SANDIA REPORT

SAND2013-0962
Unlimited Release
Printed February 2013

Task Mapping for Non-contiguous Allocations

Vitus J. Leung, David P. Bunde, Johnathan Ebbers, Nicholas W. Price, Matthew Swank, Stefan P. Feer, Zachary D. Rhodes

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
amp; U.S. Department of Energy
amp; Office of Scientific and Technical Information
amp; P.O. Box 62
amp; Oak Ridge, TN 37831
amp;
amp; Telephone: amp; (865) 576-8401
amp; Facsimile: amp; (865) 576-5728
amp; E-Mail: amp; reports@adonis.osti.gov
amp; Online ordering: amp; http://www.osti.gov/bridge

Available to the public from
amp; U.S. Department of Commerce
amp; National Technical Information Service
amp; 5285 Port Royal Rd
amp; Springfield, VA 22161
amp;
amp; Telephone: amp; (800) 553-6847
amp; Facsimile: amp; (703) 605-6900
amp; E-Mail: amp; orders@ntis.fedworld.gov
amp; Online ordering: amp; http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-0962
Unlimited Release
Printed February 2013

Task Mapping for Non-contiguous Allocations

Vitus J. Leung
Data Analysis and Informatics Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1327

David P. Bunde, Johnathan Ebbers, Nickolas W. Price, and Matthew Swank
Department of Computer Science
Knox College
Galesburg, IL 61401

Stefan P. Feer
3M Health Information Systems, Inc.
Wallingford, CT 06492

Zachary D. Rhodes
Allstate Corporation
Northbrook, IL 60062

Abstract

This paper examines task mapping algorithms for non-contiguously allocated parallel jobs. 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, making them completely isolated from each other. A middle

ground between these two cases is the mapping problem for non-contiguous jobs having a specific communication pattern. We propose several task mapping algorithms for jobs with a stencil communication pattern and evaluate them using experiments and simulations. Our strategies improve the running time of a MiniApp by as much as 30% over a baseline strategy. Furthermore, this improvement increases markedly with the job size, demonstrating the importance of task mapping as systems grow toward exascale.

Acknowledgments

We thank Courtenay Vaughan and Kevin Hastings for helpful discussions. D.P. Bunde, J. Ebbers, S.P. Feer, N.W. Price, Z.D. Rhodes, and M. Swank were partially supported by contract 899808 from Sandia National Laboratories. Z.D. Rhodes also acknowledges support from a Post-baccalaureate Fellowship from Knox College. We also thank all those who contributed traces to the Parallel Workloads Archive.

Contents

List of Figures

List of Tables

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 processors. It has a long history (eg. [12]) 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. [4, 22, 18, 11, 16, 20, 6, 14]). These experiments include actual applications, one of which exhibited a speedup of 1.64 times when the task mapping was improved [18]. The issue now is contention for limited bandwidth. Poorly placed tasks lead to messages traveling more hops in the network, consuming part of the capacity of each link along the route. Of course, messages traveling longer distances have always consumed more bandwidth, but the importance of this fact is being fueled by two ongoing trends. First of all, processors continue improving faster than networks, increasingly making bandwidth the limiting factor in performance. Second of all, processor counts in state of the art HPC systems have continued to grow, which both increases the number of hops between processors 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 like that of Blue Gene assume structured communcation patterns that fold and stretch one grid into another and ignores non-contiguous allocations.

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 processors allocated to other jobs interfere with the mapping. Figure 1 shows a good 2D mapping of a 4×3 job onto 12 allocated processors of a 5×4 machine. The solid lines connecting allocated processors show which pairs of processors communicate; actually communication must use the mesh edges (shown as dotted lines).

One of our algorithms, which recursively divides both the task graph and the set of allocated processors, is shown to greatly outperform the other algorithms, including the current mapper on Cielo.

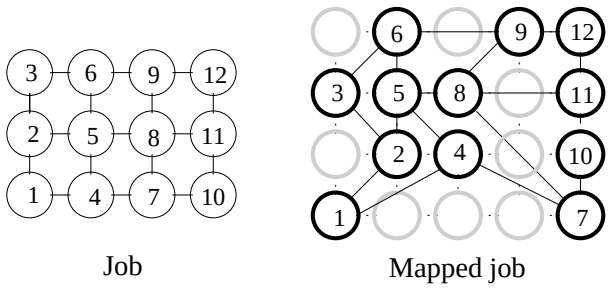

Figure 1. Possible mapping of a 4×3 job onto a 5×4 machine. Left: Job tasks with communication pattern shown. Right: Tasks mapped to dark processors.

