SAND2013-9853C

Task Mapping Stencil Computations for Non-Contiguous Allocations

David P. Bunde
Knox College
dbunde@knox.edu

Johnathan Ebbers
Knox College
jebbers@knox.edu

Stefan P. Feer
3M Health Information Systems
sfeer@mmm.com

Vitus J. Leung
Sandia National Labs
vjleung@sandia.gov

Nickolas W. Price
Knox College
nprice@knox.edu

Zachary D. Rhodes
Allstate Corporation
rhodesz87@gmail.com

Matthew Swank
Knox College
mswank@knox.edu

Abstract

We examine task mapping algorithms for non-contiguously allocated parallel jobs, such as those on Cray X systems. Several studies have shown that task placement affects job running time for both contiguously and non-contiguously allocated jobs. Traditionally, work on task mapping either uses a very general model where the job has an arbitrary communication pattern or assumes that jobs are allocated contiguously, completely isolating them from each other. Between these cases is mapping for non-contiguous jobs having a specific communication pattern. We apply novel and adapted task mapping algorithms to this setting and evaluate them using experiments and simulations. Our focus is on jobs with a stencil communication pattern. We evaluate them with a miniApp whose communication behavior mimics CTH, a shock physics application with this pattern. Our strategies improve its running time by as much as 35% over a baseline strategy. Furthermore, this improvement increases markedly with job size, demonstrating the importance of task mapping as systems grow.

Categories and Subject Descriptors D. Software [*D.1 Programming Techniques*]: D.1.3 Concurrent Programming

Keywords Task mapping, stencil communication pattern, shock physics, parallel jobs, non-contiguous allocation, improved scalability, Cray X systems

Sandia National Laboratories is a multi-program laboratory operated and managed by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1. Introduction

This paper focuses on improving the performance of parallel jobs by optimizing the placement of their tasks. This problem is called *task mapping* because the tasks are being mapped to processing elements. When using MPI, it is equivalent to reordering the tasks so that each MPI rank works on the desired task. This problem has a long history (eg. [14]) and parallel algorithms used to be designed for specific architectures so that the task placement could be specified. This changed in the mid-1980s, when the adoption of wormhole routing made task mapping less important by making a message's latency independent of its size. Several generations of machines were made and used with little concern for task mapping.

Now it appears that this hiatus is ending. Several recent experiments have shown that task placement can significantly impact performance on modern systems (e.g. [5, 9, 12, 16, 18, 20, 22, 24]). These experiments include actual applications, one of which exhibited a speedup of 1.64 times when the task mapping was improved [20]. The issue now is contention for limited bandwidth. Since a message consumes part of the capacity of each link along its route, poorly placed tasks mean wasted bandwith. This has always been the case, but this fact's importance is being fueled by two ongoing trends. First of all, processors continue improving faster than networks, increasingly making bandwidth the limiting factor in performance. In addition, node counts in state of the art HPC systems have continued to grow, increasing both the number of hops between nodes and the potential for hotspots.

Growing recognition of the importance of task placement has led to a resurgence of work on the problem. Broadly speaking, prior work on task mapping falls into two categories, graph-based approaches and whole-machine approaches. Graph-based approaches are too general. The problems are hard, and the solutions do not exploit the regular structure of some common communication patterns. Whole-machine approaches are targeted at systems such as Blue Gene that allocate contiguous groups of nodes that are isolated from each other. These assume structured communication patterns that fold and stretch one grid into another. Such algorithms cannot be directly applied to systems which use non-contiguous allocation, such as those from Cray.

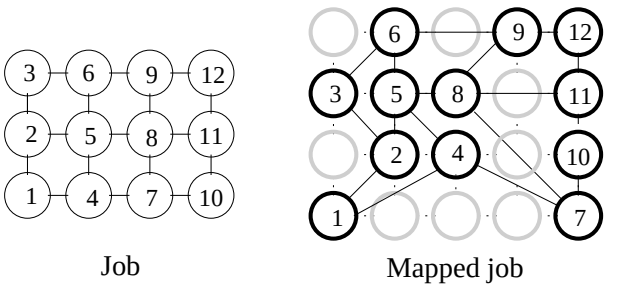

Figure 1. Possible mapping of a 4×3 job onto a 5×4 machine. Left: Job tasks with communication pattern shown. Right: Tasks mapped to dark nodes.

1.1 Contribution

In this paper, we begin the study of task mapping for jobs with structured communication patterns and non-contiguous allocations. Specifically, we look at mapping jobs that communicate in a regular 3D nearest neighbor pattern onto a 3D mesh with xyz routing. This is the simplest possible case, but non-trivial because nodes allocated to other jobs interfere with the mapping. Figure 1 shows a good 2D mapping of a 4×3 job onto 12 allocated nodes of a 5×4 machine. The solid lines connecting allocated nodes show which pairs of nodes communicate; actual communication must use the mesh edges (shown as dotted lines).

We propose new algorithms and also adapt some algorithms previously proposed for the contiguous setting. We also propose a general preprocessing step that rotates jobs so their aspect ratio more closely matches that of the allocated nodes.

We evaluate our algorithms using experiments on Cielo [25], number 22 on the June 2013 Top500 list [3]. It is a Cray XE6 organized as a 3D torus and which uses non-contiguous allocation. The application we run is miniGhost [8], a code whose structure is modeled on the computational core of CTH [21], from which it inherits nearest neighbor as its dominant communcation pattern.

One of our algorithms, which recursively divides both the task graph and the set of ranks, is shown to greatly outperform the other algorithms, including the current mapper on Cielo. This algorithm divides based on coordinates rather than using graph partitioning, exploiting the structured nature of the application for both mapping speed and quality.

In addition, our experiments show that the average number of hops between communicating tasks is strongly correlated with the running time, facilitating later simulator-based studies by others. Based on this, we perform some simulations that obtain results consistent with the experiments and also begin a comparison of allocation algorithms.

1.2 Organization of the paper

The rest of the paper is organized as follows. We summarize related work in Section 2. In Section 3, we explain our algorithms. Section 4 describes the results of our experiments. Section 5 describes simulations used to evaluate the algorithms on a variety of traces. Finally, in Section 6 we summarize our results and discuss future work.

2. Motivation and related work

As mentioned above, previous work on task mapping falls into two main categories. The first category was introduced by Bokhari [14] in the first paper on task mapping. In this category, the job is represented as a graph whose vertices are the tasks and whose edges represent communication between their endpoints. Similarly, the machine (or the available part of it) is represented as a graph of ranks with connections. In general, both graphs have weighted edges to represent the amount of communication needed between pairs of tasks and the cost of communicating between pairs of ranks respectively. Since the problem is clearly NP-complete (formally shown in [22]), the papers generally use heuristic techniques such as genetic algorithms [17] and simulated annealing [15]. Since the graph formulation of task mapping is the most general, it remains a goal for recent researchers. Chung et al. [18] use a hierarchical approach to simplify the mapping problem to reasonable complexity. Hoefler and Snir [22] use a combination of heuristics: greedy, recursive bisection, a spectral method, and a local search scheme.

