Faster high-quality processor allocation

Peter Walker [1], David P. Bunde [1], Vitus J. Leung [2]

[1] *Knox College*
pwalker@knox.edu
dbunde@knox.edu

[2] *Sandia National Laboratories*
vjleung@sandia.gov

***Abstract*—We examine fast algorithms to allocate processors to compute jobs in mesh-connected clusters. We find that a 1D curve-based strategy can give allocations of comparable quality to a fully 3D algorithm MC1x1 using a snake curve that goes along the mesh's short dimensions first. We also propose several buddy-system strategies, the best of which actually finds better allocations than MC1x1 if the job sizes and mesh dimensions are powers of 2. Furthermore, these algorithms are much faster than MC1x1, which takes more than 200 times as long in some cases.**

I. Introduction

When users submit a job to a Linux cluster, they specify the number of processors it requires and also give an estimated running time. This estimate serves as a maximum allowed time; jobs still running after their estimated running time are killed. The run time system takes submitted jobs and is responsible for deciding when to run them and which processors to assign to each job. In both research and actual systems, these decisions are typically made in 2 steps. First, the *scheduler* decides when to run each job. When the scheduler has decided to start a job, the *allocator* is responsible for assigning it to specific processors. Typically, the scheduler makes its decision based only on the number of processors available, ignoring which specific processors are available, so there is no interaction between these stages.

This paper is concerned with the allocator. The quality of an allocation on a mesh computer can have a significant effect on job running time; previous work has shown that hand-placing a pair of high-communication jobs into a high-contention configuration can roughly double their running times [1]. The placement of job tasks has been shown to speed up an actual application by up to 1.64 times [2]. As vendors return to building Linux clusters with mesh interconnects (e.g. [3]), high-quality processor allocation is again necessary for good performance on mesh-connected clusters. Furthermore, with the exponential growth in the number of cores on chips, high-quality allocation of cores will be necessary for good performance on mesh-connected chips (e.g. [4]) as well.

To minimize both latency and contention, the allocator's goal is to give each job a set of nearby processors. An ideal allocation is contiguous, but using only contiguous allocations lowers system utilization [5]. Reduced contention is not sufficient to compensate for this utilization drop [6]. Thus, research has focused on noncontiguous allocators (e.g. [7], [8], [1], [6], [9], [10]), which do not explicitly restrict the type of allocations found.

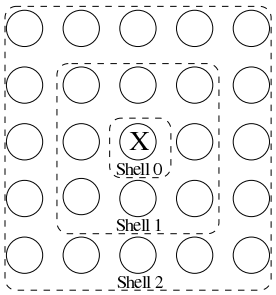


Fig. 1. MC1x1's shells around processor X

A natural algorithm for high-quality mesh allocation is MC1x1 by Bender et al. [7]. This algorithm assumes a mesh architecture. For simplicity we focus on a 2D mesh without wraparound, but the algorithm easily generalizes to other cases. MC1x1 assigns a *score* to each free processor indicating the quality of the candidate allocation centered on that processor. The free processor with the lowest score is selected and its allocation is used.

To generate the allocation for a particular center c, MC1x1 searches for free processors in *shells* around the center. Shell 0 is the single processor c. Shell 1 is the processors whose coordinates are each within 1 of c's coordinates. Larger-numbered shells are defined similarly. Thus, the shell number of a processor in MC1x1 is simply its L_∞ distance from the center, defined in 2D as $L_\infty(c,p) = \max(|c_x - p_x|, |c_y - p_y|)$ where c is the center, p is the other processor, and the subscript denotes the corresponding coordinate (x or y) of that point. Figure 1 illustrates MC1x1's shells around processor X.

MC1x1 selects as many free processors from a shell as possible before considering higher-numbered shells. An allocation's score is the sum of its processors' shell numbers. The drawback of MC1x1 is that it requires scoring every possible center to make an allocation decision. In principle, this process can be sped up by having each idle processor score itself in parallel, but the scoring process can still be relatively time consuming since each candidate center must identify its k nearest idle neighbors. The results of these computations must then be compared to select the winning allocation. As machines get larger, a slow allocation algorithm will eventually limit system performance.

With this motivation, we look at 2 alternative approaches to allocation, seeking a faster algorithm that gives allocations of comparable or better quality. The first approach we consider is curve-based allocation, previously proposed by Lo et al. [6] and Leung et al. [1]. In this approach, processors are ordered according to some curve and allocation decisions are based on the ordered ranks of free processors rather than their mesh coordinates. Thus, allocation is reduced to a 1D problem. Of course information is lost in this reduction, but the idea is that processors close on the curve will also be close in the mesh so enough information will be preserved.

The other approach we consider is based on buddy systems from memory allocation, which Lo et al. [6] generalized to 2D. By using a data structure to organize the free processors, this approach avoids having to consider them individually.

The contribution of this paper is threefold.

1) We extend the use of curve-based strategies to non-square meshes by comparing allocation qualities for a variety of different curves. These curves are evaluated using experiments on the Red Storm [11] test machine, a Cray XT3/4 cabinet, as well as simulations.
2) We propose generalizations of MBS and use simulations to compare them to MC1x1 and curve-based strategies. One of our generalizations, called Granular MBS, outperforms all the other algorithms when the mesh dimensions and job sizes are powers of 2.
3) We measure the running time of all these algorithms, showing that the curve-based and buddy-based strategies do indeed run much faster than MC1x1.