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.

Organization of the paper

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

Motivation and related work

As mentioned above, previous work on task mapping falls into two main categories. The first category was introduced by Bokhari [12] in the first paper on task mapping. In this category, the job is represented as a graph whose nodes 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 processors 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 processors respectively. Since the problem is clearly NP-complete (formally shown in [20]), the papers generally use heuristic techniques such as genetic algorithms [15] and simulated annealing [13]. Since the graph formulation of task mapping is the most general, it remains a goal for recent researchers. Chung et al. [16] use a hierarchical approach to simplify the mapping problem to reasonable complexity. Hoefler and Snir [20] 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, but is also the norm on BlueGene systems, which guarantees each job its own submesh, which is kept isolated from other jobs [5].) For example, Yu et al. [31] 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. [10]; we adapt some of their heuristics and evaluate them in our setting.

Recently, some work has been done in the middle ground of jobs on a mesh, communicating in stencil patterns, but not isolated from other jobs. Barrett et al. [6] apply a custom reordering algorithm to their miniGhost and CTH jobs on a large Cray XE6 (Cielo). Originally their miniGhost jobs employing an x major mapping of processes to processors did not scale well above four thousand processes on Cielo. They reordered the mapping to "chunk" two by two by four submeshes of the job onto each sixteen-core Cielo node and mapped the chunks onto nodes in an x major fashion. This reordering reduced both the average hops between processes in the stencil and processing time for the jobs. Brown et al. [14] performed similar work on the more general processors command for LAMMPS [1].

Algorithms

Now we describe our task mapping algorithms. All of them are stated in terms of nodes; rather than assigning tasks to individual cores, follow the "chunking" approach described above so that each node gets a submesh. This preserves the job's mesh communication pattern while reducing the problem size and resolving the additional choices presented by having multicore nodes. It also automatically enforces a Sandia policy against multiple jobs running on a single node.

Linear algorithms The first set of algorithms we considered are *linear algorithms*, in which a linear ordering is used to assign tasks to processors. These algorithms are fast and easily implemented, making them attractive as a starting point.

The baseline task mapping algorithm provided by ALPS, Moab, and miniGhost is a linear algorithm. The cores are numbered by assigning the cores of each node consecutive numbers and visiting the nodes in the allocation order, which uses a space-filling curve similar to the

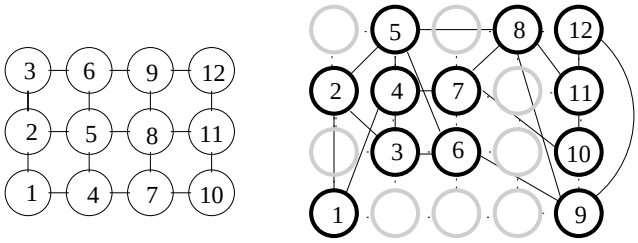

Figure 2. Mapping by ColMajor.

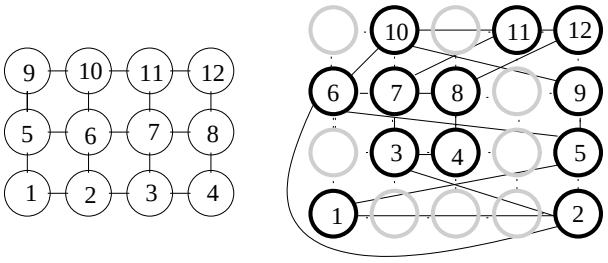

Figure 3. Mapping by RowMajor.

linear scheme described in Section ; see [3] for details. This algorithm 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 three orders, each task is assigned to its corresponding core. We call this the Baseline algorithm since it is the task mapper provided by ALPS, Moab, and miniGhost and it was in use prior to our work.