In the worst case, the graph representation of task mapping is necessary since jobs can have arbitrary communication patterns and failures or interference from other jobs can complicate networking. The generality of the model obscures some practical simplifications, however: both the network and the job are likely to exhibit useful structure. HPC systems have highly structured topologies, such as a mesh or fat-tree. Similarly, jobs often communicate in regular patterns such as trees and stencil patterns.

The other main category of work on task mapping focuses on mapping mesh communication patterns onto meshes, a restriction we adopt as well. The difference is that this work implicitly assumes that the entire machine is devoted to a single job. (This occurs for capability jobs that use the entire machine, but is also the norm on BlueGene systems, which guarantee each job its own submesh, which is kept isolated from other jobs [7].) For example, Yu et al. [32] devise strategies based on folding one mesh into another with a minimum of dilation (stretching a communication graph edge across multiple communication links). Several other heuristics for the mesh to mesh mapping problem are proposed by Bhatelé et al. [13]; we adapt some of their heuristics and evaluate them with non-contiguous allocations. (The adaptation, detailed in the next section, generalizes the idea to handle non-contiguous allocations, switches how it handles the third dimension, and adds a preprocessing step.)

Prior to our work, the task mapping algorithm used by ALPS, Moab, and miniGhost was a simple linear strategy that works with a non-contiguous allocation while avoiding the complexity of the fully general approaches. Instead, it first assigns a number to each MPI rank, assigning the ranks of each node consecutive numbers and visiting the nodes in the order given by the allocator, which uses a space-filling curve similar to the linear scheme described in Section 5; see [4] for details. It then orders tasks according to row-major order (consecutive numbers move in the x direction, then start the next row by increasing y (jumping back to $x = 0$) and eventually increasing the z coordinate). Given these two orderings, each task is assigned to its corresponding rank. We call this algorithm BASELINE.

Separate from our effort, Barrett et al. [9] developed an improvement over BASELINE, which they also tested using the miniGhost application on Cielo. They noticed that miniGhost did not scale well above four thousand processes and changed the task mapping to assign $2 \times 2 \times 4$ submeshes of the job's tasks onto each sixteen-core node of Cielo. The groups of tasks were numbered consecutively in an x major fashion and mapped onto nodes numbered in the allocation order. We call this algorithm GROUPING. It reduced both the average number of hops traversed by messsages and the job processing time. Brown et al. [16] performed similar work on the more general processors command for LAMMPS [1].

3. Our Algorithms

Now we describe our task mapping algorithms. Our examples assume one rank per node and one node per XYZ coordinate value,

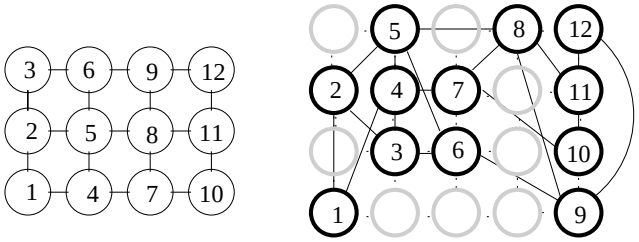

Figure 2. Mapping by COLMAJOR.

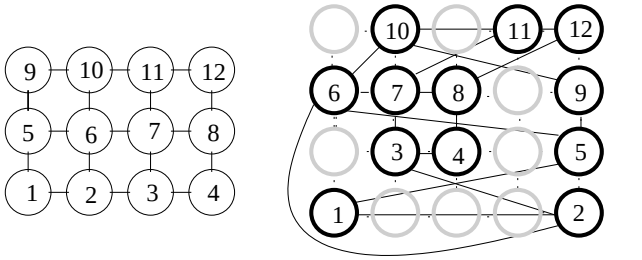

Figure 3. Mapping by ROWMAJOR.

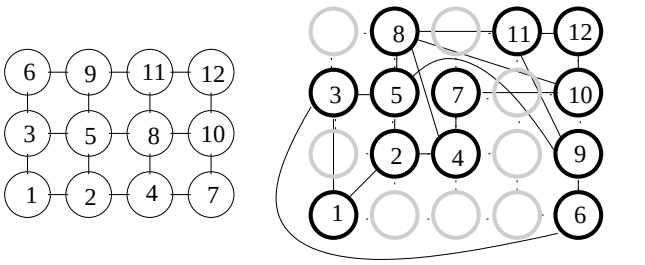

Figure 4. Mapping by CORNER.

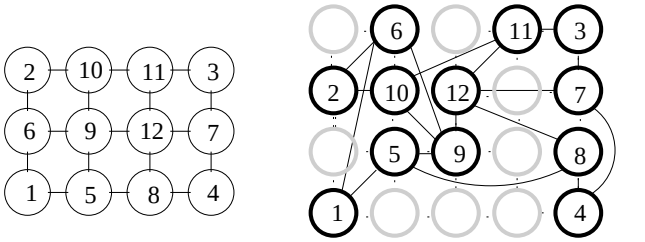

Figure 5. Mapping by ALLCORNERS.

but the algorithms also work when ranks share coordinates (as in our experiments).

Rotations As a preprocessing step, our algorithms rotate the job if doing so makes its aspect ratio match more closely with the allocated ranks. Specifically, we compare the relative orders of dimension lengths for the job and the bounding box of the set of ranks. If these orders differ, we rotate the job to make the orders match. For example, if the job has a longer x dimension while the bounding box has a longer y dimension, we rotate the job so that both have a longer x dimension.

This rotation step is performed for all algorithms we consider except BASELINE and GROUPING, which represent prior work.

Linear algorithms The first set of algorithms we considered are *linear algorithms*, in which a linear ordering is used to assign tasks to ranks. These algorithms are fast and easily implemented, making them attractive as a starting point. Naturally enough, both BASELINE and GROUPING fall into this category.

Our first new algorithm is COLMAJOR, in which both the tasks and the ranks are numbered in column major order (coordinates increase first in y, then x, and finally in z). Each task is assigned to its corresponding rank. This algorithm in 2D is illustrated in Figure 2.

The algorithm ROWMAJOR is the same as COLMAJOR except that the numberings are done in row major order, as illustrated in 2D in Figure 3. ROWMAJOR differs from BASELINE except for the latter's use of a space-filling curve for the allocation order.