Our simulations use traces from the Parallel Workloads Archive [12], which gives traces of job submission times, sizes, running times, and estimated running times for a variety of HPC systems. Because detailed information on the applications is not included, we do not model the affect of allocation quality on running time; in our simulations, all jobs run for the actual time recorded in their trace. Instead, we evaluate the allocation quality with the average pairwise distance between processors in each allocation. This metric has been used by a number of other authors (e.g. [13], [14], [15]) and experimentally shown to correlate with running times [1]. Reporting an improvement in pairwise distance instead of modeling the affect of allocation on job running time is conservative since improved allocation can start a virtuous cycle by causing jobs to finish more quickly, reducing contention and allowing later jobs to receive better allocations.

The rest of the paper is organized as follows. We study curve-based algorithms in Section II and buddy-based algorithms in Section III. We compare the relative running times of these algorithms to MC1x1 in Section IV. We review some additional related work in Section V. We conclude with some discussion in Section VI.

II. Curve-based algorithms

We begin our study of fast allocation algorithms with curve-based algorithms. There are 2 key decisions to make when designing these algorithms: what curve to use and how to select free processors from the ordered list. A curve-based algorithm was first proposed by Lo et al. [6], who considered several linear orders, including the row-major and snake curves shown in Figure 2 and "shuffled" versions of these. To select processors, they used a free list.

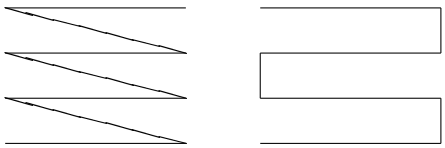


Fig. 2. Some curves used for allocation. (a) Row-major. (b) Snake.

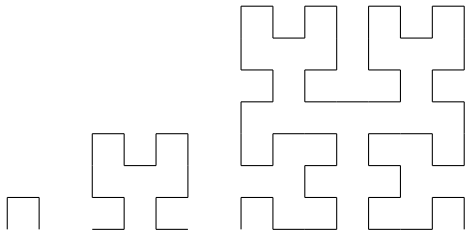


Fig. 3. Hilbert curve

A curve-based strategy was independently proposed by Leung et al. [1]. They ordered processors using a space-filling curve such as the recursively-generated Hilbert curve [16] shown in Figure 3. Although the Hilbert curve is in 2D, it has generalizations to higher dimensions [17]. To select free processors, they adapted the First Fit [18], Best Fit [18], and Sum of Squares [19] strategies from bin packing by treating each interval of free processors as a bin. For example, when using the First Fit strategy to allocate a job requiring k processors, they return the first group of k free processors that are contiguous according to the curve. (When no such group exists, they return the k processors whose difference in ranks are minimal.) In experiments on a Linux cluster, Leung et al. [1] compared this algorithm to the previously used algorithm, which was a free list with processors in row-major order. They found that both the change of ordering and the use of bin-packing heuristics gave improvements, with the curve giving the main improvement.

The main obstacle to widespread use of curve-based algorithms is selecting an appropriate curve. The experiments presented in previous work on these algorithms were primarily on square 2D meshes and generalizing some of the curves is non-trivial. Particularly challenging is the Hilbert curve, whose generation algorithm creates a curve for a square mesh whose side length is a power of 2. Bunde et al. [20] showed that a natural way of using it on a 16×22 mesh leads to poor performance. Even the snake curve presents a choice on a non-square machine since there are 2 possible orientations for the curve.

In this section, we use experiments and simulations to compare the different alternatives for curves. We begin by considering curves for the test machine for Red Storm [11],

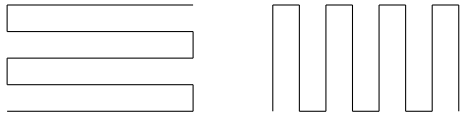


Fig. 4. Snake curves. (a) Row-major snake. (b) Column-major snake

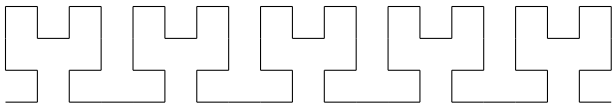


Fig. 5. Spliced curve for 20×4 mesh; five 4×4 Hilbert curves

which consists of a single Cray XT3/4 cabinet. Its processors, running a lightweight Linux, form a 20×4 2D mesh.

As mentioned above, there are 2 choices for a snake curve on a non-square mesh. These are shown in Figure 4. So that the names have consistent meaning across different machines, in this paper we will always list mesh dimensions in non-increasing order. Thus, the row-major snake always traverses the long mesh's long dimension first and column-major snake always traverses its short dimension first.

With a 5:1 aspect ratio, it is not clear what constitutes a Hilbert curve on this mesh. We examined 2 alternatives. We call the first "spliced Hilbert" since it simply splices five 4×4 curves together, as shown in Figure 5. The second, called "Zoltan Hilbert", is the generalized Hilbert curve from the Zoltan library [21]. This curve follows the aspect ratio of the underlying geometry more closely than the spliced alternative. An example curve for a 20×4 mesh is shown in Figure 6.

All runs used identical job streams containing replicas of various-sized instances of a communication test suite. The communication test suite contains all-to-all broadcasts. This test is repeated 25 times in each suite. The suite also computes a variety of statistics, which consumes a small fraction of the total running time. Because locality is most important for jobs with high communication demand, this test suite represents a best-case scenario for the benefits of allocation.

Our test stream had 1,820 jobs of size 2, 660 jobs of size 5, 620 jobs of size 15, and 660 jobs of size 20. This gives a range of "large" (approximately 1/4 or 1/5 of the machine) and small jobs. Small jobs are interspersed among the large ones to cause fragmentation. Since the machine did not have a scheduler, the jobs are scheduled with First-Come First-Served (FCFS). The machine is busy through the last job's release.

We ran the job stream 3 times for each combination of bin packing heuristic and curve. Figure 7 shows the effect of each combination on the makespan of the job stream. For this particular system and job stream, the linear orderings from best to worst are Column-major snake, Spliced Hilbert, Zoltan Hilbert, and Row-major snake.

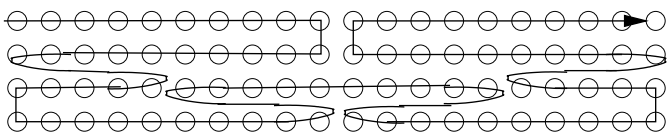


Fig. 6. Hilbert-like curve generated by Zoltan for 20×4 mesh

	Linear Ordering			
Algorithm	Col-major snake	Spliced Hilbert	Zoltan Hilbert	Row-major snake
Best Fit	1.003	1.037	1.160	1.209
Sum of Squares	1.004	1.037	1.154	1.206
First Fit	1.011	1.042	1.164	1.212

Fig. 7. Ratio of average makespan to baseline (MC1x1) in experiments on Cray XT3/4 cabinet. Lower is better.

Trace	Jobs	Processors
NASA-iPSC-1993-2.1-cln.swf	18,239	128
LANL-CM5-1994-3.1-cln.swf	122,057	1,024
LLNL-T3D-1996-1.swf	21,323	256
SDSC-SP2-1998-3.1-cln.swf	54,041	128
LLNL-uBGL-2006-1.swf	19,405	2,048

Fig. 8. Traces considered for linear ordering.

Since it is difficult to vary the dimensions of a real machine, we ran simulations using traces from the Parallel Workloads Archive [12] on meshes with aspect ratios ranging from 1:1 to 16:1. We simulated each trace on a machine of the same size, but ran all simulations on a 2D mesh, ignoring the original machine's topology. The specific traces used are listed in Figure 8. These traces were selected because they came from machines whose number of processors was a power of 2, making it easy to vary the simulated system's aspect ratio. For scheduling, we use EASY [22]. For the Hilbert curve on non-square meshes, we used the spliced Hilbert curve.

The pairwise distances achieved by each algorithm are shown in Figure 9. The results depend on the mesh's aspect ratio, with Hilbert giving the best results for square meshes and the column-major snake giving the best results otherwise. The overlapping results are consistent between these simulations and our experiments.

III. Buddy-based algorithms

Now we turn to buddy-based algorithms, which Lo et al. [6] created as a generalization of the buddy system for memory allocation. Their algorithm divides the processors of a 2D mesh into a hierarchy of square blocks, each having a power of 2 side length. The children of a block are the 4 subblocks formed by splitting it in half along each dimension. The algorithm keeps track of the free blocks of each size. If all 4 children of a block (called "buddies") are ever free, they are removed from the free block list and replaced with their parent. Similarly, blocks in the free list are split to satisfy a request for fewer processors. If a request is not for a power of 4 processors, it is satisfied with several blocks of the desired combined size. Because it may use multiple blocks to satisfy a request, this algorithm is called Multiple Buddy Strategy (MBS). Figure 10 shows the MBS block hierarchy for a 5×4 mesh, with a 4×4 block and four 1×1 blocks at the top level. The 4×4 block has four 2×2 children, each with 4 children of their own.

	NASA-iPSC	LANL-CM5			LLNL-T3D		SDSC-SP2	LLNL-uBGL
	16×8	32×32	64×16	128×8	16×16	32×8	16×8	64×32
Col-major snake BF	2,687	—	239,926	374,717	—	5,854	1,374	5,723,590
Col-major snake FF	2,701	—	241,719	378,303	—	5,908	1,385	5,753,018
Col-major snake freelist	2,733	—	249,856	394,575	—	6,093	1,420	5,754,982
Hilbert BF	2,696	210,704	240,787	374,717	5,190	5,864	1,375	5,736,044
Hilbert FF	2,714	211,652	242,928	378,303	5,217	5,922	1,391	5,782,192
Hilbert freelist	2,742	218,807	250,573	394,575	5,370	6,105	1,424	5,778,155
Row-major snake BF	3,072	225,099	317,199	546,671	5,665	8,093	1,552	7,067,454
Row-major snake FF	3,081	225,995	317,328	545,859	5,693	8,095	1,559	7,082,168
Row-major snake freelist	3,096	230,064	318,105	547,037	5,779	8,157	1,576	7,083,150

Fig. 9. Average sum of pairwise L_1 distances with EASY scheduling.