Barrett et al. [6] proposed a refinement of this which groups the tasks into $2 \times 2 \times 4$ blocks, each of which is assigned to a node (which has 16 cores since their experiments, like ours, were performed on Cielo). We use Grouping as a shorthand name for their algorithm.

In addition to these previously-implemented algorithms, we added ColMajor, in which both the tasks and the allocated processors are numbered in column major order (coordinates increase first in y, then x, and finally in z). Each task is assigned to its corresponding processor. 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.

A natural extension of ColMajor and RowMajor is to try different linear orderings. This is the basis for the algorithm Ordered, which considers these 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

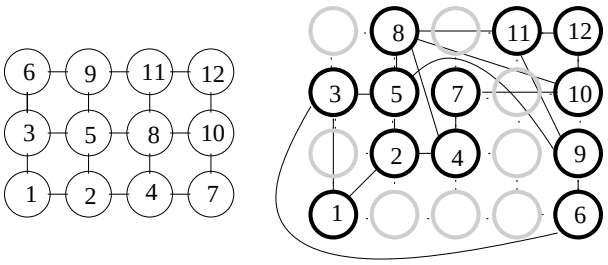

Figure 4. Mapping by CORNER.

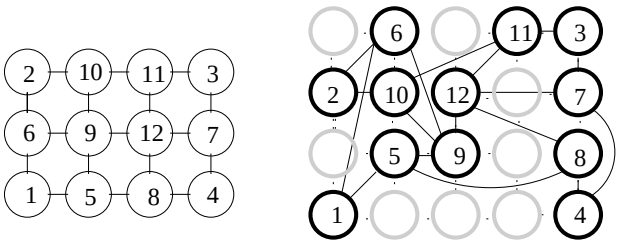

Figure 5. Mapping by ALLCORNERS.

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 processors, mapping each task to the corresponding processor. The ordering it uses is distance from front bottom left corner of the mesh, 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 processor is the one closest to the front bottom left corner, the second is closest to the front top left, the third to the front top right, the fourth to the front bottom right, and so on. 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. [10] for task mapping when all processors are available.

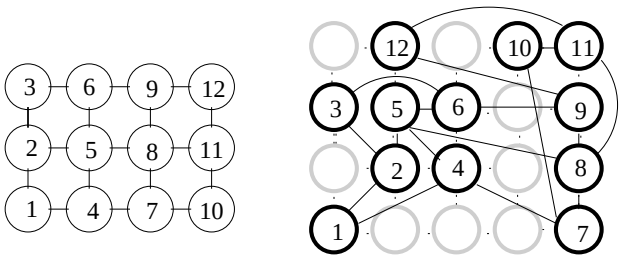

Figure 6. Mapping by Overlay.

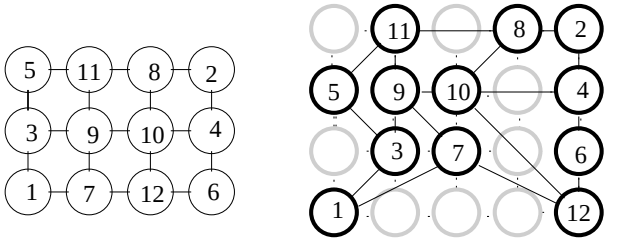

Figure 7. Mapping by TwoWayOverlay.

Overlay-based algorithms Another set of algorithms constructs the mapping by "overlaying" the job's desired shape on the mesh to find the best processor to assign each task. Each overlay is placed relative to a basepoint. The coordinates of the front lower left basepoint are the coordinatewise minimums of each coordinate among processors assigned to the job. The overlay places the front bottom left corner of the job in this location, with other tasks placed relative to it as if all processors were available. Thus, a task with coordinates (i, j, k) within the job is placed within the overlay i positions from the basepoint in the x direction, j positions from the basepoint in the y direction, and k positions from the basepoint in the z direction. The algorithm Overlay uses this overlay and, considering tasks in column major order, assigns each task to the unassigned processor closest to that task's location in the overlay. 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 location is the coordinatewise maxima of processors assigned to the job. This algorithm in 2D is illustrated by Figure 7.

Recursive Coordinate Bisection Our final algorithm, recursive coordinate bisection (RCB) [9], works by recursively dividing both the job and the set of allocated processors.