We also proposed a novel algorithm ORDERED, which extends COLMAJOR and ROWMAJOR by trying multiple linear orderings. Specifically, it considers row- and column-major orderings plus their "flips", where the dimensions are traversed in the opposite direction. (For example, using a row major ordering, but traversing the rows right to left instead of left to right.) In two dimensions, there are $2 \cdot 2^2 = 8$ such orderings. ORDERED compares all these orderings and takes the one that yields the lowest average hops. (As we show later, this metric is highly correlated with running time.)

Corner-based algorithms Our next set of algorithms are called *corner-based algorithms* since they build mappings from the corners rather than the sides. The first of these is the CORNER algorithm. As in the linear algorithms, CORNER numbers both the tasks and ranks, mapping each task to the corresponding rank. The ordering it uses is distance from $(0, 0, 0)$, with ties broken by the z coordinate, then the y coordinate, and finally the x coordinate. This algorithm in 2D is illustrated in Figure 4.

The ALLCORNERS algorithm is similar except that it rotates between the corners. Thus, its first rank is the one closest to $(0, 0, 0)$, the second is closest to $(0, y_{\max}, 0)$, the third to $(x_{\max}, y_{\max}, 0)$, the fourth to $(x_{\max}, 0, 0)$, and so on. (We use $x_{\max}$ and $y_{\max}$ for the maximum coordinates in these dimensions.) This ordering in 2D is illustrated by Figure 5.

The corner-based algorithms are adaptations of heuristics "Expand from corner" and "Corners to center" described by Bhatelé et al. [13] for task mapping when the allocated ranks form a contiguous rectangle. These previous versions were presented in 2D and then "stacked" to form 3D algorithms for contigous allocations. Our corner-based algorithms differ by being a "native 3D" version of the idea and incorporating rotations as described above.

Overlay-based algorithms An entirely new set of algorithms constructs the mapping by "overlaying" the job on the mesh to find a desired location for each task. The desired location of the task at $(0, 0, 0)$ is the coordinatewise minimums of each coordinate among ranks assigned to the job. We call this location the *basepoint*. Denote its coordinates with (b_x, b_y, b_z). The desired location of each other task is placed relative to this. Thus, a task with coordinates (i, j, k) within the job has desired location $(b_x + i, b_y + j, b_z + k)$. The algorithm OVERLAY considers tasks in column major order and assigns each task to the unassigned rank closest to that task's desired location. Figure 6 illustrates the OVERLAY algorithm in 2D.

The algorithm TWOWAYOVERLAY extends this idea to work from both directions, with the sequence of tasks to assign alternately selected in column major order from the front bottom left and the reverse order from the back top right. When assigning a task reached in the forward direction, it behaves identically to OVERLAY. When assigning a task reached in the reverse direction, it uses the overlay computed from a back top right basepoint whose loca-

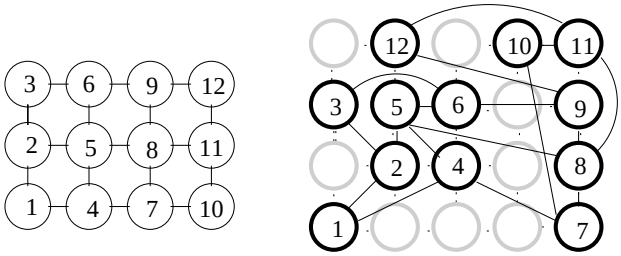

Figure 6. Mapping by OVERLAY.

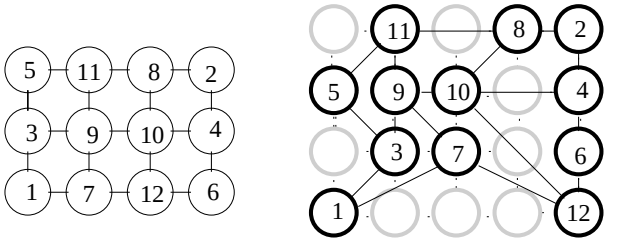

Figure 7. Mapping by TWOWAYOVERLAY.

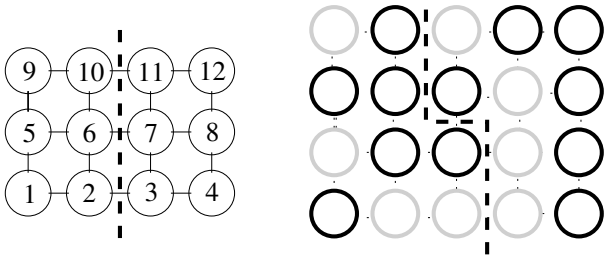

Figure 8. First cut made by RCB.

tion is the coordinatewise maxima of ranks assigned to the job. This algorithm in 2D is illustrated by Figure 7.

Recursive Coordinate Bisection Our final algorithm, recursive coordinate bisection (RCB), works by recursively dividing both the job and the set of allocated ranks. Specifically, it splits the job into two parts along its largest dimension as evenly as possible. Then it divides the allocated ranks into two parts of the same size along the same dimension (with tie-breaking). Figure 8 shows a division based on x coordinate. Next, the algorithm recursively maps each half of the job onto the corresponding half of the ranks. The recursion stops when a part contains just a single task, at which point that task is mapped to the single rank. The completed mapping in 2D is illustrated in Figure 9.

Recursive bisection has long been used for problem partitioning (e.g. [11]). Our approach turns this into a task mapping algorithm by also partitioning the set of allocated ranks in order to identify local subsets not only in the task graph, but also in the allocation graph. Hoefler and Snir [22] also use a recursive bisection heuristic for mapping, but they seek to map an arbitrary graph to an arbitrary graph. This is a more general problem, but the general formulation misses some of our problem's geometry since the "direction" of the cuts can be inconsistent between levels of recursion. The addition of rotations is also a departure from these other applications of recursive bisection.

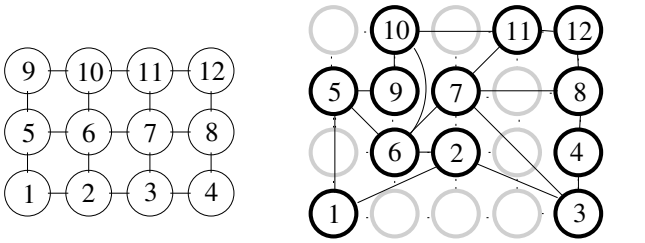

Figure 9. Mapping by RCB.

