(32)	14784(42)	14784(42)	14784(42)
Leaf-Agg	8192(32)	7744(20)	7744(20)	7744(20)
Agg-Core	8192(32)	3200(20)	1600(10)	800(5)

Table II
Total no. links in fat-tree configurations having 4K nodes. Parentheses show links per switch. Links are per-port basis, Multiple links might connect the same switches.

In our tapered configuration, the number of switches decreases on higher levels. Additionally, each switch dedicates more links to downward traffic than upward traffic. Each fat-tree level therefore decreases in bisection bandwidth. The total number of switches and links for different tree configurations are shown in Tables I and II. In the current work, we explore three different tapering configurations.

- *Baseline*: The total bandwidth on each level decreases by roughly a factor of two. Almost double the injection bandwidth is allocated per switch relative to the untapered configuration.
- *2x taper*: Same as the baseline tapered case, but the number of aggregation-core links is cut by 1/2.
- *4x taper*: Same as the baseline tapered case, but the number of aggregation-core links is cut by 1/4.

Our topology assumes netlink blocks grouping nodes together (Figure 5). Rather than allocating individual switch ports to nodes, we share available injection bandwidth across the netlink blocks as done in the Cray XC systems.

IV. Methodology

A. Applications

We use MPI application traces that were collected to evaluate future exascale architectures from [38]. Most applications studied here are reduced versions of the parent application that mimic the main communication patterns.

- Lulesh [39]: An explicit hydrodynamics code exhibiting primarily nearest neighbor communication.
- MiniDFT: A minimal density functional theory code extracted from the Quantum Espresso package stressing all-to-all communication on MPI subcommunicators during an FFT.
- GTC [40]: A gyrokinetic toroidal code exhibiting heavy neighbor communication of particles and subcommunicator all-reduces.

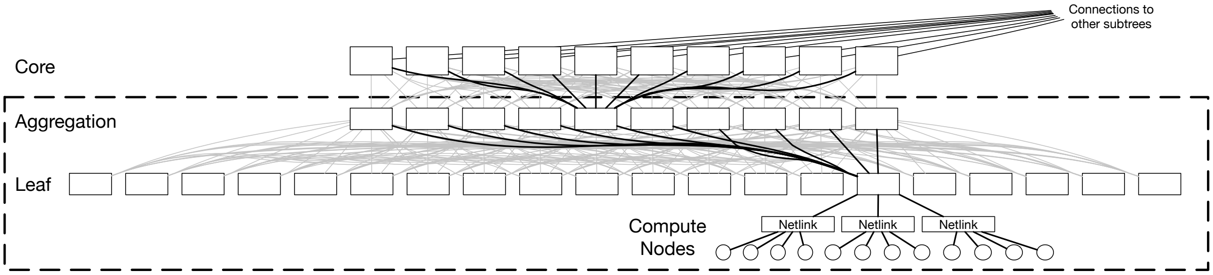


Figure 5. Tapered fat-tree topology showing connectivity of a single subtree (outlined). Leaf, aggregation, and core switches are shown. Node concentration per switch is 12, but nodes are grouped into netlink blocks to increase effective injection bandwidth. Links shown only provide connectivity and may consist of several distinct links/ports in actual implementation.

- Nekbone [41]: A subset of Nek5000 that performs primarily conjugate gradient linear solves as part of the spectral element method.
- MiniPIC: A particle-in-cell code from Sandia National Labs that combines heavy neighbor particle communication with a basic linear field solve.
- Multigrd: Extracted from BoxLib [30], implements a V-cycle multigrid solver

We expect these applications can have placements that improve on both random and linear placements. Note that no application creates global all-to-all communication, for which improved placement has limited potential. Communication maps for more applications are available in [38]. The number of ranks ranges from 1K to over 10K. Placement is performed for each application separately. APHiD and Treematch are each implemented in MATLAB. Generated output placements are passed as an input to our simulator. We use the top–down version of APHiD for all experiments because that produces higher quality placements.

B. Simulation

We choose simulations instead of real-platform experiments because a real system would provide us with fragmented placements that differ for each run, depending on other workloads on the system; this would introduce significant noise to our results. In addition, controlling application task placement, and not just task placement into nodes already allocated to an application, in many systems requires administrator access to affect the scheduler or has limited flexibility. Also, simulation allows exploring performance results on speculative exascale configurations.