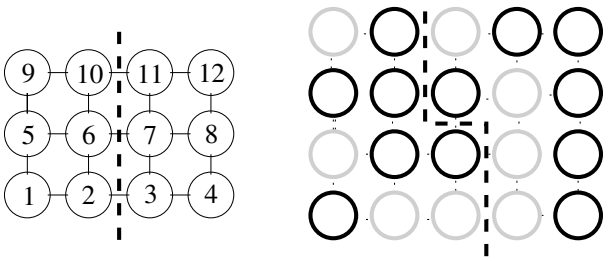

Figure 8. First cut made by RCB.

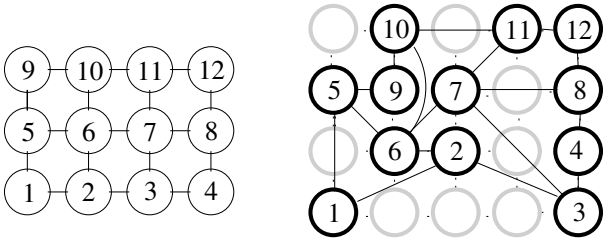

Figure 9. Mapping by RCB.

Specifically, it splits the job into two parts along its largest dimension as evenly as possible. Then it similarly divides the allocated processors into two parts of the same size based on the same dimension (with tie-breaking). Figure 8 shows a division based on x coordinate. Next, the algorithm recursively maps the tasks to the left onto the processors on the left and the tasks to the right onto the processors on the right. (Substitute above/below or in-front-of/behind for left/right when the division is made along other dimensions.) The recursion stops when a part contains just a single task, at which point that task is mapped to the single processor. The completed mapping in 2D is illustrated in Figure 9.

Rotations As a preprocessing step for most of the algorithms, we rotate the job if doing so makes its aspect ratio match more closely with the chosen set of allocated processors. Specifically, we compare the relative orders of dimension lengths for the job and the bounding box of the set of allocated processors. If these orders differ, we rotate the job to bring them into the same order. 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 operation is performed for all the algorithms except the baseline and Grouping. Grouping is excluded because it was specifically motivated by a desire to lessen the number of hops between jobs in the z dimension and rotations would interfere with this.

Experiments

Experiment setup

The experiments were run on the Los Alamos National Laboratory Cielo machine [24]. Cielo is a Cray XE6 with 143,104 compute cores in 8,944 compute nodes. Each node is a dual AMD Opteron 6136 eight-core "Magny-Cours" socket G34 running at 2.4 GHz. The service nodes are 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 November 2012, Cielo was number eighteen on the top 500 list [2].

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 [7] is a bulk-synchronous message passing code whose structure is modeled on the computational core of CTH [19]. 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 1, 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. A second set of experiments took 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 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 differences in running time for the task mapping algorithms 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.

Total	Cores per Rank (X, Y, Z)				
Cores	16	8	4	2	1
16	1, 1, 1	1, 2, 1	2, 2, 1	2, 4, 1	2, 4, 2
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

Table 1. Job Dimensions

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 MP and OpenMP used by the job rather than its size. Figure 10 shows the running time for our largest size job (64K cores) for different numbers of core per rank. Each entry is the average of two 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 contain a single data point.

Figure 10 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 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).

Comparison between mappers

Figure 11 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

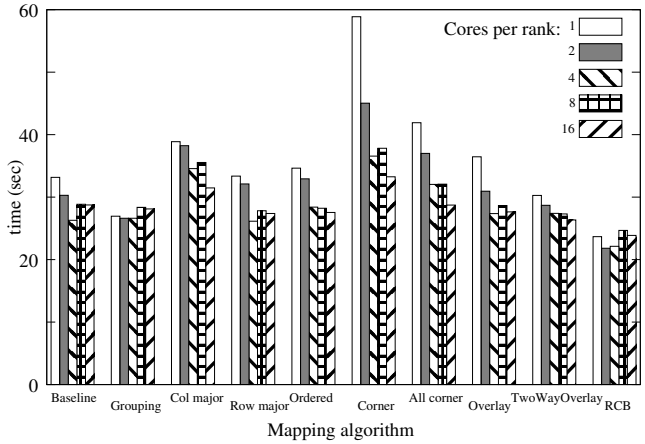

Figure 10. Running time for 64K-core job as a function of the number of cores per MPI rank.