Total Cores	Cores per Rank (X, Y, Z)				
	16	8	4	2	1
32	1, 2, 1	1, 2, 2	2, 2, 2	2, 4, 2	2, 4, 4
64	1, 4, 1	1, 4, 2	2, 4, 2	2, 8, 2	2, 8, 4
128	2, 4, 1	2, 4, 2	4, 4, 2	4, 8, 2	4, 8, 4
256	2, 4, 2	2, 4, 4	4, 4, 4	4, 8, 4	4, 8, 8
512	2, 8, 2	2, 8, 4	4, 8, 4	4,16, 4	4,16, 8
1K	4, 8, 2	4, 8, 4	8, 8, 4	8,16, 4	8,16, 8
2K	4, 8, 4	4, 8, 8	8, 8, 8	8,16, 8	8,16,16
4K	4,16, 4	4,16, 8	8,16, 8	8,32, 8	8,32,16
8K	8,16, 4	8,16, 8	16,16, 8	16,32, 8	16,32,16
16K	8,16, 8	8,16,16	16,16,16	16,32,16	16,32,32
32K	8,32, 8	8,32,16	16,32,16	16,64,16	16,64,32
64K	16,32, 8	16,32,16	32,32,16	32,64,16	32,64,32

Figure 10. Job Dimensions

4. Experiments

4.1 Experiment setup

The experiments were run on the ACES [6] system Cielo [25], located at Los Alamos National Laboratories. Cielo is a Cray XE6 with 143,104 compute cores in 8,944 compute nodes. Each compute node is a dual AMD Opteron 6136 eight-core "Magny-Cours" socket G34 running at 2.4 GHz. Each service node is a 272 AMD Opteron 2427 six-core "Istanbul" socket F running at 2.2 GHz. The high speed interconnect is a Cray Gemini 3D torus in a sixteen by twelve by twenty-four (XYZ) topology. There are two nodes (sockets) per Gemini. The bi-section bandwidth is 6.57 by 4.38 by 4.38 (XYZ) TB/s. As of June 2013, Cielo was number 22 on the Top500 list [3].

The applicaton used in the experiments was miniGhost. As part of the exascale research program, the DOE lab community is developing mini applications (miniApps) that are representative of the computational core of major advanced simulation and computing codes. MiniGhost is a miniApp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. The miniGhost application [8] is a bulk-synchronous message passing code whose structure is modeled on the computational core of CTH [21]. CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories.

A set of experiments consists of miniGhost runs for various numbers of total cores and cores per MPI rank as shown in Figure 10, which gives the job dimensions. All jobs in a set of experiments were submitted at roughly the same time. Due to system load, the first set of experiments took almost two weeks to get on and off of Cielo. Others were faster; the second set ran in less than a day.

For a given number of cores, a single script (allocation) was used. Ten task mapping algorithms were then run for each core per rank on that allocation. The entire set of ten algorithms were run one job after another. This was done to minimize the experimental

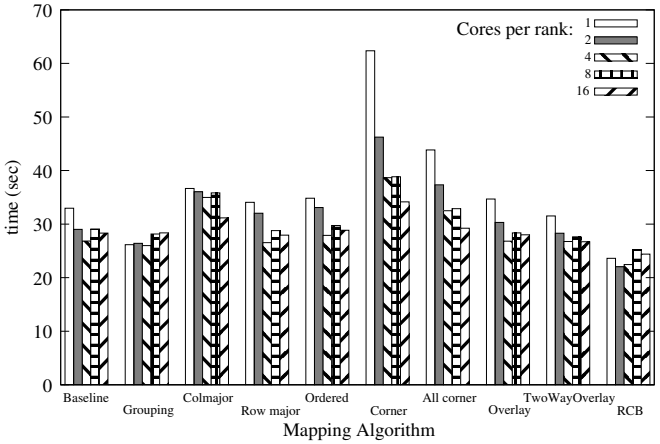

Figure 11. Running time for 64K-core job as a function of the number of cores per MPI rank.

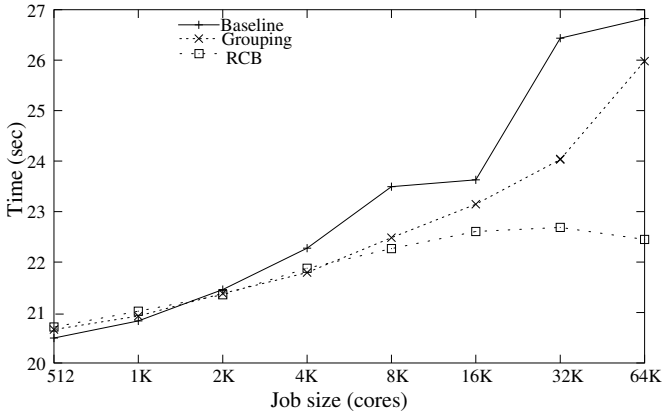

Figure 13. Running time as a function of job size for the best mapping algorithms

variances other than the cores per rank and task mapping algorithms for a given number of cores in a single set of experiments.

The task mapping algorithm is selected early in the miniGhost application. All the task mapping algorithms are implemented in a similar manner. The results show that the task mapping algorithms themselves (as opposed to the application being mapped) are fast and differences in their running times are insignificant. The miniGhost output includes total time, communication time as a percentage of total time, and average hops between neighboring ranks in the application. The application spends about thirty percent of its time communicating.

4.2 Number of cores per MPI rank

The miniGhost application has as a tuning option the number of cores assigned to each MPI rank. This can range from 1 (each core gets its own MPI rank) to 16 (one MPI rank per socket). Note that we place multiple MPI ranks per socket as we increase the number of ranks; these experiments are about changing the balance of MPI and OpenMP used by the job rather than its size. Figure 11 shows the running time for our largest size job (64K cores) for different numbers of core per rank. Each entry is the average of five runs except for the 1 core per rank entries of CORNER, ALLCORNERS, OVERLAY, and TWOWAYOVERLAY; one of our runs for each of these timed out so those entries is the average of only 4 runs.

Figure 11 shows two different performance trends. For some of the mappers, the best performance is at 16 cores per MPI rank, the maximum value considered. For others, it is at an intermediate value. Because the latter behavior is consistent with results obtained by others [9] and the algorithms with the best overall performance fall into the second camp, we focus on the results for 4 cores per MPI rank (best for most of these algorithms).

We note that using 16 cores per rank minimizes the number of ranks and means that each node has a single rank. That the more poorly-performing algorithms tend to favor this setting and deteriorate monotonically as the number of cores per rank decreases suggests that they are network bound, while the higher quality mappings provided by better-performing algorithms allow other factors to come into play.

4.3 Comparison between mappers

Figure 12 shows the running time as a function of job size for all ten algorithms. RCB performs consistently well and is the best algorithm for most job sizes.

Of the linear algorithms, COLMAJOR is consistently worse than ROWMAJOR and it turns in some poor performances at large sizes. We attribute this to the job dimensions, in which the y dimension is often larger than the others (see Figure 10). This is the dimension that COLMAJOR's order moves down first and going down the long dimension is bad for the linear algorithms since long communication distances occur when the ranks in a line along that dimension end before the tasks in the corresponding line, causing the last rank in one line to communicate with the first rank in the next.