We run simulations based on MPI traces collected using the DUMPI tool and replayed using macroscale network models from the Structural Simulation Toolkit (SST). [42], [43]. We report results for a fat tree sized to fit the application [19]. All simulations are performed on the three different tapered fat-tree configurations proposed in Section III. We assign one MPI rank per endpoint (compute node). Traffic uses deterministic routing with round-robin packet scattering for load-balancing. SST/macro uses a coarse-grained packet-level network model. Routing is performed on individual packets. Rather than model individual flits, however, flit-level contention is modeled using flow bandwidth approximations. The simulator uses a detailed queueing model with token control-flow for each packet. Details are available in the SST/macro manuals [44].

DUMPI traces contain timestamps for each MPI call. Time between consecutive MPI calls measures compute times. When replaying traces, the communication delays within each MPI call are simulated. Compute time can be scaled to simulate faster systems relative to the platform traces were collected on. Most traces were collected on the Cray XC30 Edison system at NERSC in an MPI-only fashion. Projecting to pre-exascale or exascale, we examine time scalings up to 100x or 1000x those from Edison. We assume effective network bandwidths of 50GB/s per link (port) are feasible, which corresponds approximately to 50% payload efficiency for a 1Tb/s link.

C. Placement Algorithm

In our experiments, we discard edges in the traffic graph with weight less than 5% of the maximum (Section V-D). We use recursive bisection, found in the METIS library for graph partitioning [45], [46]. We configure the algorithm to grow an initial bisection in a greedy manner, perform randomized heavy-edge matching, and perform one-sided node-based refinement.

Results are compared against the default (linear) placement where MPI rank i is placed at node i, as well as the version of Treematch found in appendix D of [8]. Treematch is applicable to the networks and optimization goals (cost) we consider and is a highly-regarded well-performing algorithm. To evaluate locality already exploited by default linear placement, we also define a random task placement as median cost value out of 1000 random placements. Default and random placements approximate default system placements in some production systems today, whereas Treematch represents the state of the art.

V. Evaluation

A. Bandwidth and Cost Reduction

Figure 6 shows bandwidth usage for a binary (2-ary) fat tree sized to fit each application. Although binary trees are not implemented in practice, they clearly illustrate traffic

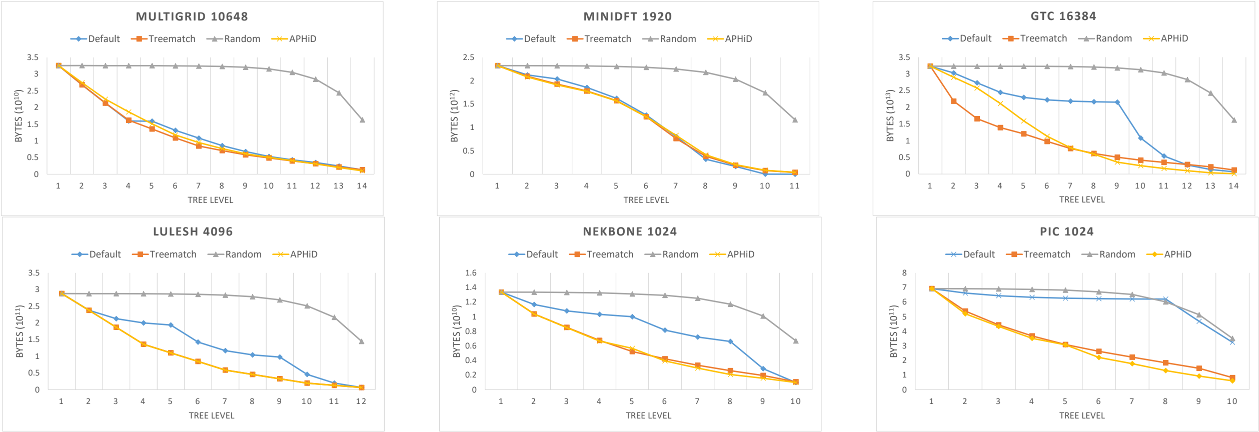


Figure 6. Bytes passing through each level of a 2-radix fat tree sized to accommodate the application.

tapering and the quality of each placement. A binary tree mapping can also be easily converted to match other hierarchical topologies. For instance, a dragonfly with groups of size eight will merge the first three levels of the binary tree. Traffic in each tree level is equivalent to hop distance for packets on a fat tree. Packets reaching a particular level n must travel up and down for a total hop distance of $2n$.