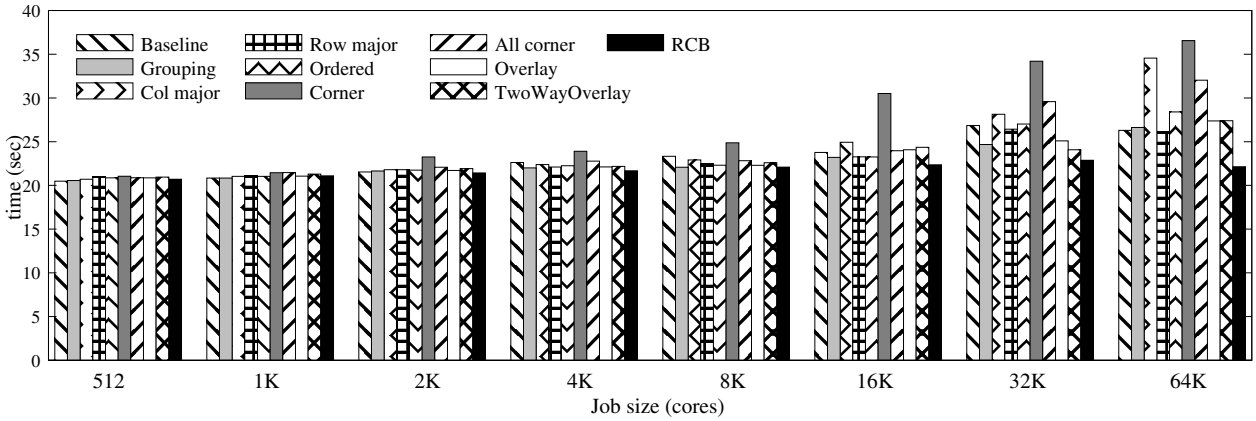

Figure 11. Running time as a function of job size

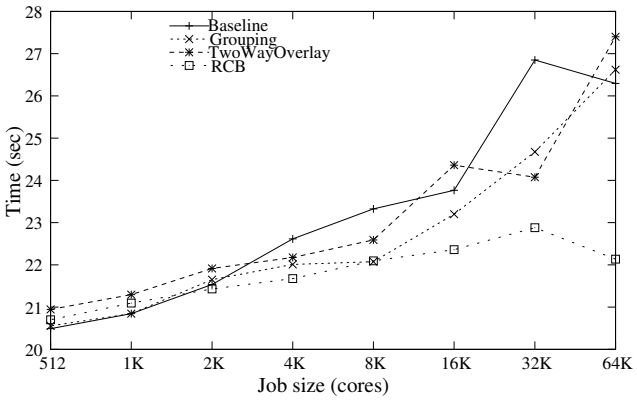

Figure 12. Running time as a function of job size for the best mapping algorithms

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 1). This is the dimension that ColMajor's order moves down first and going down the long dimension is bad for the linear algorithms since one or two previously-allocated nodes in a column create long communication distances because a column of tasks gets split between the top of one column of nodes and the bottom of 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.

Our results consistently favor the algorithms that work from multiple ends of the job, with AllCorners outperforming Corner and TwoWayOverlay outperforming Overlay. Indeed, this intuition might extend to explain why RCB does so well since it builds the mapping from both halves of the task at each step of the recursion.

Figure 12 focuses on the baseline and the 3 most promising other algorithms (Grouping, TwoWayOverlay, and RCB). There is some noise in the results, but it is clear that RCB performs much better at large scale than the others. Its outperformance of the baseline algorithm increases with the job size, reaching nearly 17% 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 27.9% improvement over the baseline (the individual runs achieved 27.6% and 28.4%) and with 1 core per MPI rank it achieved a 28.6% improvement (26.3% and 30.8% for individual runs).

Beyond the specific numbers, the lesson of Figure 12 is that task mapping becomes increasingly important with job size. Since the number of nodes and cores are expected to continue growing as we move to exascale, the apparently-scalable performance of RCB (or successor algorithms) could be crucial to keeping communication costs reasonable.