Surprisingly, ORDERED is sometimes worse than COLMAJOR and ROWMAJOR even though it chooses between these (and other) possible orderings. We attribute this to the imperfect knowledge with which ORDERED makes its decision; even though we show that average hops and running time are correlated, ORDERED can misjudge the relative quality of its choices.

Figure 13 focuses on the algorithms from prior work (BASELINE and GROUPING) plus RCB, which performs best on the largest jobs. The results are mixed for relatively small jobs, but it is clear that RCB performs much better than the others at large scale. Its outperformance of BASELINE increases with the job size, reaching just over 16% at 64K cores. The outperformance is even better with different numbers of cores per MPI rank. For 64K cores, with 2 cores per MPI rank RCB achieved a 24.1% improvement over BASELINE and with 1 core per MPI rank it achieved a 28.4% improvement.

These gains are fairly consistent across runs. For jobs with 64K cores, RCB with 4, 2, and 1 cores per rank gave improvements over BASELINE of 12.8–18.6%, 12.5–29.6%, and 22.1–35.5% respectively. The 12% improvements were both on the 5th run in which BASELINE performed relatively well; without this run the ranges become 15–18.6%, 20.2–29.5%, and 25.8–35.4% respectively. Note that most of the variation within these ranges comes from BASELINE; the standard deviations for running time with RCB were 0.6 seconds, 0.4 seconds, and 0.2 seconds while with BASELINE they were 0.6 seconds, 2.2 seconds, and 2.3 seconds. Thus, RCB gives both better and more predictable performance.

Furthermore, the overall best running time for jobs with 8K and more was nearly always RCB with 2 cores per rank. (This value does not give the best improvement over BASELINE because BASELINE performs worse with 1 core per rank than 2.) Of the twenty experiments with these settings (5 repetitions of 4 sizes), RCB with 2 cores per rank was only beaten twice by RCB with 1 core per rank (both on jobs with 32K cores) and once by GROUPING with two cores per rank (16K cores). Even in these exceptions, RCB with 2 cores resulted in a running time within 1.2% of the best.

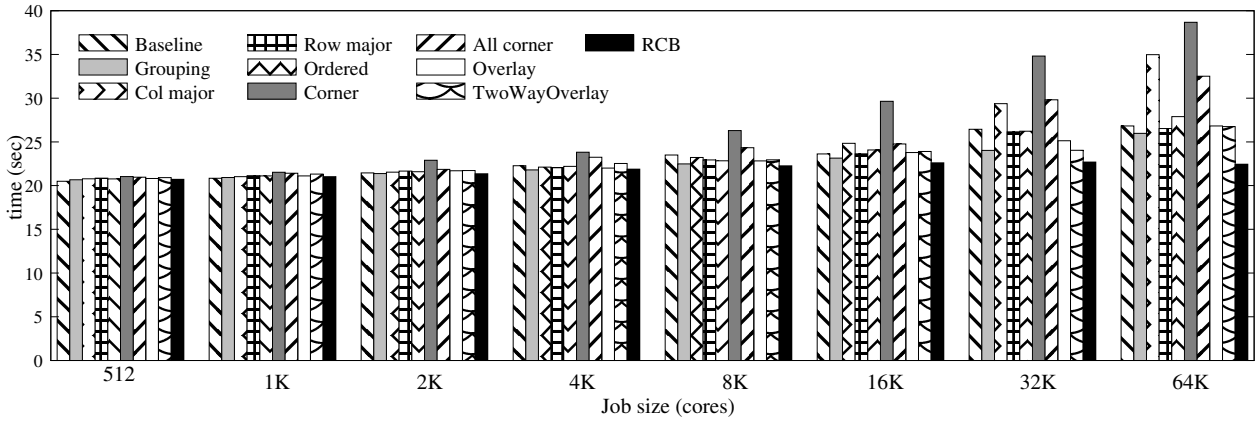

Figure 12. Running time as a function of job size

Beyond the specific numbers, the lesson of Figure 13 is that task mapping becomes increasingly important with job size; the spread between the largest and smallest values starts at 2% at 512 cores and grows to over 16% at 64K cores. Since the number of nodes and cores are expected to continue growing, the apparently-scalable performance of RCB (or successor algorithms) could be crucial to keeping communication costs reasonable.

4.4 Benefit of rotations

We were also interested in measuring the impact of rotations on task mapping performance. To do this, we ran a set of experiments with some of the algorithms above and versions of those same algorithms where the rotation code had been removed. For each job size, we looked at all the numbers of cores per rank.

On average over all jobs of size 512 cores and larger, RCB did 0.77% better with rotations, performing better in 24 of 40 trials. (If rotations were irrelevant, we would expect a 0% average improvement and performing better on exactly half of the trials; changes in machine state between the runs mean that actual ties are unlikely.) When restricted to jobs with 8K cores or more, it did 1.03% better, performing better on 13 or 20 trials.

For the other algorithms we considered (ROWMAJOR, CORNER, and ALLCORNERS), having rotations was always better on average for jobs with 8K+ cores and better on average for jobs with 512+ cores except for the ROWMAJOR mapper (where it was worse by 0.05% on average). For all of these mappers, the average improvement was always bigger for 8K+ core jobs than 512+ core jobs. The benefit of rotations was greatest for the CORNER mapper, where they improved average performance by 3.33% for jobs with 512+ cores and 5.9% for jobs with 8K+ cores.

Thus, despite the limitation of having just a single set of runs, we conclude that rotations provide a small but consistent gain on average and that, like the effect of mapping overall, this effect seems to increase with job size.

4.5 Correlation with hop metrics

To more deeply understand our results, we investigated the relationship between experimentally-observed running time and several abstract metrics. The first of these is the average L_1 distance between ranks with communicating tasks. Assuming x-y-z routing, this is equal to the *average number of hops* (average hops) required for a message to traverse a communication path. The number of hops that a message travels has a direct impact on its latency, but also serves as a proxy for contention since the message consumes bandwidth on each traversed link. If the job performs the same amount of communication between each adjacent pair of tasks, this metric is equivalent to the hops-bytes metric considered in prior work (e.g. [13]).

Our second metric is the *variance* of the number of hops, the average of the squared deviation from the average number of hops. This provides a measure of the "jitter" in communication times.

Our final metric is the *maximum hops*, the longest distance of any communication path. This is justified by the observation that long communication distances can have disproportionate impact on job running time because other tasks may wait for delayed messages.