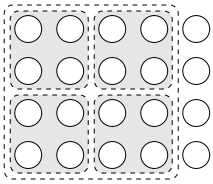
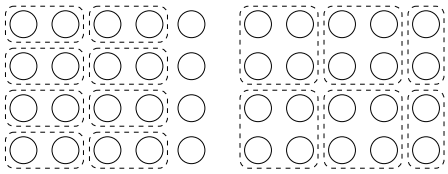


Fig. 10. MBS block hierarchy illustrated for a 5×4 mesh



(a) (b)

Fig. 11. Blocks formed in the first 2 phases of initializing the buddy structure for Granular MBS on a 5×4 mesh

A. Generalizing MBS

There are 2 obvious generalizations of MBS to 3D meshes. The first, which we call *Octet MBS*, performs as MBS except that each block is a cube instead of a square. In this scheme, each subblock has 7 buddies and all the block sizes are a power of 8. The other obvious generalization, again identical to MBS except for how the blocks are divided, computes the block structure for a 2D slice of the system and uses it on each layer of the 3D mesh. As with the original MBS algorithm, each subblock has 3 buddies and all the block sizes are powers of 4. We call this second generalization *Layered MBS*.

Although these schemes are natural, each has drawbacks. Because block sizes are powers of 8 in Octet MBS, a system will have relatively few block sizes and most jobs require multiple blocks. The result is unnecessary dispersal when these blocks are not near each other. Layered MBS avoids this problem, but does not exploit connections between processors in the third dimension since each block is flat. Every job allocated to a single block will get processors lying in a plane.

To avoid both of these pitfalls, we propose a different and better way to divide the machine into blocks. The hierarchy of blocks is built in an iterative way, beginning with each processor being its own block. To build larger blocks, we proceed in a series of phases. In the first phase, each block attempts to find a buddy of the same size and dimensions in the x dimension. This makes blocks with dimensions 2×1×1 composed of 2 matched buddies, as shown in Figure 11(a). For the second phase, blocks seek buddies in the y direction. In the third, they seek buddies in the z dimension. It is important to note that all blocks seek buddies in each phase, even if they failed to find a buddy in the previous phase. Thus, the second phase joins blocks as shown in Figure 11(b), forming both 2×2 blocks and 1×2 blocks. These sequences of 3 phases are repeated until no block succeeds in finding a buddy.

This way of building a block hierarchy has the advantage of creating blocks of each power of 2 size. Thus, the block sizes have finer granularity than even the original MBS algorithm in which all block sizes are powers of 4. In honor of this trait, we call the algorithm that applies the MBS algorithm to this block hierarchy *Granular MBS*. In addition, the scheme handles machines whose dimensions are not powers of 2.

B. Simulations

To evaluate these alternatives, we again use Parallel Workload Archive [12] traces to drive simulations evaluated based on the achieved average pairwise distance. As before, our simulated systems have the same numbers of processors as the machines on which traces were collected, but not the same topologies. Scheduling is performed by EASY [22]. We compare against the curve-based strategy using best fit and the snake curve that goes along dimensions in length order.

Figure 12 shows the relative performance of the curve- and MBS-based algorithms against MC1x1. For each trace, we chose two shapes, one a 2D mesh as square as possible and the other a 3D mesh as close to a cube as possible. The plotted value is the ratio of the algorithm's sum of pairwise distances over the sum of pairwise distances achieved by MC1x1; lower is better and values below 1 occur when the algorithm performs better than MC1x1. The figure also shows the percentage of serial jobs in each trace and the percentage of parallel jobs whose size is a power of 2, 4, or 8.

The results show that Granular MBS is the best performing generalization of MBS. It beats Layered MBS and Octet MBS in all cases, sometimes substantially. Clearly, its fine-grained block sizes pay off. The importance of fine-grained blocking

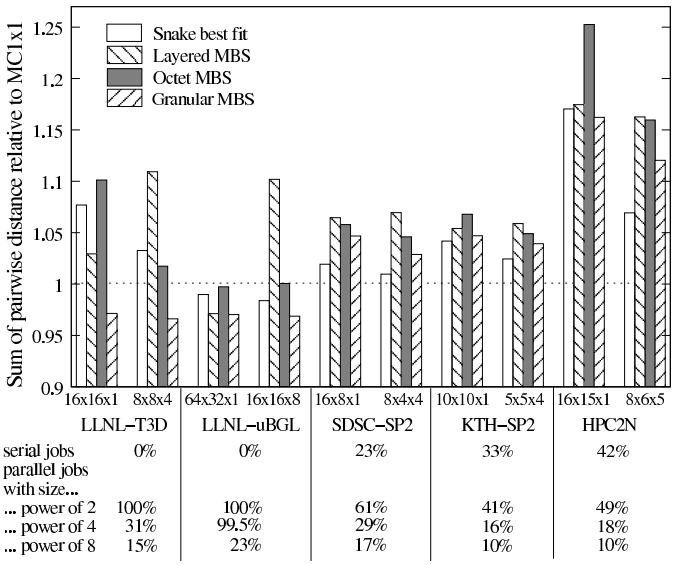


Fig. 12. Performance against MC1x1 for different traces and machine shapes

is further emphasized by the case where Layered MBS comes closest to beating Granular MBS, the $64\times32\times1$ configuration of the LLNL-uBGL trace. Layered MBS does so well in this case because the system allocates processors in groups of 64 [23]. As shown in the figure, this causes more than 99% of the jobs in this trace to have a size that is a power of 4, negating the granularity advantage of Granular MBS. Also relevant is that the configuration is a 2D mesh since that forces both algorithms to allocate flat blocks; Granular MBS wins more handily on the $16\times16\times8$ configuration since a 3D block has smaller average sum of pairwise L_1 distances than a 2D block with the same number of processors.

The use of 3D blocks also benefits Octet MBS, which beats Layered MBS on all the 3D meshes. Octet MBS performs poorly on 2D meshes, however. Because of the way it builds its block hierarchy, Octet MBS uses entirely $1\times1\times1$ blocks on 2D meshes, essentially making it a curve-based strategy without a carefully-selected curve or a bin packing heuristic.

The relative performance of Granular MBS against MC1x1 and the curve-based strategy depends on the trace. Granular MBS beats both in both configurations of the LLNL traces, loses by a small margin in the SDSC-SP2 and KTH-SP2 traces, and then is substantially worse than MC1x1 on the HPC2N trace. We believe this results from a combination of two factors. First of all, Granular MBS performs best when the trace is dominated by jobs whose size is a power of two. These jobs can potentially be allocated using a single block, whereas jobs of other sizes requires a multi-block allocation and potentially causes a large block to split. The percentage of jobs whose size is a power of two roughly corresponds to the observed relative quality of Granular MBS, since the LLNL traces on which it does best are composed entirely of this type of job. The correspondence is not exact, however, since the HPC2N trace has more power-of-two sized jobs than KTH-SP2, but Granular MBS performs better on the latter.

A more complete understanding of Granular MBS's performance comes by considering a second factor. Granular MBS benefits when machine dimensions are powers of two, allowing it to form large blocks. When all machine dimensions are powers of two, the block hierarchy has a single top block encompassing the entire machine, with 2 children each containing half the machine, and so on down the hierarchy. This symmetry increases the chances that a job can be allocated from within a single block and provides maximal flexibility since any subblock can merge if its buddy becomes free. This factor further benefits runs with LLNL-T3D and LLNL-uBGL traces. At the other extreme is the $16\times15\times1$ configuration of the HPC2N trace, whose dimension of length 15 is just shy of a power of two. This means that its top-level block ($8\times8\times1$) is relatively small, containing only 27% of the machine's processors. Thus, most allocations will require processors from the other blocks. Since the other blocks are distributed around the edges of the largest top-level block, this increases the average sum of pairwise L_1 distances.

Now that we have presented two factors as important in determining the performance of Granular MBS, a natural question is which of these factors is more important. The unimpressive performance of the configurations of SDSC-SP2, which has powers of two machine dimensions running jobs with other sizes, suggests that job sizes may be more important than machine dimensions. For other evidence, we compared the results of removing jobs whose size was not a power of two from the traces and running the unmodified trace on a machine whose dimensions are powers of two. When jobs whose size is not a power of two are removed, Granular MBS's average pairwise distance relative to MC1x1 is 1.024 on a 10×10 mesh and 1.016 on a $5\times5\times4$ mesh. The ratio of pairwise distances is 1.093 when the unmodified trace is run on a 16×8 mesh and 1.138 when run on $8\times4\times4$ mesh. Thus, we see that restricting the jobs to have sizes that are powers of two gives better performance relative to MC1x1 than running on a slightly larger machine (128 processors instead of 100) so that its dimensions can be powers of two.

Removing jobs also lets us compare the importance of power of two sized jobs with another possible factor: serial jobs. Jobs of size 1 are also potentially disruptive to the block structure since they can force large blocks to split. To compare the importance of serial jobs and jobs with sizes not powers of two, we looked at the KTH-SP2, SDSC-SP2, and HPC2N traces without each of these types of jobs. As above, we compared the ratios of the sum of pairwise distances of Granular MBS and MC1x1. Both configurations of the KTH-SP2 and SDSC-SP2 traces gave better relative performance when removing non-power-of-two-sized jobs than removing serial jobs. The reverse was true for the configurations of the HPC2N trace, though the ratios were close. These results suggest that whether job sizes are powers of two is more important than whether there are a large number of serial jobs. Removing serial jobs did help in nearly all cases, however. Their large share of the HPC2N trace may help explain why Granular MBS does so poorly on that trace.

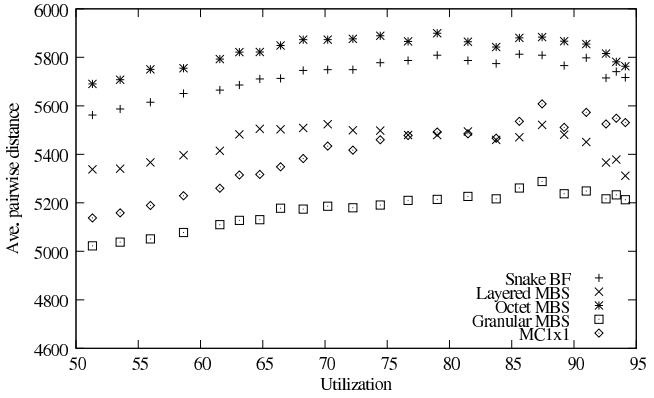


Fig. 13. Ave. pairwise distance for LLNL-T3D trace on 16×16×1 mesh

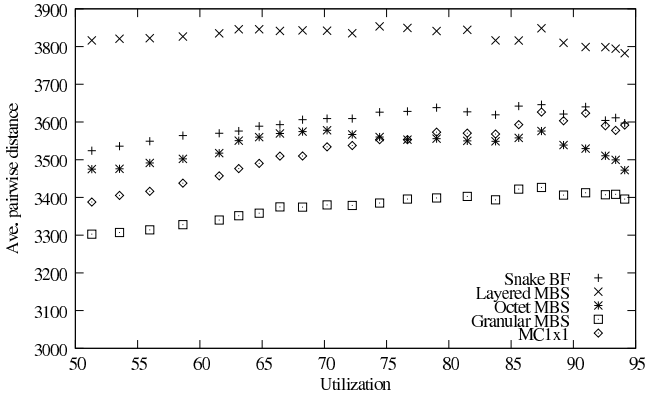


Fig. 14. Ave. pairwise distance for LLNL-T3D trace on 8×8×4 mesh

We were encouraged that Granular MBS performs so well, at least within its specific domain. To see how the algorithms respond to varying system load, we ran simulations with job arrival times multiplied by different constants. This process creates traces with different utilizations while preserving the trace's essential characteristics (idle periods, workload varying throughout the day and week, etc). We note that it is not entirely uncontroversial since it alters relationships between job parameters, the daily cycle, and system load [24], but we use fairly moderate multiplier values (between 0.55 and 1.2).

The results are shown in Figures 13–16, which plot the achieved utilization against average pairwise distance. The points are closer together at high utilization because changing the inter-arrival times has less effect on utilization as the machine saturates. While we expect the overall trend to be that average pairwise distance increases with utilization, the results show some large violations of this expectation. To explain this, we note that these large violations do not occur when FCFS scheduling is used. Thus, we attribute them to small jobs (which tend to backfill easily) running earlier on "spare" processors. Since allocating a small job can require that an MBS-based strategy break up large blocks, these jobs can be very disruptive to other jobs, which may then be allocated using multiple blocks. By moving these jobs earlier in the schedule, the disruption is removed and the average pairwise distance improves even though the average utilization is higher. Further support for this theory is that the largest and most frequent violations of the expected trend occur with Octet MBS, whose block sizes increase most rapidly.

Another surprise in the data is that Granular MBS and Layered MBS exhibit essentially the same performance on the 64×32×1 machine running the LLNL-uBGL trace (Figure 16). This occurs because all jobs on the BlueGene system have sizes that are multiples of 64 [23]. This factor of the system combined with how this particular system was used makes nearly all job sizes a power of 4 (see the statistics in Figure 12). For these jobs, there is no difference between Layered MBS and Granular MBS on a two-dimensional machine.

Though the picture is clouded by these anomalies, the general trend is that pairwise distance increases with utilization. This is less so for Granular MBS on the LLNL-uBGL trace

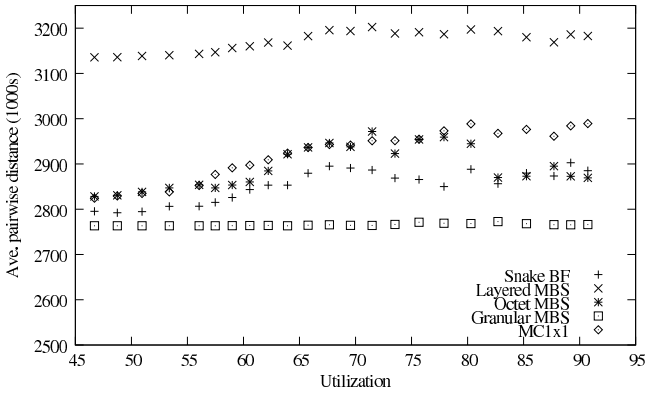


Fig. 15. Ave. pairwise distance for LLNL-uBGL trace on 16×16×8 mesh

since that trace is so well suited to the algorithm. Granular MBS's best observed improvement over MC1x1 is 8.3%, achieved by that trace at 67.5% utilization on a 64×32 mesh. (We observed a 9.6% improvement with FCFS scheduling.)

IV. Relative running times

Now that we have established that the faster algorithms find allocations that are comparable or better, we quantify their speed advantage. To do this, we timed simulation runs performed on an otherwise-idle MacBook Pro with a 2.16GHz Intel Core Duo. The times in seconds for a variety of runs are shown in Figure 17. These times are for the entire simulation

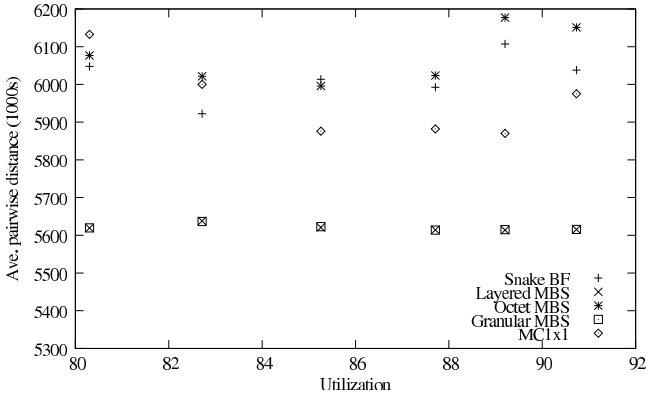


Fig. 16. Ave. pairwise distance for LLNL-uBGL trace on 64×32×1 mesh

	HPC2N		LANL-CM5	
	16×16	8×8×4	32×32	16×8×8
MC1x1	1,130	984	18,059	16,871
Snake BF	67	69	203	195
Layered MBS	43	42	88	89
Octet MBS	44	44	225	92
Granular MBS	42	42	90	91

Fig. 17. Times in seconds to run simulations

so they include time to read the trace, make scheduling decisions, etc. Since these components of the running time are the same for each allocator, the table actually understates the relative speed of the faster algorithms. In addition, the timed version of MC1x1 simply returns the first processors it finds at the appropriate distance, making it faster than the version whose pairwise distance results are reported elsewhere in the paper, which is slightly more sophisticated. Again, this tends to understate the relative speed of the faster algorithms.

Figure 17 clearly shows that the "faster" algorithms considered in this paper are indeed much faster than MC1x1, by a factor of up to 27 on the HPC2N trace and 205 on the LANL-CM5 trace. The curve-based strategies are consistently slower than the buddy systems, but still much faster than MC1x1. Though the absolute magnitude of the time per job is not high (18,059/122,057=0.15 seconds), these results show that MC1x1's running time grows fairly quickly with machine size; from 0.0056 seconds/job for the 16×16 mesh to 0.15 seconds for the 32×32 mesh. This shows how allocation speed can become an issue as machines grow. Using these algorithms to allocate cores on a multicore chip would make the difference immediately important by shrinking the timescale involved.

Figure 17 also shows that the running time of MC1x1 is sensitive to the machine shape as well as its size. It runs significantly faster on the cube-like shapes than the flat ones. We attribute this to the fact that using a cube-like shape makes the dimension lengths smaller, allowing MC1x1 to find the needed processors in fewer shells. The curve- and MBS-based algorithms are relatively insensitive to the system's shape. The exception is the anomalously high running time for OctetMBS with the LANL-CM5 trace on the 32×32 mesh. We cannot explain this beyond the previous observation that OctetMBS is not really appropriate for a flat machine.

V. Other related work

Now we discuss other algorithms for allocation and related problems. Of particular interest is the MC algorithm by Mache et al. [10] and its relatives. MC is the algorithm upon which MC1x1 is based. MC assumes that users submit jobs with a desired shape. For example, instead of a job requesting 6 processors, it can request a 2×3 shape. The runtime system is not obligated to find a submesh of this shape, but the hope is that the extra information allows the system to find a group of processors whose topology is similar to the job's actual communication pattern. The MC algorithm uses this information by making shell 0 have the desired dimensions. Subsequent shells expand by one in each direction as in MC1x1. Other than the initial shell, the two algorithms are the same, with MC1x1 being MC with a 1×1 initial shell.

Also related to MC1x1 is a family of algorithms parameterized by the candidate centers used, how candidates allocations are formed around these centers, and how candidate allocations are evaluated. Another member of this family is Gen-Alg, by Krumke et al. [15]. Gen-Alg's set of candidate centers are the idle processors, just as in MC1x1. It chooses processors based on the number of hops from the center. In a mesh, the number of hops is the same as L_1 (Manhattan) distance, which makes Gen-Alg search in a diamond shape. Each candidate allocation is evaluated using its sum of pairwise distances. Krumke et al. [15] showed that Gen-Alg always finds an allocation whose sum of pairwise distances is within $(2 - 2/k)$ of the best possible value, where k is the job size.

Also in the same family of algorithms as MC1x1 is MM, proposed by Bender et al. [7]. MM evaluates candidate centers in the same way as Gen-Alg, but considers more of them. Specifically, a candidate center is any processor sharing x, y, and z coordinates with (possibly different) free processors. Bender et al. [7] showed that MM always finds an allocation whose sum of pairwise distances is within a factor of $2-1/(2d)$ of the best possible value in a d-dimensional mesh. (This approximation factor is $7/4$ in a two-dimensional mesh and $11/6$ in a three-dimensional mesh.) They also gave a PTAS, an algorithm to that solves within a factor of $1+\epsilon$ in time polynomial in the mesh size and $1/\epsilon$ for any $\epsilon > 0$.

The first curve-based allocation algorithm was Paging by Lo et al. [6]. Paging divides the machine's processors into blocks and uses the curve to order the blocks. When an allocation is needed, the first free blocks in this list are used. In this work, we restrict our attention to "blocks" consisting of a single processor to avoid internal fragmentation, when a job is allocated more processors than it needs. (We also use the bin packing heuristics as proposed by Leung et al. [1] rather than always taking the first free processors in the list.) Using a larger blocks has the potential advantage of reducing average pairwise distance, however, since the processors in a block are guaranteed to be close together. Mache et al. [25] give another curve-based strategy that strives to minimize network contention caused by I/O as well as intrajob communication.

Some allocation algorithms are neither center-based nor curve-based. In particular ANCA [8] and GABL [26] both work from the job rather than a representation of the free processors. They first find a contiguous allocation for as much of the job as possible, then repeat for any remaining processors needed. We did not consider these algorithms because, like MC, they assume that jobs come with desired dimensions.

Slightly less related are algorithms that only give contiguous allocations, delaying jobs until one is available if necessary. For example, the single buddy algorithm [6] maintains a hierarchy of blocks as in MBS except that a job is only allocated if a single block is big enough. When an eligible block is found, the job gets the entire block. Contiguous allocation algorithms eliminate contention between jobs, but

they have been repeatedly shown to limit overall system utilization (e.g. [5], [27]). Despite this, they are needed for some systems such as BlueGene/L which guarantee each job an appropriate submesh or sub-tori, a task facilitated by the presence of extra network links [23].

A problem that is related to processor allocation is task mapping. In task mapping the goal is to assign the tasks of a job to a group of preselected processors in a way that minimizes contention. This is an old problem [28], but one that continues to attract attention (e.g. [29], [2], [30], [31]).

VI. Discussion

We have shown that our faster allocation algorithms can find allocations whose quality is comparable, and in some cases better than, the best truly three-dimensional algorithm. In particular, the curve-based algorithm with a curve that goes along the smallest dimension first is worthy of consideration on any mesh. Our Granular MBS algorithm is even better when the mesh dimensions and expected job sizes are powers of two. The faster algorithms also run as much as 200 times faster.

Possible future work is to try making Granular MBS more broadly applicable so systems outside of its special case could benefit. Alternately, the algorithm could be improved by using a bin packing heuristic or an MC1x1-like search to assign blocks from the free list. Also interesting would be other ideas for fast but effective allocation algorithms.

Acknowledgments

P. Walker and D.P. Bunde were partially supported by contract 899808 from Sandia National Laboratories. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract No. DE-AC04-94AL85000. We also thank Zaid Abudayyeh, Bob Heaphy, Barry Jack Oliphant, and Kevin Pedretti for work supporting the experiments on the Cray XT3/4 cabinet.

References

[1] 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*, 2002, pp. 296–304.

[2] 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*, 2006.

[3] Top500.org, "Red sky," http://www.top500.org/system/10188.

[4] J. Balfour and W. Dally, "Design tradeoffs for tiled cmp on-chip networks," in *Proc. 20th Intern. Conf. Supercomputing*, 2006, pp. 187–198.

[5] P. Krueger, T.-H. Lai, and V. Dixit-Radiya, "Job scheduling is more important than processor allocation for hypercube computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 5, pp. 488–497, 1994.

[6] V. Lo, K. Windisch, W. Liu, and B. Nitzberg, "Non-contiguous processor allocation algorithms for mesh-connected multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 7, pp. 712–726, 1997.

[7] 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*, vol. 50, no. 2, pp. 279–298, 2008.

[8] C. Chang and P. Mohapatra, "Improving performance of mesh connected multicomputers by reducing fragmentation," *J. Parallel and Distributed Computing*, vol. 52, no. 1, pp. 40–68, 1998.

[9] J. Mache, V. Lo, and S. Garg, "Job scheduling that minimizes network contention due to both communication and I/O," in *Proc. 14th Intern. Parallel and Distributed Processing Symp.*, 2000.

[10] J. Mache, V. Lo, and K. Windisch, "Minimizing message-passing contention in fragmentation-free processor allocation," in *Proc. 10th IASTED Intern. Conf. Parallel and Distributed Computing and Systems*, 1997, pp. 120–124.

[11] J. Tomkins and S. Kelly, "Red storm," 2008, http://www.cs.sandia.gov/platforms/RedStorm.html.

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

[13] J. Mache and V. Lo, "Dispersal metrics for non-contiguous processor allocation," University of Oregon, Technical Report CIS-TR-96-13, 1996.

[14] ——, "The effects of dispersal on message-passing contention in processor allocation strategies," in *Proc. 3rd Joint Conf. on Information Sciences, Sessions on Parallel and Distributed Processing*, vol. 3, 1997, pp. 223–226.

[15] S. Krumke, M. Marathe, H. Noltemeier, V. Radhakrishnan, S. Ravi, and D. Rosenkrantz, "Compact location problems," *Theoretical Computer Science*, vol. 181, no. 2, pp. 379–404, 1997.

[16] D. Hilbert, "Über die stetige abbildung einer linie auf ein flachenstück," *Math. Ann.*, vol. 38, pp. 459–460, 1891.

[17] J. Alber and R. Niedermeier, "On multi-dimensional Hilbert indexings," *Theory of Computing Systems*, vol. 33, pp. 295–312, 2000.

[18] D. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1973.

[19] J. Csirik, D. Johnson, C. Kenyon, J. Orlin, P. Shor, and R. Weber, "On the sum-of-squares algorithm for bin packing," *J. ACM*, vol. 53, no. 1, pp. 1–65, 2006.

[20] D. Bunde, V. Leung, and J. Mache, "Communication patterns and allocation strategies," in *Proc. 3rd Intern. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2004.

[21] E. Boman, K. Devine, L. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, and W. Mitchell, "Zoltan home page," 1999, http://www.cs.sandia.gov/Zoltan.

[22] D. Lifka, "The ANL/IBM SP scheduling system," in *Proc. 1st Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS, no. 949, 1995, pp. 295–303.

[23] 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*, vol. 49, no. 2/3, p. 425, 2005.

[24] E. Frachtenberg and D. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *Proc. 11th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS, no. 3834, 2005, pp. 257–282.

[25] J. Mache, V. Lo, and S. Garg, "The impact of spatial layout of jobs on i/o hotspots in mesh networks," *J. Parallel and Distributed Comput.*, vol. 65, no. 10, pp. 1190–1203, 2005.

[26] S. Bani-Mohammad, M. Ould-Khaoua, I. Abaneh, and L. Mackenzie, "An efficient processor allocation strategy that maintains a high degree of contiguity among processors in 2d mesh connected multicomputers," in *Proc. ACS/IEEE Intern. Conf. Computer Systems and Applications*, 2007.

[27] ——, "A performance comparison of the contiguous allocation strategies in 3D mesh connected multicomputers," in *Proc. 5th Intern. Symp. Parallel and Distributed Processing and Applications*, ser. LNCS, vol. 4742, 2007, pp. 645–656.

[28] S. Bokhari, "On the mapping problem," *IEEE Trans Computers*, vol. C-30, no. 3, 1981.

[29] 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*, 2004, p. 57.

[30] 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.

[31] A. Bhatele and L. Kale, "Benefits of topology-aware mapping for mesh topologies," *Parallel Processing Letters*, vol. 18, no. 4, pp. 549–566, 2008.