For Lulesh and MiniDFT, APHiD shows comparable cost to Treematch, but for other applications APHiD shows a range of improvements. APHiD tends to increase traffic in lower levels of the network in order to reduce traffic at the higher levels. The default linear placement is considerably better than random placement, showing the locality already inherent - partly because programmers expect this placement and shape their communication pattern. However, some applications such as MiniPIC and to some degree GTC and Nekbone are exceptions, partly because of long communication distances. Both APHiD and Treematch improve on the default placement but for MiniDFT and Multigrid the default placement already produces high locality.

The optimization of task placement directly by the programmer is typically easier when one communication pattern dominates. Lulesh, Nekbone, and GTC are examples where a main pattern exists but some additional traffic might not be fully considered by a programmer. The traffic depends either on input data or even collective algorithms. In contrast, APHiD can consider all the traffic generated by the application. The involvement of multiple patterns, complexity of the pattern, or not having a single dominant pattern suggests better placement strategies like APHiD should be used.

B. Cost Model

For the fat-tree topologies in Section III, we can consider both procurement cost and power. We focus specifically on the network link (cable) costs between switches since tapered configurations primarily reduce the number of optical cables between aggregation and core switches. We use rough estimates of $100 for short-range electrical cables and $600 for medium-range optical cables based on current Mellanox Infiniband prices. For current systems [2], copper links at intra-rack distances consume approximately 3× less energy per bit compared to optical links (10-12 pJ/bit compared to 30-60 pJ/bit) [47]. We attempt rough exascale estimates based on previously reported extrapolations [4], [47]. For short-range electrical cables, we assume roughly 5pJ/bit at 1Tb/s for electrical and 10pJ/bit at 1Tb/s for optical cables (link + transceiver). Results are summarized in Table III.

	Cost ($)	Power (kW)
Untapered	5.73M	123
Baseline Tapered	2.69M	71
0.5x Core-Tapered	1.73M	55
0.25x Core-Tapered	1.25M	47

Table III
CABLING COST (DOLLARS) AND POWER (MW) FOR INTER-SWITCH LINKS IN FAT-TREE CONFIGURATIONS DESCRIBED IN SECTION III

C. APHiD Execution Cost

The complexity of the recursive bisection algorithm we use is $O(E \log k)$ for E edges and k partitions [22], [48]. For a hierarchical topology with N levels, a degree of k at each level, and E edges in the traffic graph, the overall complexity of APHiD is therefore no greater than $O(E \log k \log N)$. This is for both bottom–up and top–down versions. In contrast, Treematch has a complexity of $O(E^k)$ for each execution of "GroupProcesses", thus $O(E^k \log N)$ overall [8].

To illustrate the effects of higher complexity, Table IV shows the execution time required to compute the APHiD and Treematch placements, using the speed-optimized version of Treematch [8]. Execution time does not include I/O time to load the traffic matrix and record results. As shown, top–down APHiD is 59% faster on average compared to Treematch. Also, execution time is insignificant for smaller-scale applications, but ranges to tens of minutes for large applications. Execution time is a function of the number of groups at each step of APHiD. This results in top–down APHiD being faster, because bottom–up APHiD has to form $\frac{13824}{4} = 3456$ groups in the first step in the case of algebraic

Application	Top–down APHiD	Treematch
Multigrid 10648	12.81	40.96
MiniDFT 1920	0.36	0.62
GTC 13384	12.81	34.17
Lulesh 4096	12.93	20.3
Nekbone 1024	0.18	0.15
PIC 1024	0.18	0.15

Table IV
APHiD execution time (minutes), using MATLAB on a single Intel 'Haswell' 2.3-GHz core. Top–down APHiD is on average 59% faster than Treematch.

multigrid solver (AMG) 13824.