Of these metrics, average hops turns out to be the best correlated with job running time. For each combination of job size and number of threads per MPI rank, we calculated the Spearman's rank correlation. (Recall that we used job sizes that were powers of 2 from 32 to 64K and that the number of threads per MPI rank was a power of 2 from 1 to 16.) The rank correlation coefficients are generally increasing with job size and decreasing with the number of threads per MPI rank. Based on the achieved values, we can reject the null hypothesis with significance level less than 0.05 for all configurations of jobs having at least 1024 cores (all numbers of threads per rank). The significance level was less than 0.01 for all configurations having at least 8K cores and all configurations having at least 2K cores except those with 16 threads per rank. We used the table in [31] for the confidence values. In addition, if we combine the data points for different numbers of threads per rank, we find that all jobs sizes except 64 cores achieve the 0.005 significance level (determined by multiplying by $\sqrt{n-1}$ to convert the distribution of rank correlation coefficients to normal).

We used the same procedure to examine the other metrics, but the correlation was less strong. For variance, the rank correlation coefficient still generally increased with job size, but there was no clear pattern involving the number of threads per MPI rank. Among configurations having at least 1K cores, all but three acheived the 0.05 significance level. (The exceptions were 1K cores with 8 threads/rank, 4K cores with 16 threads/rank, and 16K cores with 1 thread/rank.) When combining the data points for different numbers of threads per rank, it achieved the 0.005 significance level for jobs of size 2K and larger.

For max hops, the rank correlation coefficient again generally increased with job size, but 4 threads per rank most often achieves the 0.05 significance level. It does so for jobs having 8K or more cores. This level is achieved rarely for jobs smaller than 64K having 16 threads/rank or 1 threads/rank and for jobs smaller than 32K having 8 threads/rank or 2 threads/rank.

In addition to these piecewise results, all three metrics achieved the 0.005 significance level when all the runs were combined. Com-

paring the values of the rank correlation coefficients individual configurations and the values for each size or number of threads/rank, suggests that average is most highly correlated with running time, then variance, and finally max hops. The values when all runs are combined actually slightly favor max over variance.

5. Trace-based simulation

To further explore the performance of our algorithms and to see them in more varied scenarios, we examined them with a high-level trace-based simulator. This simulator was used to schedule and allocate jobs from the traces using algorithms similar to those used in practice. The resulting allocations were then used as input to our task mapping algorithms. The traces used to run our simulator did not provide any information about the messages sent by each job so we assumed a stencil pattern and evaluated the quality of the mappings by their average hops. Thus, our simulations represent a tradeoff in terms of fidelity: we used actual traces from HPC systems and performed scheduling and allocation with algorithms used in practice, but did not model the interaction between jobs and evaluated mappings using an approximation of their true quality. By making these approximations, we were able to evaluate the mapping algorithms on more than a million combinations of job and allocator. (For each job in a trace, the allocator's previous decisions determine which nodes are free and this, plus its decision on the job itself, determine the ranks that are passed to the mapping algorithm for that job.)

5.1 Simulation setup

To drive the simulator, we draw on the Parallel Workloads Archive [19], which contains job logs from a variety of HPC systems. From these logs, we are able to get each job's arrival time, size, running time, and (in many cases) the running time estimate submitted by the user. Unfortunately, the logs do not provide any information on job communication patterns and very few give any guidance about the job's desired shape. Since this information was unavailable, we supposed that every job used a stencil pattern on a mesh whose dimensions we assigned. Although not all jobs would use a stencil pattern in practice, the scheduler and allocators are independent of the communication pattern and that our evaluation function (average hops) works on only a single job. Thus, assuming all jobs have this pattern allows us to look at how the mappers would perform on each job if it used a stencil pattern without our assumptions causing any interference between jobs.

To assign dimensions with as little dependence on relative shapes as possible, we made everything as square as possible. Specifically, we used traces for machines whose number of nodes allowed those nodes to be arranged in a perfect square. To assign job dimensions to a job that wanted p ranks, we assigned it dimensions x by y (by 1) where x was the largest integer $\leq \sqrt{p}$ such that x divides p and $y = p/x$. Since some algorithms could perform poorly on long, skinny jobs, we skipped jobs whose dimensions x or y exceeded the machine dimensions. (In practice, these jobs would need to be mapped of course. Some ideas would be to "fold" or "squish" them in a preprocessing step so that they would fit.) We also skipped serial jobs, which are uninteresting from a mapping perspective. Figure 14 lists the traces used, the shape assigned to each machine, and the number of jobs that were included in our simulations, along with the percent of total trace jobs they represent. The traces where we are using less than 90% of the jobs both have large numbers of serial jobs; 32.9% of the jobs in the KTH-SP2 trace and 15.2% of the jobs in the SDSC-Par95 trace are serial jobs. (Note that we simulate all the jobs since skipped jobs still affect when the others run and which ranks they are allocated due to the use of non-contiguous allocation.)

Log name	Machine	# jobs used
DAS2-fs0-2003-1.swf	12×12	204,777 (93.3%)
DAS2-fs1-2003-1.swf	8×8	37,392 (95.0%)
DAS2-fs2-2003-1.swf	8×8	61,214 (93.6%)
DAS2-fs3-2003-1.swf	8×8	64,876 (98.1%)
DAS2-fs4-2003-1.swf	8×8	32,506 (98.6%)
KTH-SP2-1996-2.swf	10×10	18,603 (65.3%)
LLNL-T3D-1996-1.swf	16×16	21,323 (100%)
SDSC-Par-1995-2.1-cln.swf	20×20	45,238 (83.8%)
SDSC-Par-1996-2.1-cln.swf	20×20	29,172 (90.8%)
LLNL-Atlas-2006-2.1-cln.swf	96×96	37,378 (98.0%)

Figure 14. Summary of traces used in simulations

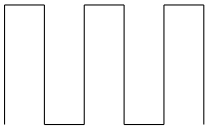

Figure 15. Snake curve

To schedule these traces, we used EASY [28], an algorithm that maintains a FIFO queue but allows a job not at the front to start anyway (called backfilling) if it is not expected to interfere with the job at the front of the queue when it does so. EASY is used in practice and it is often used as a baseline in scheduling research.

To allocate jobs once EASY decides to run them, we used two different allocators. The first is MC1x1 [10], which identifies ranks for an allocation by adding those at successively greater L_∞ (Manhattan) distances from a rank it selects as the center. The ranks at a given L_∞ distance are called a *shell*. If all ranks are free, the resulting allocation is a square with odd side length except that the outermost shell may not be completely populated depending on the number of tasks being allocated. Ranks that are busy with other jobs become holes in this square, potentially requiring additional shells to be examined.

The second allocation algorithm we considered is a linear scheme called snake best fit that combines ideas of Lo et al. [29] and Leung et al. [27]. This algorithm organizes the nodes in a linear order along a "snake" or "s-curve", which goes along the machine's short dimension and then curves back as shown in Figure 15. The free nodes are grouped into intervals according to their position along the curve and the algorithm allocates nodes from the smallest interval containing enough nodes (best fit). If no interval is large enough, then nodes are selected to minimize the *span*, the maximum distance along the curve between selected nodes. If all nodes are free, snake best fit will tend to create rectangular allocations that cross the entire machine, possibly with gaps in the boundary columns. If there is no interval entirely free, then busy nodes again create holes in the allocation. The snake best fit algorithm is much faster than MC1x1 and has been shown to generate allocations of comparable quality to MC1x1 [30] when "quality" is measured in terms of the average pairwise distance between nodes allocated to a job. This is equivalent to our average hops metric if the job's communication pattern is all-to-all, the worst case and perhaps the safest assumption if nothing is known about the job's actual communication pattern.

This linear scheme is similar to allocation algorithms provided as options in common cluster management software. SLURM [26] provides one that organizes the nodes using an approximation to a Hilbert curve (also considered by Leung et al. [27]). ALPS [23] orders the nodes based on a curve selected from a number of

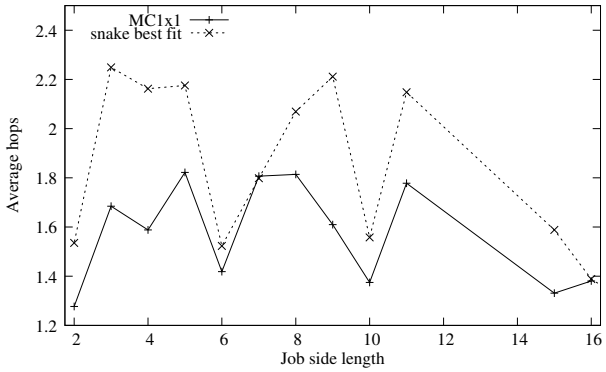

Figure 18. Average hops as a function of job side length for the RCB mapper over "squareable" jobs in all traces.

options at startup. ALPS does not use best fit packing, which was shown to be of lesser importance than curve selection [27].

5.2 Incremental improvement mapper

To provide context for the average hops metric, we also ran a simple incremental improvement or local search task mapping algorithm which we call INCIMPROVE. This algorithm starts with RCB and then swaps the mappings of pairs of tasks as long as doing so improves the average hops. We do not intend INCIMPROVE to be used in practice, but present it as an estimate of the best possible mapping. Note that INCIMPROVE is not guaranteed to find the absolute best possible since it can get caught in a local minima, but it serves as a useful proxy since there are too many possibilities to use brute force search to find the best mapping even for small jobs.

5.3 Results on traces

Figures 16 and 17 show the average hops for each trace using the MC1x1 and snake best fit allocators, respectively. Note that we only ran the LLNL-Atlas trace with the snake allocator; MC1x1 ran too slowly to include in our simulations. (Comparing centers takes cubic time in our implementation so the first 10 jobs of this trace averaged around 40 minutes each.)

As in the experiments, RCB looks to be the best algorithm of the ones discussed in Section 3. It was 2nd best to INCIMPROVE for all traces when using the MC1x1 allocator and all but one when using the snake allocator. (CORNER beat both RCB and INCIMPROVE by a small amount on the DAS2-fs4 trace with the snake allocator.) In fact, RCB's average hops are consistently quite close to INCIMPROVE, suggesting that it gives nearly optimal solutions according to the average hops metric.

We were interested in whether the allocators would favor different sized jobs. To examine this, we compared the average mapping quality between the two allocators on just the jobs that can form squares. Figure 18 shows the average hops for RCB over all the jobs in all the traces except LLNL-Atlas (for which we only ran the snake allocator). The results are broken out by side length. Note that there are no jobs of side length 12, 13, 14, 17, 18, or 19. We also excluded side length 20 since that occupies all of the largest machine, meaning they get the entire machine and a perfect mapping.

The results for the two mappers are quite similar, giving curves with similar shapes even though the specific values differ. For both mappers, the snake best fit allocator is nearly always worse by the average hops measure. We believe this is explained by the tendency of MC1x1 to give more "rounded" allocations while snake best fit favors skinnier allocations, with small jobs receiving a group of ranks all in a line. Since all the jobs considered in these results are square, mappers working with MC1x1 generally have to "stretch" the mesh communication pattern less to make it fit onto the allocated ranks, giving better average hop counts. On larger jobs, however, the disadvantage is somewhat lessened since the curve comes back and the job begins to widen. In addition, rectangles that run the entire length of the machine will pack more easily than the squarish allocations that MC1x1 tends to produce.

6. Summary and future work

Our work shows that task mapping can improve job running times, with the effect becoming crucial to high performance as the job size grows. We also showed that RCB is an effective task mapping algorithm for jobs using a stencil communication pattern. Future research will also benefit from our result that average hops is highly correlated with job running time— this facilitates simulations to identify promising algorithms and also provides a tractable metric for theoretical analysis.

We plan a number of steps going forward. Currently, RCB is implemented as a rank remapping performed within miniGhost. We will transfer it into a library so that other programs can easily adopt it. We also plan on investigating other communication patterns, with extensions of RCB being a natural place to start. In addition, we are interested in further investigating INCIMPROVE. Currently, it can run for a potentially-unbounded time, which is clearly unacceptable, but it might be possible to capture some of its benefits while adding limits (e.g. no more than x swaps, only make swaps that improve by $x\%$, etc).

References

[1] LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov/`.

[2] 2006.

[3] Top 500 list - November 2012. `http://www.top500.org/list/2012/11/`.

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

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

[6] J. Ang, D. Doerfler, S. Dosanjh, S. Hemmert, K. Koch, J. Morrison, and M. Vigil. The Alliance for Computing at the Extreme Scale. In *Proc. 52nd Cray User Group*, 2010.

[7] Y. Aridor, T. Domany, O. Goldshmidt, J. Moreira, and E. Shmueli. Resource allocation and utilization in the Blue Gene/L supercomputer. *IBM J. Research and Development*, 49(2/3):425, 2005.

[8] R. Barrett, C. Vaughan, and M. Heroux. MiniGhost: A miniapp for exploring boundary exchange strategies using stencil computations scientific parallel computing. Technical Report SAND2011-5294832, Sandia National Laboratories, 2011.

[9] R. Barrett, S. Hammond, C. Vaughan, D. Doerfler, J. Luitjens, and D. Roweth. Navigating an evolutionary fast path to exascale. In *Proc. 3rd Intern. Workshop Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS)*, 2012.

[10] M. Bender, D. Bunde, E. Demaine, S. Fekete, V. Leung, H. Meijer, and C. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 50 (2):279–298, 2008.

[11] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36(5):570–580, 1987.

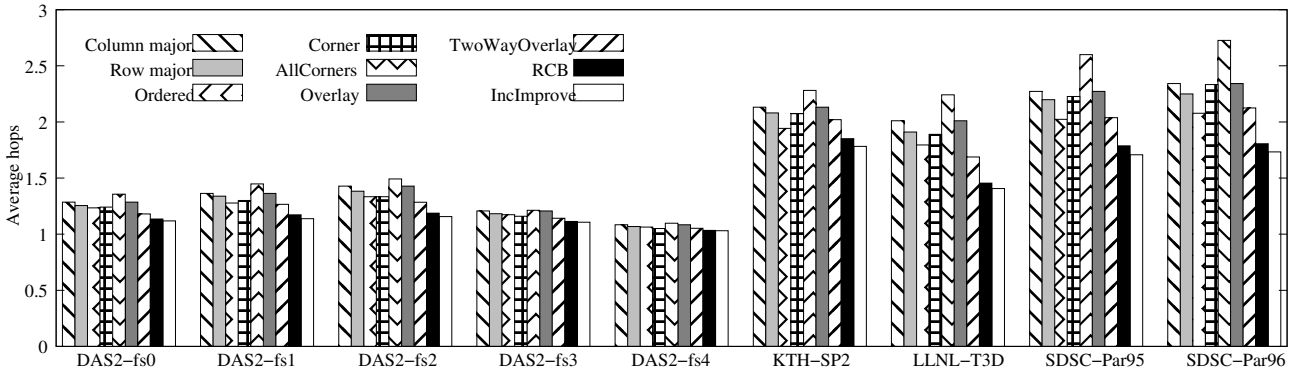

Figure 16. Average hops for each trace using the MC1x1 allocator.

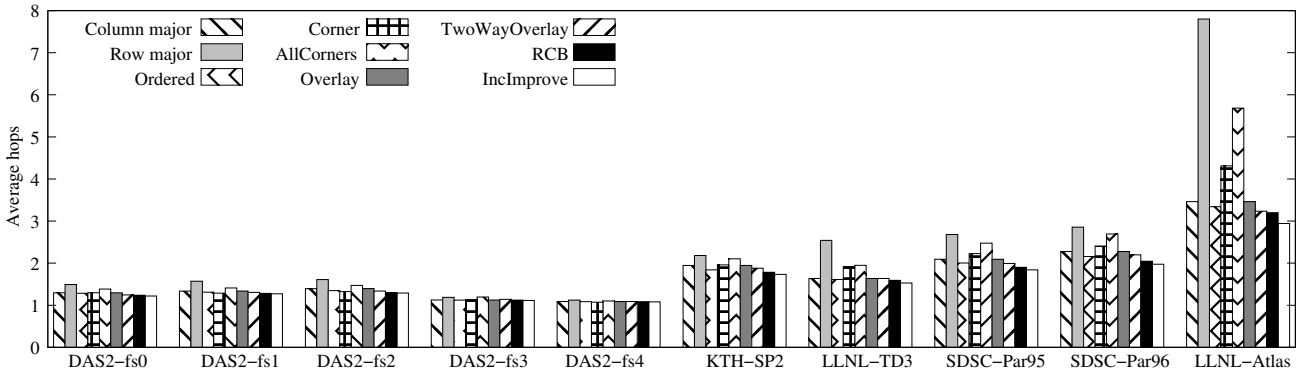

Figure 17. Average hops for each trace using the snake best fit allocator.

[12] A. Bhatele and L. Kale. Benefits of topology-aware mapping for mesh topologies. *Parallel Processing Letters*, 18(4):549–566, 2008.

[13] A. Bhatelé, G. Gupta, L. Kalé, and I.-H. Chung. Automated mapping of regular communication graphs on mesh interconnects. In *Proc. Intern. Conf. High Performance Computing (HiPC)*, 2010.

[14] S. Bokhari. On the mapping problem. *IEEE Trans Computers*, C-30 (3), 1981.

[15] S. Bollinger and S. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans. Computers*, 40(3), 1991.

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

[17] T. Chockalingam and S. Arunkumar. Genetic algorithm based heuristics for the mapping problem. *Computers and Operations Research*, 22(1):55–64, 1995.

[18] I.-H. Chung, C.-R. Lee, J. Zhou, and Y.-C. Chung. Hierarchical mapping for HPC applications. In *Proc. Workshop on Large-Scale Parallel Processing*, pages 1810–1818, 2011.

[19] D. Feitelson. The parallel workloads archive. `http://www.cs.huji.ac.il/labs/parallel/workload/index.html`.

[20] F. Gygi, E. W. Draeger, M. Schulz, B. de Supinski, J. Gunnels, V. Austel, J. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform. In *Proc. 2006 ACM/IEEE Conf. on Supercomputing* sc0 [2].

[21] E. Hertel, R. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proc. 19th International Symposium on Shock Waves*, 1993.

[22] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc. 25rd ACM Intern. Conf. Supercomputing (ICS)*, 2011.

[23] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing. The application level placement scheduler. In *Proc. Cray User's Group (CUG)*, 2006.

[24] H. Kikuchi, B. Karki, and S. Saini. Topology-aware parallel molecular dynamics simulation algorithm. In *Proc. Intern. Conf. Parallel and Distributed Processing Techniques and Applications*, 2006.

[25] L. A. N. Laboratory. High-performance computing: Cielo supercomputer. `http://www.lanl.gov/orgs/hps/cielo/index.html`, .

[26] L. L. N. Laboratory. SLURM: A highly scalable resource manager. `https://computing.llnl.gov/linux/slurm/`, .

[27] V. Leung, E. Arkin, M. Bender, D. Bunde, J. Johnston, A. Lal, J. Mitchell, C. Phillips, and S. Seiden. Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. In *Proc. 4th IEEE Intern. Conf. on Cluster Computing*, pages 296–304, 2002.

[28] D. Lifka. The ANL/IBM SP scheduling system. In *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 295–303, 1995.

[29] V. Lo, K. Windisch, W. Liu, and B. Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 8(7):712–726, 1997.

[30] P. Walker, D. Bunde, and V. Leung. Faster high-quality processor allocation. In *Proc. 11th LCI Intern. Conf. High-Performance Clustered Computing*, 2010.

[31] R. Walpole and R. Myers. *Probability and statistics for engineers and scientists*. Macmillan Publishers, 4th edition, 1989.

[32] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for Blue Gene/L supercomputer. In *Proc. 2006 ACM/IEEE Conf. on Supercomputing* sc0 [2].