Correlation with hop metrics

Since computer time on large systems for these experiments is scarce, we are also interested in identifying metrics by which we can judge mapping quality in simulations. We investigated a couple of possibilities. The first of these is the average L_1 distance between processors 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 link that it traverses. If the job performs the same amount of communication between each adjacent pair of tasks, this metric is equivalent to the hops-bytes used by others (e.g. [10]).

In addition, we wanted to take long communication distances into account since delayed messages can have disproportionate impact on job running time if tasks must wait for them. The first of our added metrics is *variance*, 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, which is justified by the observation that tasks wait for the last message of a communication round before continuing.

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 16 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 nearly all configurations of jobs having at least 512 cores and 8, 4, 2, or 1 threads per rank and all configurations of jobs having at least 8K cores for larger numbers of threads per rank. (The single exception is 2K jobs having 2 threads per rank.) We used the table in [30] 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 this same 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. The best numbers were 8 threads per rank, where all but one job size (2K) at least 1K achieved the 0.05 significance level, and 2 threads per rank, where 2K, 4K, 32K, and 64K did so. Of the 40 combinations of threads per rank and job sizes 2K or larger, 16 of them achieved this level. When combining the data points for different numbers of threads per rank, it did so for jobs of size 256 and larger except for 8K.

For max hops, the rank correlation coefficient again generally increased with job size and decreased with number of threads per MPI rank, but only reached the 0.05 significance level

for jobs with at least 32K cores and only for some of the numbers of threads per rank. When combining the data points for different numbers of threads per rank, it did so for jobs of size 16K and larger.

In addition to these piecewise results, all three metrics achieved the 0.01 significance level when all the runs were combined. Comparing the values of the rank correlation coefficients suggests that average is most highly correlated with running time, then max hops, and finally variance.

Trace-based simulation

To further explore the performance of our algorithms and to see them in more varied scenarios, we examined them with a trace-based simulation. 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, allowing us to run the task mapping algorithms in situations very similar to those encountered on actual systems. A partial description of the simulator is presented in Rodrigues et al. [28, Section 4.7].

Simulation setup

To drive the simulator, we draw on the Parallel Workloads Archive [17], 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 decided to remove issues of machine and job shape from our experiments by focusing on the mapping of square jobs on square machines. Thus, we used only the logs for machines whose number of processors is a perfect square; these are listed in Figure 2 along with the shape each was assigned. From these logs, we present statistics on the mapping quality for only the jobs that can be arranged in a perfect square of size greater than 1 (serial jobs are uninteresting from a mapping perspective); the figure also shows the number of such jobs in each trace along with the percent of jobs they represent. (Note that we simulate all the jobs since non-square jobs still affect when the square jobs run and which processors they are allocated.)

To schedule these traces, we used EASY [26], 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

Log name	Machine	# jobs used
DAS2-fs0-2003-1.swf	12×12	30,265 (13.8%)
DAS2-fs1-2003-1.swf	8×8	3,984 (10.1%)
DAS2-fs2-2003-1.swf	8×8	8,451 (12.9%)
DAS2-fs3-2003-1.swf	8×8	9,368 (14.2%)
DAS2-fs4-2003-1.swf	8×8	2,281 (6.9%)
KTH-SP2-1996-2.swf	10×10	5,949 (20.9%)
LLNL-T3D-1996-1.swf	16×16	6,538 (30.7%)
SDSC-Par-1995-2.1-cln.swf	20×20	19,108 (35.4%)
SDSC-Par-1996-2.1-cln.swf	20×20	15,330 (47.7%)
LLNL-Atlas-2006-2.1-cln.swf	96×96	12,474 (32.7%)

Table 2. Summary of traces used in simulations

first is MC1x1 [8], which identifies processors for an allocation by adding those at successively greater L_∞ (Manhattan) distances from a processor it selects as the center. The processors at a given L_∞ distance are called a *shell*. If all processors 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 processors being allocated. Processors 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. [27] and Leung et al. [25]. This algorithm organizes the processors 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 13. The free processors are grouped into intervals according to their position along the curve and the algorithm allocates processors from the smallest interval containing enough processors (best fit). If no interval is large enough, then processors are selected to minimize the *span*, the maximum distance along the curve between selected processors. If all processors 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 processors 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 [29] when "quality" is measured in terms of the average pairwise distance between processors 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 [23] provides one that organizes the processors using an approximation to a Hilbert curve (also considered by Leung et al. [25]). ALPS [21] orders the processors based on a curve selected from a number of options at startup. ALPS does not

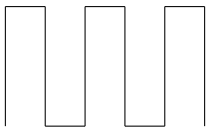

Figure 13. Snake curve

use best fit packing, which was shown to be of lesser importance than curve selection [25].