These numbers would differ for an implementation using a native programming language such as C++, but we use them as a relative comparison to illustate the increased complexity. Native language implementations have been observed to produce approximately a 500× speedup over MATLAB [49]. Since APHiD relies on METIS partitioning algorithms, it can also be easily accelerated using ParMETIS. As shown by the complexity analysis and runtime results, APHiD has a low overhead and thus is realistic to apply at runtime.

D. Traffic Graph Pruning

To motivate our choice of a 5% pruning factor, Figure 7 shows how pruning affects the number of bytes in the top level of the binary fat tree used in Section V-A. A pruning factor of 5% refers to disregarding any edges in the application's communication graph that have a lower weight than 5% of the maximum weight in the graph. Lulesh is indifferent for pruning up to 50% and is thus not shown.

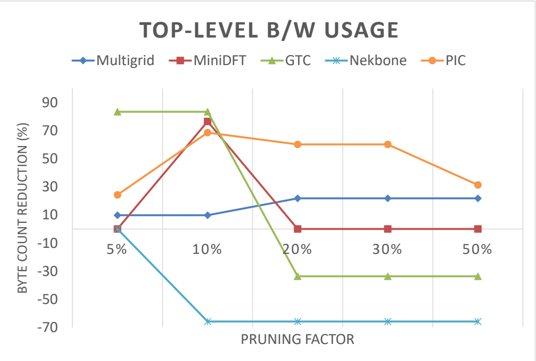


Figure 7. Reduction (%) in the number of bytes at the top level of the fat tree based on the pruning factor, compared to 0% pruning.

To begin with, we note the substantial benefit (up to 83% for GTC and 76% for MiniDFT) pruning can have. We notice that all applications perform best for 5% or 10% pruning. We choose 5% because it provides the largest benefit on average, and because it avoids penalizing Nekbone. In Nekbone, a few src-dst pairs exchange several MBs while many src-dst pairs exchange 10s or 100s of KBs. Many entries with less than 5-10% of the maximum, in aggregate, account for a large volume of traffic. Nekbone is therefore an example where pruning more aggressively than 5% disregards useful edges in the communication graph. The same is true for GTC and PIC above 10% pruning. Multigrid shows the reverse behavior, but the improvement beyond 10% pruning is only 10%. Even though we choose a single pruning factor for all applications, these results motivate choosing different factors for different applications.

Given that the complexity of APHiD is no greater than $O(E \log k \log N)$ where E is the number of edges, pruning essentially lowers the complexity of APHiD by reducing the number of edges in the graph. In our MATLAB implementation of APHiD running on a single Intel 'Haswell' 2.3-GHz core, pruning reduces the algorithm's execution time by 2% to 20%. Therefore, pruning not only lowers the complexity, but also increases the quality of the resulting graph partitions.

E. Simulation Results

We examine the performance of the placement algorithms on the three different tapered fat-tree topologies from Section III. For each application, we assume a fixed network configuration and speed. To assess the effect of tapering and placements, we accelerate the compute nodes from 1x to 1000x. If speedup relative to baseline is not proportional to the compute node speedup, it indicates the network is "under-provisioned" for the combination of placement and compute capability. Figure 8 shows both raw speedups and parallel efficiencies, e, computed as

$$e = \frac{T(n)}{nT(1)} \tag{1}$$

where n is the compute node acceleration, $T(1)$ is the baseline time with $n = 1$ and $T(n)$ is the time with n-fold compute acceleration. An efficiency of 1.0 (100%) indicates the network is sufficiently provisioned for the compute.

Lulesh maintains performance even for a 4x-tapered core and 100x speedup. For 1000x, parallel efficiency dips significantly. However, APHiD enables 75% efficiency even in this configuration, far better than than Treematch at 60% and the linear placement at 50%. The 2x core taper shows nearly equivalent performance to the baseline configuration. Some performance degradation is seen for 4x core taper relative to 2x at 1000x compute acceleration. Thus whether a 4x tapering configuration is feasible will depend critically on the compute/communication speed ratio. MiniPIC shows similar results to Lulesh, with the 2x core tapering showing scalable performance and APHiD improving performance slightly over Treematch and significantly over linear and random placement. For GTC, Treematch slightly outperforms APHiD. In general, GTC is even more strongly communication-bound than Lulesh or MiniPIC. Thus, even for the baseline topology, 100x speedups already show a dip in parallel efficiency. Interestingly, though, the 2x-tapered topology with placement nearly matches the baseline case showing the importance of placement on tapered networks.

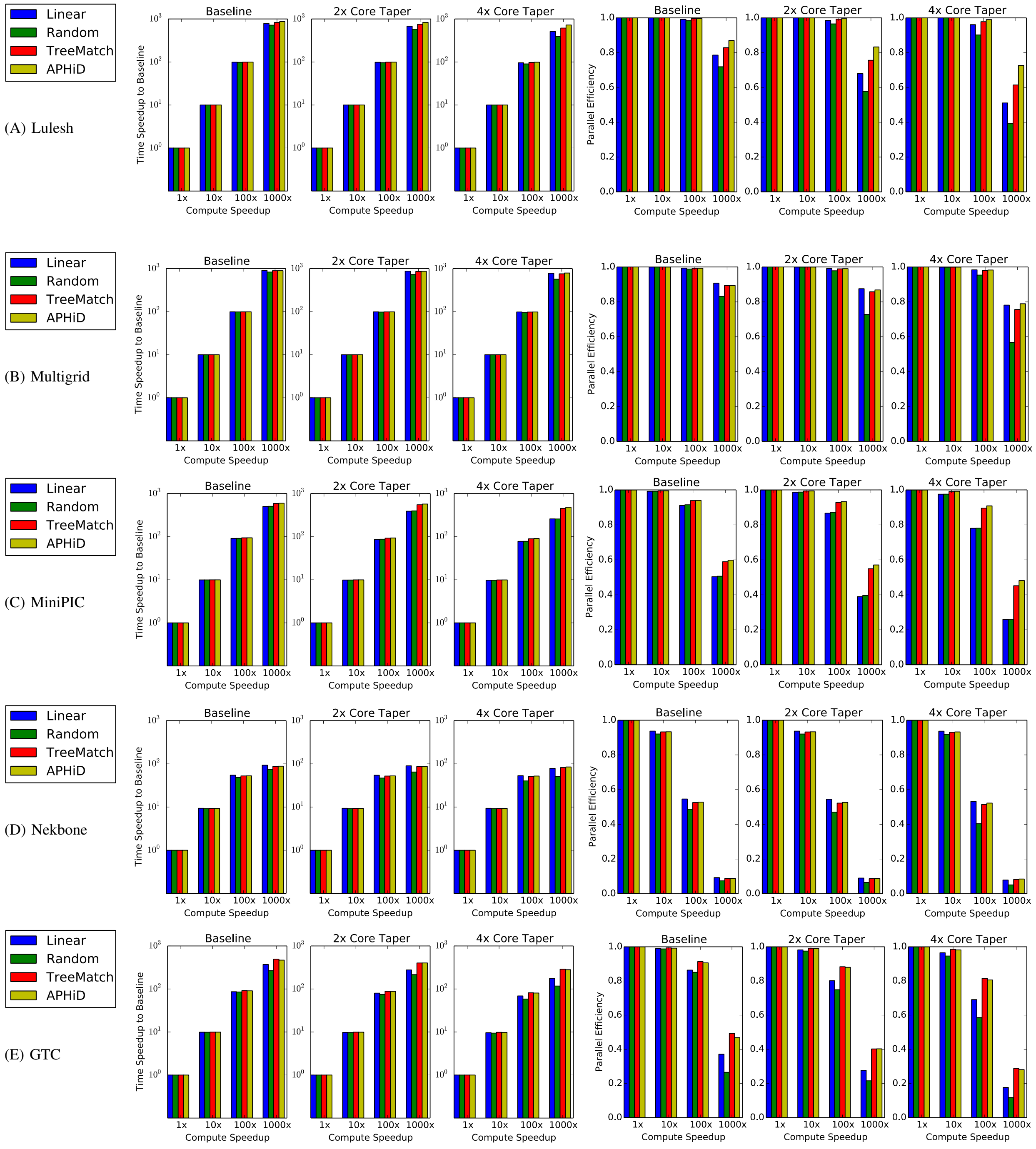


Figure 8. Simulation results show speedups as individual compute node speeds are increased for a fixed network configuration. Parallel efficiencies demonstrate whether the network is over- or under-provisioned for the compute. Results are shown for (A) Lulesh 10,648 ranks, (B) Multigrid 10,648 ranks, (C) MiniPIC 1,024 ranks, (D) Nekbone 1,024 ranks and (E) GTC 16,384 ranks. As shown, APHiD allows most applications to perform at above 90% of idealized speedup even as nodes are accelerated past 100x current baseline.

Like Nekbone (see below), GTC likely has other communication factors than top-level bandwidth limiting scalability.

Multigrid shows different results. In these cases, linear placement already performs very well leaving little room for improvement from optimized placement. Again, the baseline and 2x tapered fat-tree configurations maintain good performance while the 4x tapered configuration starts to show notable performance dips, demonstrating the limits of tapering that can be achieved before performance suffers.

Nekbone results are not scalable on any of the fat tree configurations nor with any of the placements. A basic sensitivity analysis (not shown) indicates that at exascale bandwidths (50GB/s), the limiting factor is actually hop latency and credit latency in the network rather than any bandwidth limitation. The Nekbone case will require further study beyond the scope of the current work. However, it illustrates an important case where considerations other than bandwidth will be critical to performance.

Further work will consider multi-job workloads. Since APHiD pushes traffic into leaf switches and out of aggregation/core switches (where job interference occurs), APHiD should greatly improve contention between jobs. However, further study is required to quantify this benefit.

VI. Related Work

Even though past work typically focuses on performance instead of cost, a placement strategy's objective functions can optimize for load balancing [7], communication cost [8], [9], [50] real-time processing [9], or a combination [51]. Past work has examined placement algorithms that are aware of the underlying hardware architecture, such as heterogeneity and the memory structure [51], computing capacity of each resource [52], proximity of resources [53], or the network topology [15], [54]. Other work predicts network congestion and performance in advance of execution [55]. Routing algorithm aware hierarchical task mapping (RAHTM) [20] and TreeMatch [8] have similar goals with our work, but take a different approach. Treematch uses a similar bottom–up hierarchical structure, but uses greedy metrics to find optimal groupings at each step. In particular, the speed-optimized version we compared against divides edges of the communication graph into eight bins in increasing weight order, and then sorts each bin. It then iterates through the edges starting from the largest weights and places that edge's nodes in the same group. On the other hand, RAHTM has only been applied to n-ary cubes.

Other studies have considered either tapered fat trees directly [6] or examined cabling configurations and system performance in hierachical (dragonfly) networks [56], [57]. Rather than placing tasks, other studies have examined how bandwidth steering via optical switches can improve performance in hierarchical networks [32], [58].

VII. Conclusions

In the path to exascale, the power consumption and procurement cost of bandwidth in system-wide networks will be a primary constraint. Considering that the trend of computation scaling faster than network bandwidth is expected to hold, this threatens to make network communication a primary performance and cost bottleneck in the future. Bandwidth over-provisioning will not be feasible. In this paper, we exploit cost-oriented task placement to reduce bandwidth at the higher (and most expensive) levels of hierarchical topologies. To this end, we describe APHiD, a hierarchical task placement algorithm that uses graph partitioning to group highly communicating tasks together. To make APHiD practical, we advance the state of the art by investigating how to increase the quality of the heuristic solvers at each step of APHiD, such as by disregarding low-weight edges of the communication graph (pruning) and choosing between top–down and bottom–up executions.

Our ultimate goal is to demonstrate the effect of placement algorithms on heavily-tapered networks, in this case a tapered 3-level fat tree. APHiD steers traffic towards "leaf" switches, reducing cost in higher levels of the tree. For most applications, placement algorithms (and APHiD in particular) help maintain scalable performance on tapered networks even as compute nodes are accelerated to 1000x of today's systems. For some applications, APHiD improves performance by more than 50% relative to random placement, 38% relative to linear placement, and even by 15% relative to state-of-the-art Treematch for Lulesh. Overall, applications still perform well even with severe tapering of the fat-tree core, suggesting cost without significant performance loss. However, intelligent placement is critical to shifting the scaling curve so that performance remains proportional to compute node speed.

Acknowledgments

Work at Lawrence Berkeley National Laboratory was supported by the Director, Office of Science, of the U.S. Department of Energy (DOE) under Contract No. DE- AC02-05CH11231. Work was also supported by the DOE National Nuclear Security Administration (NNSA) Advanced Simulation and Computing program at Sandia National Labs, managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

References

[1] ASCAC Subcommittee, "Top ten exascale research challenges," Office of Science, U.S. Department of Energy, Washington, D.C., Tech. Rep. SAND2015-1027, 2014.

[2] M. Al-Fares *et al.*, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008.

[3] T. Hoefler, "Software and hardware techniques for power-efficient HPC networking," *Computing in Science Engineering*, vol. 12, no. 6, pp. 30–37, Nov 2010.

[4] S. Rumley *et al.*, "End-to-End Modeling and Optimization of Power Consumption in HPC Interconnects," in *ICPPW: International Conference on Parallel Processing Workshops*, 2016.

[5] A. Bhatele and T. Gamblin, "OS/Runtime challenges for dynamic topology aware mapping," ser. ExaOSR, 2012.

[6] E. A. León *et al.*, "Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree," in *Supercomputing*, 2016.

[7] J. Paudel *et al.*, "Hybrid parallel task placement in X10," ser. X10 '13, 2013, pp. 31–38.

[8] E. Jeannot *et al.*, "Process placement in multicore clusters:algorithmic issues and practical techniques," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 4, pp. 993–1002, April 2014.

[9] S.-Y. Lee and J. Aggarwal, "A mapping strategy for parallel processing," *Computers, IEEE Transactions on*, vol. C-36, no. 4, pp. 433–442, April 1987.

[10] A. Rosenberg, "Issues in the study of graph embeddings," ser. Lecture Notes in Computer Science, H. Noltemeier, Ed. Springer Berlin Heidelberg, 1981, vol. 100, pp. 150–176.

[11] C. Gotsman and M. Lindenbaum, "On the metric properties of discrete space-filling curves," ser. IAPR, Oct 1994.

[12] A. Bhatele and L. V. Kalé, "Benefits of topology aware mapping for mesh interconnects," *Parallel Processing Letters*, vol. 18, no. 4, pp. 549–566, 2008.

[13] A. Bhatelé *et al.*, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," ser. ICS '09, 2009, pp. 110–116.

[14] J. Navaridas *et al.*, "Effects of job and task placement on parallel scientific applications performance," ser. PDP, Feb 2009, pp. 55–61.

[15] B. Prisacari *et al.*, "Efficient task placement and routing of nearest neighbor exchanges in dragonfly networks," ser. HPDC '14, 2014, pp. 129–140.

[16] J. Kim *et al.*, "Technology-driven, highly-scalable dragonfly topology," ser. ISCA, 2008, pp. 77–88.

[17] J. H. Ahn *et al.*, "HyperX: Topology, routing, and packaging of efficient large-scale networks," ser. SC '09, 2009.

[18] J. Traff, "Implementing the MPI process topology mechanism," ser. SC '02, Nov 2002, pp. 28–28.

[19] C. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *Computers, IEEE Transactions on*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[20] A. H. Abdel-Gawad *et al.*, "RAHTM: Routing algorithm aware hierarchical task mapping," ser. SC '14, pp. 325–335.

[21] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," ser. SC '95, 1995, pp. 28–28.

[22] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, Jan. 1998.

[23] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM J. Sci. Comput.*, vol. 18, no. 5, Sep. 1997.

[24] J. Kim *et al.*, "Flattened butterfly: a cost-efficient topology for high-radix networks," ser. ISCA, 2007.

[25] A. McPherson *et al.*, *Static Approximation of MPI Communication Graphs for Optimized Process Placement*, ser. LCPC, 2014.

[26] J. Hursey and J. M. Squyres, "Advancing application process affinity experimentation: Open MPI's LAMA-based affinity interface," ser. EuroMPI '13, 2013, pp. 163–168.

[27] M. J. Rashti *et al.*, "Multi-core and network aware MPI topology functions," ser. EuroMPI'11, 2011, pp. 50–60.

[28] T. Hoefler *et al.*, "The scalable process topology interface of MPI 2.2," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 4, pp. 293–310, Mar. 2011.

[29] G. Almasi *et al.*, "Implementing MPI on the BlueGene/L Supercomputer," ser. Euro-Par, 2004, pp. 833–845.

[30] Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory. Boxlib. [Online]. Available: https://ccse.lbl.gov/BoxLib/

[31] T. Hoefler *et al.*, "An overview of process mapping techniques and algorithms in high-performance computing," in *High Performance Computing on Complex Environments*. Wiley, Jun. 2014, pp. 75–94. [Online]. Available: https://hal.inria.fr/hal-00921626

[32] S. Kamil *et al.*, "Reconfigurable hybrid interconnection for static and dynamic scientific applications," ser. CF '07, 2007.

[33] A. Bhatele *et al.*, "Mapping applications with collectives over sub-communicators on torus networks," ser. SC '12, 2012.

[34] B. Prisacari *et al.*, "Bandwidth-optimal all-to-all exchanges in fat tree networks," ser. ICS '13, 2013, pp. 139–148.

[35] R. Thakur *et al.*, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[36] T. Hoefler *et al.*, "Implementation and performance analysis of non-blocking collective operations for mpi," ser. SC '07.

[37] B. Alverson *et al.*, "Cray XC Series Network," Tech. Rep.

[38] Characterization of the DOE Mini-apps. (2014) http://portal.nersc.gov/project/cal/designforward.htm.

[39] R. Hornung *et al.*, "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[40] S. Ethier *et al.*, "Petascale parallelization of the gyrokinetic toroidal code," ser. VECPAR 2010, 2010.

[41] Paul Fischer and Katherine Heisey. Proxy-apps for thermal hydraulics. [Online]. Available: https://cesar.mcs.anl.gov/content/software/thermal_hydraulics

[42] H. Adalsteinsson *et al.*, "A simulator for large-scale parallel computer architectures," *International Journal of Distributed Systems and Technologies*, vol. 1, no. 2, pp. 57–73, Apr. 2010.

[43] H. Adelsteinson and J. Wilke. (2016) DUMPI. [Online]. Available: https://github.com/sstsimulator/sst-dumpi

[44] J. Wilke and J. Kenny. (2016) SST/macro GitHub. [Online]. Available: https://github.com/sstsimulator/sst-macro

[45] G. Karypis and V. Kumar, "METIS – unstructured graph partitioning and sparse matrix ordering system, version 2.0," University of Minnesota, Tech. Rep., 1995.

[46] G. Karypis and V. Kumer, "Multilevel k-way hypergraph partitioning," ser. DAC '99, 1999, pp. 343–348.

[47] J. Shalf *et al.*, "Exascale computing technology challenges," ser. VECPAR, vol. 6449. Springer, 2010, pp. 1–25.

[48] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[49] T. Andrews, "Computation Time Comparison Between Matlab and C++ Using Launch Windows," Tech. Rep., 2012.

[50] E. Jeannot and G. Mercier, "Near-optimal placement of MPI processes on hierarchical NUMA architectures," ser. Euro-Par'10, 2010, pp. 199–210.

[51] G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for MPI applications in multicore environments," ser. PVM/MPI meeting, 2009, pp. 104–115.

[52] D. Barthou and E. Jeannot, "SPAGHETtI: Scheduling/placement approach for task-graphs on HETerogeneous architecture," 2014, pp. 174–185.

[53] M. Deveci *et al.*, "Exploiting geometric partitioning in task mapping for parallel computers," ser. IPDPS '14, 2014.

[54] A. Bhatele; *et al.*, "Optimizing communication for Charm++ applications by reducing network contention," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 211–222, Feb. 2011.

[55] T. Agarwal *et al.*, "Topology-aware task mapping for reducing communication contention on large parallel machines," ser. IPDPS'06, 2006, pp. 145–145.

[56] A. Bhatele *et al.*, "Analyzing network health and congestion in dragonfly-based supercomputers," in *IPDPS '16: International Parallel & Distributed Processing Symposium*, 2016.

[57] T. Groves *et al.*, "(SAI) Stalled, Active and Idle: Characterizing Power and Performance of Large-scale Dragonfly Networks," in *Cluster*, 2016.

[58] K. Wen *et al.*, "Flexfly: Enabling a reconfigurable dragonfly through silicon photonics," in *Supercomputing*, 2016.