A Tie-Breaking Strategy for Processor Allocation in Meshes

Christopher R. Johnson [1], David P. Bunde [1], Vitus J. Leung [2]

[1] *Department of Computer Science*
Knox College
Galesburg, IL USA
`crjohnso@knox.edu`
`dbunde@knox.edu`

[2] *Computer Science & Informatics Department*
Sandia National Laboratories
Albuquerque, NM USA
`vjleung@sandia.gov`

***Abstract*—Many of the proposed algorithms for allocating processors to jobs in supercomputers choose arbitrarily among potential allocations that are "equally good" according to the allocation algorithm. In this paper, we add a parametrized tie-breaking strategy to the MC1x1 allocation algorithm for mesh supercomputers. This strategy attempts to favor allocations that preserve large regions of free processors, benefiting future allocations and improving machine performance. Trace-based simulations show the promise of our strategy; with good parameter choices, most jobs benefit and no class of jobs is harmed significantly.**

I. Introduction

This paper focuses on the processor allocation strategy known as MC1x1. A processor allocation strategy is used to place a parallel job on specific nodes of a massively parallel cluster or supercomputer once the job has been scheduled to run. MC1x1 targets mesh-connected systems. At Sandia National Labs, there is a long history of mesh-connected systems in the top 500 lists of supercomputer sites [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], including the Intel Paragon [1], [2], [3], [4], [6], [7], Cray XT [13], [14], [16], and Sun Blade System [15] families of computers. After Intel discontinued the Paragon, it appeared that the market for mesh-connected systems had disappeared, and Sandia had to build their own mesh-connected systems in the form of their Cplant clusters [5], [8], [9], [10], [11], [12]. However, with the Cray XT and Sun Blade System, it appears that the market for mesh-connect systems has returned.

MC1x1 was originally implemented and tested on Sandia's Cplant clusters. In its final incarnation, the compute partition of Cplant was composed of 1,536 Compaq DS10L 1U servers in a 3D mesh with wrap-around in two dimensions. The implementation was then ported to Red Storm, a Cray XT3/4. The compute partition of Red Storm consists of 12,960 AMD 64 bit Opteron processors in a 3D mesh with wrap-around in one dimension.

When users submit a job to such a system, 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 each job. In both research papers and actual systems, these decisions are typically made in two stages. First, the *scheduler* decides when to run each job. Then, when the scheduler decides to start a job, the *allocator* is responsible for assigning it 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 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 [17]. In order to minimize both latency and contention, the allocator's goal is to give each job a group of processors close together. An ideal allocation is contiguous, but using only contiguous allocations lowers system utilization [18]. This additional idle time lowers system performance even when message-passing contention is taken into account [19]. Thus, most research has focused on noncontiguous allocators [20], [21], [17], [19], [22], [23], which attempt to assign each job a group of nearby processors, but have no explicit restrictions on the types of allocations found.

This paper focuses on the noncontiguous allocation algorithm MC1x1, introduced by Bender et al. [20], which assumes the processors are organized as a mesh. For simplicity, we focus on a 2D mesh without wraparound, but it easily generalizes to three dimensions or meshes with wraparound. For each free processor, MC1x1 finds a candidate allocation centered on that processor. It assigns a *score* to each candidate allocation and takes the allocation with the lowest score. To generate the allocation for a particular center c, MC1x1 searches for free processors in *shells* around that center. Shell 0 is the processor itself. Larger shells are defined recursively: a previously-unassigned processor is in shell $i + 1$ if each

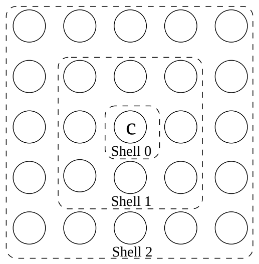

Fig. 1. Shells generated by MC1x1 around processor c.

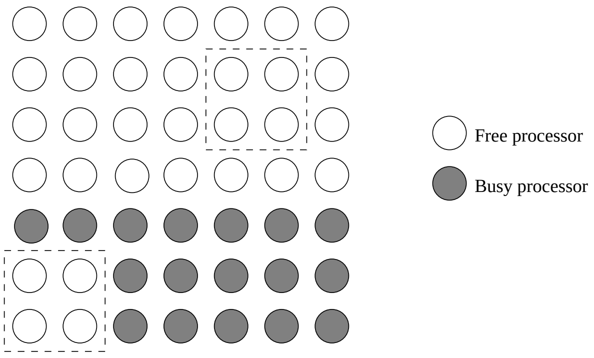

Fig. 2. Two possible allocations for a four-processor job. MC1x1 assigns them the same score.

of its coordinates differ by at most 1 from a processor in shell i. More concisely, the shell number of a processor in MC1x1 is simply its L_∞ distance from the center, defined in two dimensions as follows:

$$L_\infty(c,p) = max(|c_x - p_x|, |c_y - p_y|)$$

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 these shells. MC1x1 selects as many free processors from a shell as possible before considering higher-numbered shells and the score assigned to a potential allocation is the sum of the shell numbers of its processors.

MC1x1 does not specify what to do if more than one candidate allocation gets the same score. A straightforward implementation would simply take the first allocation found that has the best value, but we propose to add a specific tie-breaking procedure. To see how deliberate tie-breaking may benefit the system, suppose the system is as shown in Figure 2 when a four-processor job is scheduled. The two groups of circled processors are equally good potential allocations as far as MC1x1 is concerned and there are many others as well. The allocation on the lower left, however, preserves the large group of free processors and can be expected to benefit future allocations, particularly large jobs.

Although the particular situation shown in Figure 2 is a contrived case, ties are actually quite common. To measure their frequency, we ran simulations using traces from the Parallel Workloads Archive [24]. Figure 3 lists the specific traces and machine configurations used.[1] These traces were selected because they came from machines whose number of processors were perfect squares. For our simulations, we assumed that each machine was a square mesh. This differs from the actual machine configurations, but was done to make the results comparable between machines by eliminating any differences in topology. For scheduling, we use First-Come First-Served (FCFS) and EASY [25]. FCFS is simple and is commonly used in allocation research, but EASY allows higher utilization and is more realistic for production systems. It allows jobs to run before their turn provided that doing so does not delay the first job in the queue. Running jobs out of order in this way is called *backfilling*. EASY is also called *aggressive backfilling* because it exploits nearly all its backfilling opportunities, though the single constraint that the first job not be delayed suffices to ensure that no job is delayed indefinitely [26].

[1][24] credits the LANL-CM5 trace with 3 more jobs. We omit these jobs because their number of processors is -1 (i.e. "unknown").

The results are shown in Figure 4. As you can see, ties occur in over 36% of the allocations for each of our simulations, and in most cases, they occur in over 60% of the allocations. Nor were two- or three-way ties typical; the average number of potential allocations with the same score was between 16 and 138. These numbers imply that the MC1x1 algorithm has a great deal of flexibility that has not previously been exploited. This paper begins to do so.

The rest of this paper is organized as follows. In Section II, we present our tie-breaking strategy. In Section III, we discuss the improvements achieved, both overall and for each size of job. In Section IV, we discuss related work. Finally, in Section V, we discuss possible ways to continue and extend this work.

II. Tie-breaking strategy

Now we describe our strategy for breaking ties. We designed the strategy by identifying properties of allocations that tend to leave the machine in a good state and measuring how well candidate allocations meet these properties. The scores from different properties are then combined into a *tie-breaking score*. MC1x1 resolves ties by selecting the allocation with the lowest tie-breaking score. By tie-breaking in this manner, we seek to preserve good places for future allocations without impacting the job being allocated. We identified the following three properties of good allocations:

1) Good allocations are near few free processors. This property prevents large contiguous free areas from being broken up and favors allocations that fill in smaller regions of available processors.
2) Good allocations are near walls when possible. This prevents the allocations from filling up the area in the middle of the machine before it is needed. Having these processors free increases the chances that large groups of free processors will be available in the future.
3) Good allocations border other allocations. Being far from other allocations is actually good for a specific allocation, but can result in fragmented groups of processors.

Trace	Jobs	Processors	Machine used
KTH-SP2-1996-2.swf	28,489	100	10×10 mesh
LLNL-T3D-1996-1.swf	21,323	256	16×16 mesh
SDSC-Par-1995-2.1-cln.swf	53,970	400	20×20 mesh
SDSC-Par-1996-2.1-cln.swf	32,135	400	20×20 mesh
LANL-CM5-1994-3.1-cln.swf	122,057	1024	32×32 mesh

Fig. 3. Traces considered for tie-breaking.

	FCFS			EASY		
Trace	# ties	% allocations tied	Ave. # tied	# ties	% allocations tied	Ave. # tied
KTH-SP2	17,611	61.8	16.7	17,590	61.7	22.7
LLNL-T3D	14,839	69.6	66.7	15,778	74.0	74.6
SDSC-Par95	34,941	64.7	60.8	37,051	68.7	88.6
SDSC-Par96	20,096	62.5	79.0	20,015	62.3	67.7
LANL-CM5	48,915	40.1	110.2	44,186	36.2	137.9

Fig. 4. Ties for the traces using MC1x1 allocator. Gives number of allocation decisions with a tie, percent of all allocation decisions that have a tie, and average number of allocations with same score (averaged over the allocation decisions with ties).

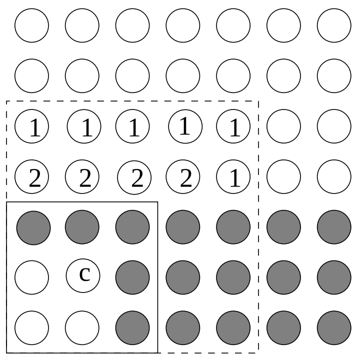

Fig. 5. Terms in the available score for four-processor allocation centered at “c”. The solid line is the allocation’s outermost shell and the dashed line marks the edge of a scan radius of 2.

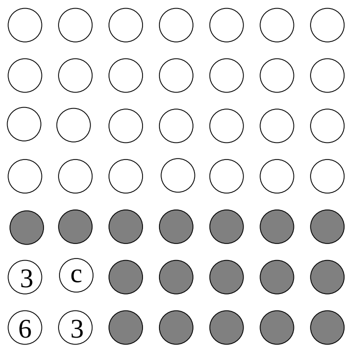

Fig. 6. Terms in the wall score for the potential allocation of four processors centered at the “” with scan radius two.

Each of these factors could be used separately as a measure of allocation quality, but we use a linear combination of them for our tie breaking. Now we describe how these properties are measured and how their scores are combined.

a) Available factor: The first property is captured by the *available score*, which penalizes allocations that are near other free processors. The available score is the sum of penalties assigned to unused free processors near the center. The amount of the penalty and also the definition of “near” is defined in terms of the *scan radius*, which is the number of extra shells examined beyond the farthest shell required by the allocation. For example, see Figure 5, which focuses on one candidate allocation of a four-processor job. The candidate center is marked with “c”. One shell, denoted with the solid line, suffices to get 4 processors. The dashed line shows a scan radius of 2, meaning two additional shells are examined. The numbered processors are free processors within the scan radius that are not needed for the allocation.

The shell that is scan radius shells beyond the allocation’s furthest shell is called the *max shell* since it is the highest-numbered shell considered. A processor’s penalty depends on its shell number so that unused free processors near the center are more strongly discouraged. Specifically, the penalty for a processor in the max shell is 1 and the penalty increases by 1 for each additional shell inward. We call this the *reverse L_∞ distance* since it is roughly the L_∞ distance inward from the max shell. The penalties for the unused free processors within the scan radius are shown in Figure 5.

b) Wall score: We capture the second property with the *wall score*, which rewards allocations along the walls. Each processor in the allocation is assigned a term equal to its reverse L_∞ distance times the number of walls it borders. The wall score is negative one times the sum of these terms. An example of the terms contributing to the wall score is shown in Figure 6. Note that the corner processor borders two walls so its term is twice as large. The wall score is negative because we wish to encourage being near walls. Note that which processors are assigned a non-zero term is independent of the scan radius, but that the scan radius controls the magnitude of these terms via the reverse L_∞ distance. Scaling the wall score with the scan radius helps make the available and wall scores of similar magnitude.

c) Border score: To encourage allocations along the walls, the *border score*. is calculated by examining one shell beyond the allocation’s last shell and assigning each currently-

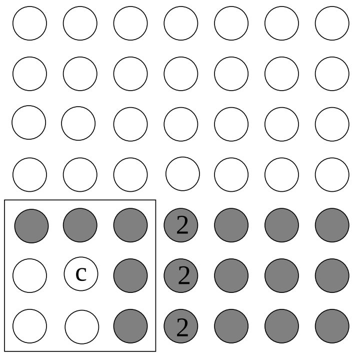

Fig. 7. Terms in border score for a four-processor allocation centered at “c” with scan radius 2. The line shows the allocation’s outermost shell.

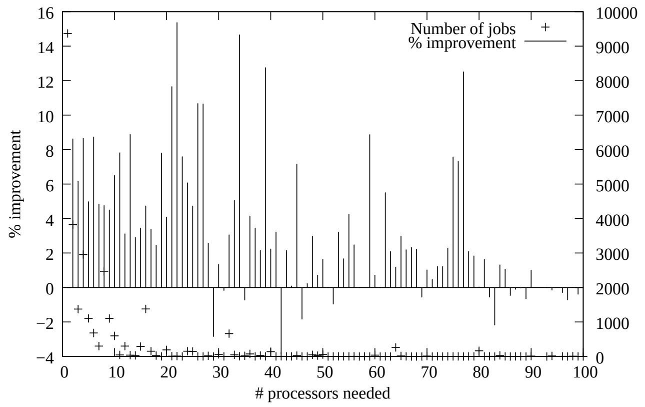

Fig. 9. % improvement by job size for $(2, 13, 20, 6)$ on KTH-SP2

used processor a term equal to its reverse L_∞ distance, as shown in Figure 7. Note that previously-used processors within the allocation’s shells do not contribute to the score to avoid encouraging “swiss cheese” allocations whose bounding boxes contain many processors assigned to other jobs. The border score is negative one times the sum of these terms. Again, this score is negative to encourage allocations that border existing allocations and the scan radius is not directly used, but indirectly scales this score with the others.

d) Combining the scores: As described above, the properties give us three scores: the available score, the wall score, and the border score. The overall tie-breaking score is a weighted sum of these scores. We call their weights the *available factor* (AF), *wall factor* (WF), and *border factor* (BF), respectively. Along with the scan radius (SR), these factors parameterize our tie-breaking strategy. We denote a set of parameter values with the vector (SR, AF, WF, BF).

As explained above, the scan radius changes the area used to search for available processors and the magnitude of the other scores. The factor parameters set the relative importance of the three properties. Since the relative order rather than the actual magnitude of tie-breaking scores is important, (w, x, y, z) behaves the same as (w, cx, cy, cz) for any constant $c \neq 0$.

III. Results

We ran a large number of simulations to identify good parameter values. We did a brute force search of the parameter space for integral factor values 0–10 using scan radius 1–9 for KTH-SP2, 1–11 for SDSC-Par95, 1–11 for SDSC-Par96, and 10–13 for LANL-CM5. We also examined larger scan radii for some traces and ran a search algorithm that tried parameter values near the best results so far. We did not run the brute force search on the LLNL-T3D trace, but ran our “supplementary” search for many values on this trace. Our search was most extensive with the KTH trace since it ran most quickly. Since the LANL-CM5 trace is much slower to run (it is the largest machine and the longest trace), we selected promising scan radii based on results from the smaller traces.

Our simulations show that tie-breaking consistently improves MC1x1 except when the border factor is very high. Figure 8 gives the best values found for each trace.

The overall percent improvements are quite small. This is not the complete picture, however, since some sizes of jobs cannot be improved significantly. For example, jobs whose size is either 1 or the machine size are always perfectly allocated. In several of the traces, these categories include a significant percentage of the jobs; the two categories contain 9,371=32.9% of the jobs in KTH-SP2, 8,224=15.2% of the jobs in SDSC-Par95, and 2,680=8.3% of the jobs in SDSC-Par96. In addition to these categories of jobs that cannot be improved at all, jobs that are so large they use most of the machine see little variation. To see past the influence of these job sizes, we examine the results grouped by job size.

Figure 9 shows an example of this grouping for the KTH-SP2 trace with parameters $(2, 13, 20, 6)$, which provide the best overall improvement for that trace. The horizontal line is the baseline (MC1x1 without tie breaking) and the vertical lines are the percent improvement in average pairwise L_1 distance (left axis) for each job size. The $+$ marks show the number of jobs of each size (right axis). Improvements for more common job sizes are more significant than improvements in sizes containing fewer jobs.

From Figure 9, it is clear that most job sizes had a significant improvement over MC1x1, while some suffered very slightly. Figure 10 highlights how few jobs suffered by ordering the job sizes by the number of jobs they contain. The 36 most common job sizes all improved. The 37th size (with 20 jobs) worsened by less than 1%. The most harmed job size has 42 processors (rank 49), which worsens by only 3.879%.

Unfortunately, for some traces, the tie-breaking strategy with the best improvement does not have such a nice profile. Figure 11 shows the improvement by job size for SDSC-Par96 $(4, 29, 5, 0)$. Though these parameters give the best overall improvement, some job sizes get significantly worse allocations. The greatest worsening is 29.128%, with 5 job sizes getting worse by more than 15%. These are all fairly small job classes, as is apparent when the job sizes are ordered by the number of jobs they contain (see Figure 12).

The top several parameter values for SDSC-Par96 have very similar ratios and all suffer from the 30% worsening exhibited by $(4, 29, 5, 0)$. The first that avoids it is $(2, 12, 1, 1)$, the fourth best overall and the first that uses a non-zero border

Trace	Parameters	Ratios	Ave. Pairwise L_1 Dist.	% Improvement
KTH-SP2	MC1x1	—	575.399	—
	(2, 13, 20, 6)	(1, 1.538, 0.461)	564.865	1.8309
	(2, 36, 50, 15)	(1, 1.388, 0.416)	565.015	1.8046
	(2, 59, 90, 27)	(1, 1.525, 0.457)	565.032	1.8017
LLNL-T3D	MC1x1	—	5,309.586	—
	(3, 9, 3, 2)	(1, 0.333, 0.222)	5,224.970	1.594
	(3, 35, 8, 6)	(1, 0.229, 0.171)	5,225.700	1.580
	(5, 64, 10, 7)	(1, 0.156, 0.109)	5,225.735	1.579
SDSC-Par95	MC1x1	—	5,183.029	—
	(10, 1, 3, 2)	(1, 3.000, 2.000)	5,080.373	1.981
	(7, 13, 1, 0)	(1, 0.077, 0.000)	5,082.621	1.937
	(7, 10, 1, 0)	(1, 0.100, 0.000)	5,082.697	1.936
SDSC-Par96	MC1x1	—	3,484.677	—
	(4, 29, 5, 0)	(1, 0.172, 0.000)	3,330.287	4.430
	(4, 52, 9, 0)	(1, 0.173, 0.000)	3,330.287	4.430
	(4, 23, 4, 0)	(1, 0.173, 0.000)	3,330.311	4.429
LANL-CM5	MC1x1	—	210,310.843	—
	(12, 42, 41, 15)	(1, 0.796, 0.357)	205,386.282	2.342
	(11, 31, 27, 8)	(1, 0.871, 0.258)	205,426.429	2.322
	(12, 28, 27, 10)	(1, 0.964, 0.357)	205,426.429	2.322

Fig. 8. Three best tie-breaking strategy parameter sets found for each trace. Jobs scheduled using FCFS.

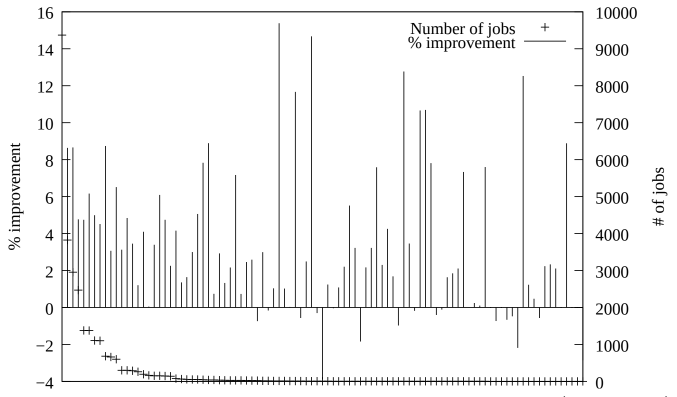

Fig. 10. % improvement by ordinal rank of job frequency for $(2, 13, 20, 6)$ on KTH-SP2

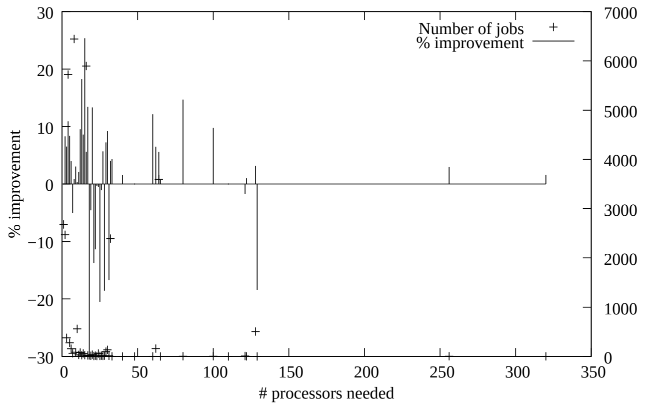

Fig. 11. % improvement by job size for $(4, 29, 5, 0)$ on SDSC-Par96

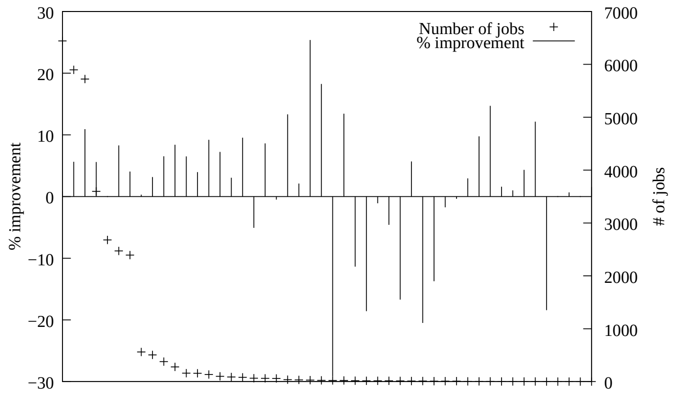

Fig. 12. % improvement by ordinal rank of job frequency for $(4, 29, 5, 0)$ on SDSC-Par96

factor. The performance of this parameter vector is shown in Figures 13 and 14. The greatest worsening for $(2, 12, 1, 1)$ is "only" 20.397%, though note that it does cause a relatively large job class (2,471 jobs; the 6^{th} largest class) to suffer a 3.832% worsening. This is the class of jobs using 2 processors.

It is hard to summarize the best parameter values, though the border factor is nearly always the lowest of the three and the available factor if often the highest. More encouraging is that fact that good solutions generally "cluster", with slightly different ratios giving similar performance. An excellent example of a cluster is the SDSC-Par96 results shown in Figure 8; clusters like this occur throughout the top parameter values for each trace. These clusters reassure us that the allocator quality is a "somewhat smooth" function of the factors.

Since the parameter values giving the best tie-breaking strategy differ for each trace, a natural question is how dependent

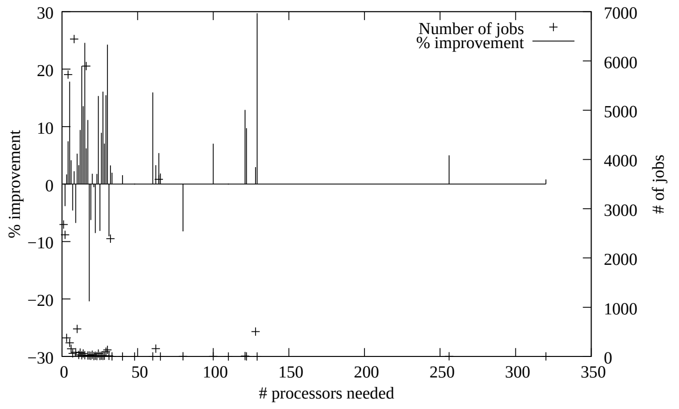

Fig. 13. % improvement by job size for $(2, 12, 1, 1)$ on SDSC-Par96

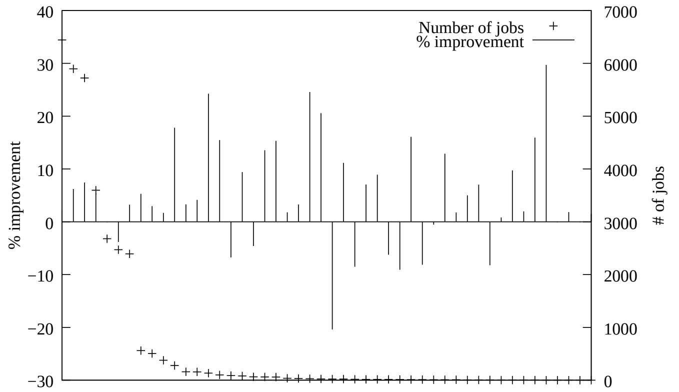

Fig. 14. % improvement by ordinal rank of job frequency for $(2, 12, 1, 1)$ on SDSC-Par96

these values are on the trace. To investigate this, we tested how well the parameters $(2, 13, 20, 6)$, which are best for the KTH-SP2 trace, do on the other traces. We also considered scaling the scan radius with the machine, keeping it around one fifth of the machine width. Figure 15 shows the results. From these, we conclude that the scan radius should be adjusted for machine size; using scan radius equal to 1/5 of the machine width (the bold line) gives a better improvement than leaving it at 2 for all traces except LLNL-T3D, where both the scan radius and improvement are close.

The most important conclusion from Figure 15, however, is that good parameter sets are fairly portable. The exact amount of improvement varies by trace, but adding tie breaking with vector $(\text{SR}, 13, 20, 6)$ improves results on all the traces for both $\text{SR} = 2$ and $\text{SR} = \frac{\text{width}}{5}$ (and all other values we tested). Thus, tie-breaking gives consistent improvement right "out of the box", though greater improvement may be possible by tuning to the specific system and job characteristics.

Further evidence of our system's portability was provided by our examination of the effect of EASY scheduling. We took the best parameter vector for each trace under FCFS scheduling (from Figure 8) and reran these with EASY. The results appear in Figure 16. For all traces except SDSC-Par96, tie-breaking actually gives a larger improvement with EASY than FCFS. (Note that tie-breaking is still better for the SDSC-Par96 trace under EASY scheduling.) That tie-breaking is

Trace	SR	Ave. Pairwise L_1 Dist.	% Imp.
LLNL-T3D	2	5,247.661	1.166
	3	**5,255.086**	**1.026**
	4	5242.902	1.256
SDSC-Par95	2	5,136.914	0.890
	3	5,099.893	1.604
	4	**5,113.313**	**1.345**
	5	5,106.607	1.474
SDSC-Par96	2	3,377.613	3.072
	3	3,353.929	3.752
	4	**3,353.731**	**3.758**
	5	3370.576	3.274
LANL-CM5	2	206,426.347	1.847
	3	206,373.745	1.872
	6	**206,011.123**	**2.044**
	7	206,031.378	2.035

Fig. 15. Effect of tie-breaking with $(SR, 13, 20, 6)$. Bold values correspond to a scan radius of 1/5 of the machine width (rounded to the nearest integer). The last column gives the % improvement over MC1x1 without tie-breaking.

helpful under EASY scheduling is not entirely unexpected since backfilling increases utilization, making allocation harder and giving more room for improvement, but that the same parameters work so well illustrates their portability.

IV. Related Work

Now we describe related work in processor allocation. MC1x1 was developed as a variant of the algorithm MC, introduced by Mache et al. [23]. MC assumes that jobs comes with a desired shape, so a job might request a 2×3 submesh rather than 6 processors. MC uses the desired shape as the size of shell 0, but otherwise is identical to MC1x1. Since Sandia users do not specify desired shapes, this algorithm does not apply to the systems we consider.

Also related to MC1x1 are a family of similar algorithms, proposed independently by different researchers. We call these algorithms *center-based*. Each center-based algorithm considers a list of candidate centers, evaluating each by searching in shells until enough free processors are found and then scoring the resulting candidate allocation. In these terms, the MC1x1 algorithm uses free processors as the candidate centers, builds shells containing processors with the same L_∞ distance from the center, and scores a candidate allocation by summing its shell numbers (i.e. the L_∞ distance of each processor to the candidate center).

Krumke et al. [27] proposed another algorithm in this family called Gen-Alg. Gen-Alg uses the free processors as candidate centers just like MC1x1, but it finds the candidate allocation and computes a candidate allocation's score differently. To find the candidate allocation around a center, Gen-Alg identifies processors that are closest to the center in terms of L_1 distance, which is equivalent to selecting from diamond-shaped shells as shown in Figure 17. The score of a candidate allocation is the sum of pairwise L_1 distances between its processors. Krumke et al. [27] were explicitly trying to minimize the sum

Trace	Parameters	Ave. Pairwise L_1 Dist.	% Improvement
KTH-SP2	MC1x1	575.399	—
	(2, 13, 20, 6)	559.623	2.742
LLNL-T3D	MC1x1	5260.370	—
	(3, 9, 3, 2)	5,167.365	1.768
SDSC-Par95	MC1x1	5,194.146	—
	(10, 1, 3, 2)	5,085.598	2.090
SDSC-Par96	MC1x1	3,556.935	—
	(4, 29, 5, 0)	3,428.283	3.617
LANL-CM5	MC1x1	214,790.442	—
	(12, 42, 41, 15)	209,539.330	2.445

Fig. 16. Performance of best tie-breaking parameters from Figure 8 when scheduled with EASY

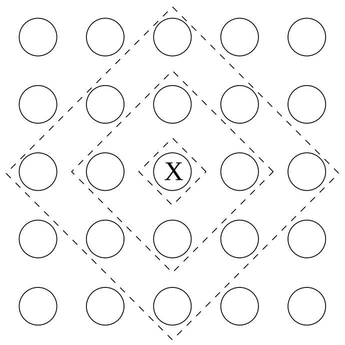

Fig. 17. Gen-Alg's shells centered on "c".

of L_1 distances between processors and they show that Gen-Alg is a $(2 - 2/k)$-approximation for this problem, meaning it always finds an allocation within a factor of $(2 - 2/k)$ of the optimal, where k is the size of the job.

The last allocation algorithm in the same family as MC1x1 and Gen-Alg is MM, proposed by Bender et al. [20]. This algorithm tries more candidate centers than the other algorithms; MM's candidate centers are all processors that share each coordinate with some free processor (possibly a distinct free processor for each coordinate). Specifically, in a 2D mesh, a center is any processor that both shares its x-coordinate with a free processor and its y-coordinate with a (possibly different) free processor. Once the centers are selected, MM evaluates them in the same way as Gen-Alg; the candidate allocation around a center is the processors that are closest in terms of L_1 distance and the score of a candidate allocation is its sum of pairwise L_1 distances. Bender et al. [20] show that MM is a $2 - 1/(2d)$-approximation algorithm in a d-dimensional mesh, meaning that it is a $7/4$-approximation in 2D meshes and an $11/6$-approximation in 3D meshes.

Another family of allocation algorithms is based on a linear order of the processors. The first of these is Paging, proposed by Lo et al. [19], which uses a linear order to maintain a sorted list of free processors.[2] When an allocation is needed, Paging simply assigns it the first processors from this list. Lo et al. [19] proposed several linear orders to use with the Paging algorithm, including the row-major, a "snake" curve that traverses alternate rows in opposite directions, and "shuffled" versions of these.

An ordering-based strategy was independently proposed by Leung et al. [17], who introduced a couple of refinements. First, they proposed getting the ordering from space-filling curves such as the recursively-generated Hilbert curve [28]. The other improvement Leung et al. [17] proposed was to use more sophisticated strategies adapted from bin packing to select processors from the list. For example, Best Fit allocates the job into the smallest interval of contiguous free processors that is large enough to fit the entire job. Later work expanded on this by considering another space-filling curve [29] and the treatment of non-square meshes [30].

Some allocation algorithms are neither center-based nor curve-based. ANCA [21] which splits jobs into multiple parts when contiguous allocation is not possible and then gives each part a contiguous allocation. Somewhat similar is Multiple Buddy [19], [30], which uses a buddy system to keep track of free processors and uses multiple smaller contiguous blocks to allocate jobs whose size does not match one of the free blocks. Bender et al. [20] give an arbitrarily-good approximation algorithm (PTAS) for finding allocations that minimize the sum of pairwise distances.

To compare our work against these algorithms, we used the LLNL-T3D trace. This trace was selected because it can be run on a 16×16 mesh; meshes whose dimensions are not powers of two complicate using space-filling curves [29]. Figure 18 compares MC1x1 with and without tie breaking to Gen-Alg, MM, and the curve-based strategy using Best Fit. Although the results are not shown, we also ran the curve-based strategies using a free list; as in previous work [17], Best Fit gave better results in all cases. MC1x1 does well even without our tie-breaking strategy, but it gives the best results of any of algorithm when the tie-breaking strategy is added.

V. Discussion

Our tie-breaking strategy improves system performance by helping MC1x1 keep the machine in a good state for future allocations without compromising the quality of the current

[2]We actually present a special case of Paging, which groups processors into blocks that are stored in the sorted free list and assigned to jobs together. We use block size 1 since we require successful allocation if enough processors are free and cannot guarantee job sizes are multiples of any larger number.

Allocator	Ave. L_1 Dist.
MC1x1 w/ tie-breaking (3, 9, 3, 2)	5,224.970
Hilbert curve with Best Fit	5,232.127
MC1x1 without tie-breaking	5,309.586
MM	5,382.295
Gen-Alg	5,386.368
Snake curve with Best Fit	5,683.268

Fig. 18. Comparison of tie-breaking with other strategies for LLNL-T3D trace with FCFS scheduling.

allocation. Though the tie-breaking strategy gives only small overall improvement, viewing the results grouped by job size shows that it significantly improves the quality of allocations for most job sizes. Although some scheduling research (eg. [31]) examines the effect of algorithms on different job categories, we believe ours is the first work to do this for processor allocation.

We see several ways to extend this work. We found a set of weight factors that improves performance for a wide variety of traces, but different sites would likely want to adjust the parameters to optimize them for specific machines and types of jobs. It would be desirable to have a way to do this without searching the parameter space or, even better, to develop a self-tuning version of our tie-breaking strategy. One could also apply the tie-breaking strategy to different machine topologies; it would be straightforward to consider 3D meshes or toroidal wraps. More ambitious would be to consider allocations that do not receive the best score but have a much better tie-breaking score. This harms the current job, but leaves the machine in a much better state and could benefit the system overall.

ACKNOWLEDGMENTS

C.R. Johnson was partially supported by contract 763836 from Sandia National Laboratories. D.P. Bunde was supported in part by contract 763836 from Sandia and Howard Hughes Medical Institute grant 52005130. Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Adminstration under Contract DE-AC04-94AL85000. We also thank contributors to the Parallel Workloads Archive. Specifically, we thank Lars Malinowsky for the KTH-SP2 trace, Moe Jette for the LLNL-T3D trace, Reagan Moore and Allen Downey for the SDSC traces, and Curt Canada for the LANL-CM5 trace.

REFERENCES

[1] Top500.org, "XP/S35," http://www.top500.org/system/747.
[2] Top500.org, "XP/S140," http://www.top500.org/system/1037.
[3] Top500.org, "ASCI Red," http://www.top500.org/system/2758.
[4] Top500.org, "ASCI Red," http://www.top500.org/system/3059.
[5] Top500.org, "CPlant cluster," http://www.top500.org/system/3578.
[6] Top500.org, "ASCI Red," http://www.top500.org/system/4048.
[7] Top500.org, "ASCI Red," http://www.top500.org/system/4428.
[8] Top500.org, "CPlant/Siberia cluster," http://www.top500.org/system/4443.
[9] Top500.org, "CPlant/Siberia cluster," http://www.top500.org/system/5323.
[10] Top500.org, "CPlant/Ross cluster," http://www.top500.org/system/5324.
[11] Top500.org, "CPlant/Ross cluster," http://www.top500.org/system/5598.
[12] Top500.org, "CPlant/Ross cluster," http://www.top500.org/system/6157.
[13] Top500.org, "Red Storm Cray XT3, 2.0 GHz," http://www.top500.org/system/7596.
[14] Top500.org, "Red Storm Cray XT3, 2.0 GHz," http://www.top500.org/system/7653.
[15] Top500.org, "Red Sky," http://www.top500.org/system/10188.
[16] Top500.org, "Sandia/Cray Red Storm," http://www.top500.org/system/10364.
[17] 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.
[18] P. Krueger, T.-H. Lai, and V. Dixit-Radiya, "Job scheduling is more important than processor allocation for hypercube computers," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 5, pp. 488–497, 1994.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] D. Feitelson, "The parallel workloads archive," http://www.cs.huji.ac.il/labs/parallel/workload/index.html.
[25] D. Lifka, "The ANL/IBM SP scheduling system," in *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing*, ser. LNCS, no. 949, 1995, pp. 295–303.
[26] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel and Distributed Syst.*, vol. 12, no. 6, pp. 529–543, 2001.
[27] 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.
[28] D. Hilbert, "Über die stetige abbildung einer linie auf ein flachenstück," *Math. Ann.*, vol. 38, pp. 459–460, 1891.
[29] D. Bunde, V. Leung, and J. Mache, "Communication patterns and allocation strategies," in *Proc. 3rd Intern. Workshop Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2004.
[30] P. Walker, D. Bunde, and V. Leung, "Faster high-quality processor allocation," in *Proc. 11th LCI Intern. Conf. High-Performance Clustered Computing*, 2010.
[31] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Proc. 6th Workshop Job Scheduling Strategies for Parallel Processing*, ser. LNCS, no. 1911, 2000, pp. 1–17.