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.

Results on traces

Figures 14 and 15 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 (on a real system, the idle processors could potentially help make allocation decisions, making an expensive algorithm more practical).

As in the experiments, RCB looks to be the best algorithm of the ones discussed in Section . In fact, its average hops are consistently quite close to IncImprove, suggesting that it gives nearly optimal solutions according to this metric.

We were interested in what the trace-based results would say about the choice of allocator. To examine this, we looked at the average mapping quality for RCB over all the jobs in all the traces except LLNL-Atlas (for which only snake allocation is available; including it makes snake look worse since the larger machine means larger average distances). Figure 16 shows the results broken out by job size; recall that all jobs are square so we just report the 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

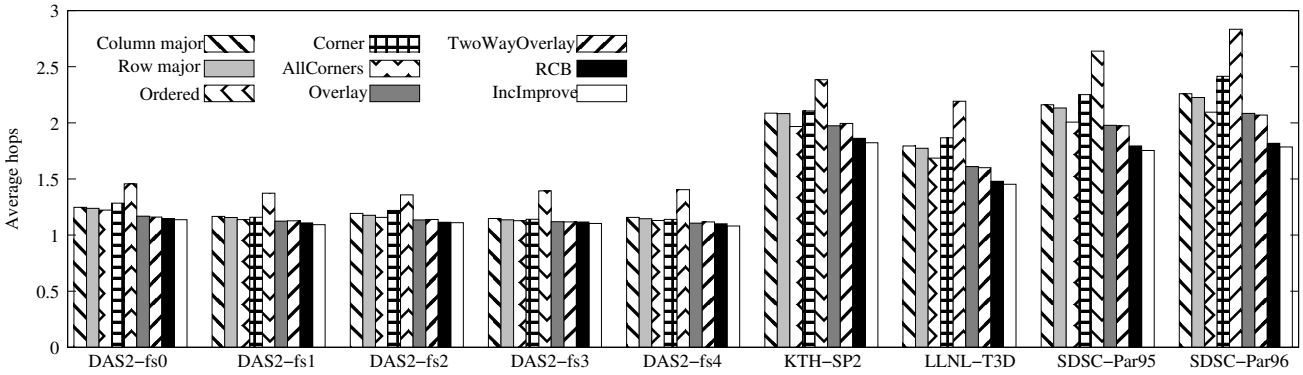

Figure 14. Average hops for each trace using the MC1x1 allocator.

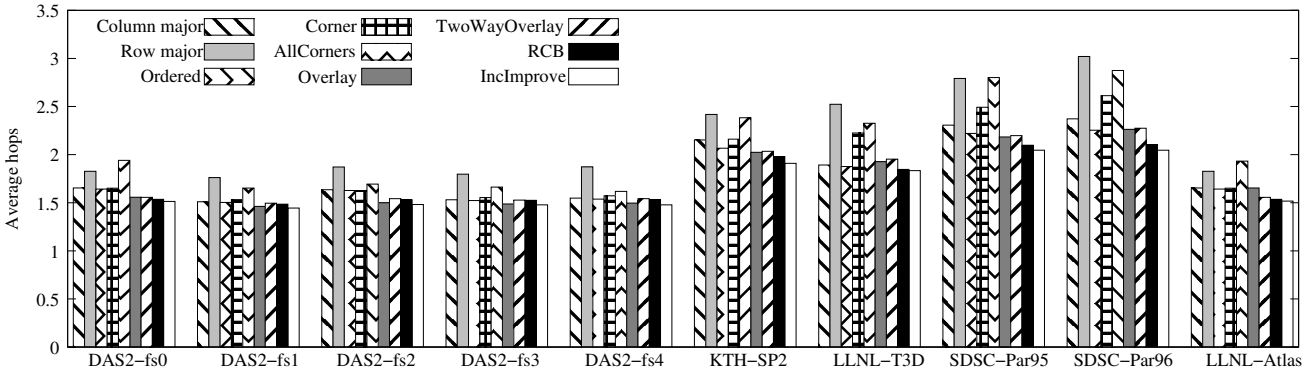

Figure 15. Average hops for each trace using the snake best fit allocator.

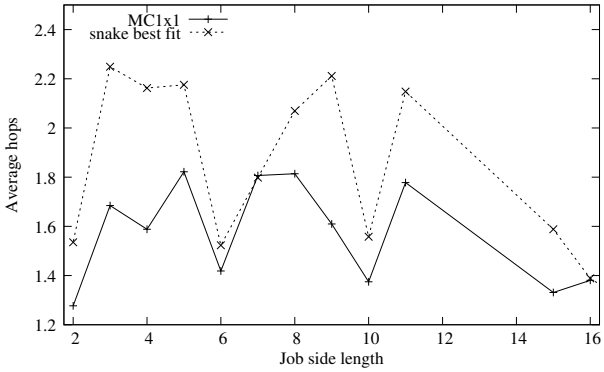

Figure 16. Average hops as a function of job size for the RCB mapper over all traces.

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 processors 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 processors, 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.

Discussion

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 since this facilitates simulations to identify promising algorithms.

We plan a number of steps going forward. Currently, RCB is implemented as a node 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] Top 500 list - november 2012. `http://www.top500.org/list/2012/11/`.

[3] 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.

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

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

[6] 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.

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

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

[9] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36, 1987.

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

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

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

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

[14] W. Michael Brown, Trung D. Nguyen, Miguel Fuentes-Cabrera, Jason D. Fowlkes, Philip D. Rack, Mark Berger, and Arthur S. Bland. An evaluation of molecular dynamics performance on the hybrid Cray XK6 supercomputer. In *Proc. Intern. Conf. Computational Science (ICCS)*, 2012.

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

[16] I-Hsin Chung, Che-Rung Lee, Jiazheng Zhou, and Yeh-Ching Chung. Hierarchical mapping for HPC applications. In *Proc. Workshop on Large-Scale Parallel Processing*, pages 1810–1818, 2011.

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

[18] F. Gygi, Erik W. Draeger, M. Schulz, B.R. de Supinski, J.A. Gunnels, V. Austel, J.C. Sexton, F. Franchetti, S. Kral, C.W. 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*, 2006.

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

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

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

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

[23] Lawrence Livermore National Laboratory. SLURM: A highly scalable resource manager. `https://computing.llnl.gov/linux/slurm/`.

[24] Los Alamos National Laboratory. High-performance computing: Cielo supercomputer. `http://www.lanl.gov/orgs/hps/cielo/index.html`.

[25] 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.

[26] 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.

[27] 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.

[28] A. Rodrigues, K. Bergman, D. Bunde, E. Cooper-Balis, K. Ferreira, K.S. Hemmert, B. Barrett, C. Versaggi, R. Hendry, B. Jacob, H. Kim, V. Leung, M. Levenhagen, M. Rasquinha, R. Riesen, P. Rosenfeld, M. del Carmen Ruiz Varela, , and S. Yalamanchili. Improvements to the structural simulation toolkit. In *Proc. 5th Intern. ICST Conf. Simulation Tools and Techniques (SIMUTools)*, 2012.

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

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

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

DISTRIBUTION:

1 David P. Bunde
Knox College Box K-100
2 E. South St.
Galesburg, IL 61401

1 Johnathan Ebbers
Knox College Box K-552
2 E. South St.
Galesburg, IL 61401

1 Stefan P. Feer
3M Health Information Systems, Inc.
100 Barnes Rd
Wallingford, CT 06492

1 Nickolas W. Price
Knox College Box K-1237
2 E. South St.
Galesburg, IL 61401

1 Zachary D. Rhodes
395 Oak Creek Drive
Apartment 411
Wheeling, IL 60090

1 Matthew Swank
Knox College Box K-1669
2 E. South St.
Galesburg, IL 61401

20	MS 1327	Vitus J. Leung, 1464
1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories