A Memory-Efficient Data Redistribution Algorithm*

Stephen F. Siegel[1] and Andrew R. Siegel[2]

[1] Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA, siegel@cis.udel.edu
[2] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA, siegela@mcs.anl.gov

Abstract. Many memory-bound distributed applications require frequent redistribution of data. Pinar and Hendrickson investigated two families of memory-limited redistribution algorithms. The first family has many advantages, but fails on certain inputs, and, if not implemented carefully, may lead to an explosion in the number of local data copies. The second family eliminates the possibility of failure at the expense of considerable additional overhead. We carefully analyze these algorithms and develop a modified method that potentially combines advantages of each. The resulting algorithm has been implemented in MADRE and experiments reveal its performance to be superior to that of other MADRE algorithms in most cases.
Key words: MADRE, redistribution, memory-limited, distributed, MPI

1 Introduction

Many distributed-memory, message-passing, parallel programs periodically redistribute data among the program's processes. Solutions to such problems have been studied extensively in a variety of contexts. For example, the well known $M \times N$ parallel data migration problem (e.g. [1, 2]) involves efficiently moving data between typically disjoint processor sets with different decomposition schemes. A related problem that has received considerable attention is the so-called block-cyclic array redistribution (e.g. [3, 7]), which seeks efficient algorithms to map data from a $cyclic(x)$ to a $cyclic(Kx)$ decomposition.

The class of algorithms discussed in this paper address a related problem but where the driving requirement is the transparent and efficient use of memory at the application level. Furthermore, unlike much of the work on block-cyclic algorithms, aspects of the physical network (processor topology, routing algorithms, etc.) are intentionally abstracted away. A large class of scientific applications are memory-bound, and cannot guarantee the availability of sufficient memory for a more straightforward approach to data migration (e.g. of invoking MPI_ALLTOALL). For these applications, a limited-memory redistribution algorithm is required.

* This research was supported by the National Science Foundation under Grants CCF-0733035 and CCF-0540948.

Pinar and Hendrickson [4] introduced a formal "phase"-based description of the limited-memory redistribution problem, showed the problem of finding a minimal-phase solution to be NP-hard, and analyzed a class of "basic" algorithms approximating optimal solutions. They also pointed out that the basic algorithms would not complete on some inputs. For this reason, and to further reduce the number of phases, a second class of "parking" algorithms was introduced. These complete on any valid input, but add significant overhead and inter-process synchronization, and in many of the experiments of [4] performed worse than the basic ones.

The Memory-Aware Data Redistribution Engine (MADRE) has been used to study these and other solutions to the limited-memory redistribution problem [5,6]. Experiments with MADRE revealed that in some cases, the time spent on local data copies could dominate instances of the Pinar-Hendrickson algorithms, an issue that was abstracted away in [4].

In this paper, we show that the basic algorithms can be modified to overcome both issues described above: the modified algorithm can complete on any input and the number of local data copies can be significantly reduced. We give a precise description of the original algorithm in Section 2, and prove that the algorithm is always deadlock-free, though it can "livelock" for certain inputs. In Section 3, we find a condition on the input that suffices for termination. Using this, we show how the basic algorithm can be be modified to terminate successfully on any input. The problem of local data copies is dealt with in Section 4. The resulting algorithm has been implemented in MADRE, and experiments comparing its performance are given in Section 5. The results show the new algorithm to be superior to all other MADRE algorithms in most cases.

2 The Basic Pinar-Hendrickson Redistribution Algorithm

We are given a distributed memory parallel program of n processes. Each process maintains in its local memory a contiguous, fixed-sized array of data blocks. At any time, some number of these blocks are *free*, i.e., do not contain useful data. The free blocks may be used as temporary space in the redistribution.

The input to a redistribution algorithm consists of the following:

1. free_p $(0 \le p < n)$: the initial number of free spaces on each process, and
2. $\mathsf{out}_p[i]$ $(0 \le p, i < n)$: the number of blocks process p is to send to process i.

We assume all of these values are non-negative and $\mathsf{out}_p[p] = 0$ for all p. (In practice, the input must specify the destination rank and index for each non-free block, but in this presentation we elide such detail.) The goal is to transfer the blocks to their new locations in a sequence of phases without ever exceeding the free space available on any process.

Let $\mathsf{in}_p[i] = \mathsf{out}_i[p]$ and

$$\mathsf{free}'_p = \mathsf{free}_p - \sum_{i=0}^{n-1} \mathsf{in}_p[i] + \sum_{i=0}^{n-1} \mathsf{out}_p[i]. \tag{1}$$

symbol	meaning
free	number of free spaces on this proc (given)
out[i]	number of blocks remaining to send to proc i (given)
in[i]	number of blocks remaining to receive from proc i (initially 0)
send[i]	number of blocks to send to i in current phase (initially 0)
recv[i]	number of blocks to receive from i in current phase (initially 0)

```
1  procedure main is
2      Alltoall(out, 1, INT, in, 1, INT);
3      nPhase ← 0;
4      while ∑_i(out[i] + in[i]) > 0 do
5          computeMoves();
6          executePhase();
7          nPhase ← nPhase + 1;
8      move blocks to final positions;
9  procedure computeMoves is
10     r ← free;
11     for i ← 0 to n − 1 do
12         recv[i] ← min{r, in[i]};
13         r ← r − recv[i];
14     foreach {i | in[i] > 0} do
15         post send of recv[i] to proc i;
16     foreach {i | out[i] > 0} do
17         post recv into send[i] for proc i;
18     wait for all requests to complete;
19 procedure executePhase is
20     sort blocks to create contiguous
       buffers;
21     foreach {i | recv[i] > 0} do
22         post recv for recv[i] blocks
           from proc i;
23         in[i] ← in[i] − recv[i];
24     foreach {i | send[i] > 0} do
25         post send of send[i] blocks to
           proc i;
26         out[i] ← out[i] − send[i];
27     free ← free + ∑_i(send[i] − recv[i]);
28     wait for all requests to complete;
```

Fig. 1. Basic Pinar-Hendrickson Algorithm using "first-fit" heuristic

Then free'_p will be the amount of free space on process p after redistribution. The input is *feasible* if $\mathsf{free}'_p \geq 0$ for all p. Ideally, a redistribution algorithm should work correctly for any feasible input. We will see, however, that the Basic Algorithm described below fails for certain feasible inputs.

The *Basic Algorithm* appears in Figure 1. The input provides the initial values for the variables free and out[i] on each process p. Process p proceeds in a sequence of phases until it has no more blocks to send or receive. In each phase, p receives into its free space while sending from some of its non-free spaces. In the first part of a phase, p allocates its current free spaces to the processes that have data to send to it. Different heuristics can be used to determine this allocation; for simplicity we use the *first-fit* heuristic of [4], though the results we describe here are valid for any heuristic. After deciding on the allocation, p sends a message consisting of a single integer to all of its source processes informing them of how many blocks they should send to p in the current phase. At the same time, p receives similar messages from its targets. Once this communication has completed, the phase is *executed* by sending and receiving the agreed quantities of data to and from each process, and updating local data structures.

We first establish the following:

Theorem 1 *The Basic Algorithm is deadlock-free on any input.*

Proof. Let E be a *maximal* execution. This means that either (1) E is infinite, or (2) E is finite and every process has either terminated or become permanently blocked waiting for some request to complete.

Say $0 \leq p < n$. If process p becomes permanently blocked in E, let M_p be the final value of nPhase. Otherwise let $M_p = \infty$. For $0 \leq j \leq M_p$ and $0 \leq i < n$, let $\mathsf{out}_{p,j}[i]$ denote the value of $\mathsf{out}[i]$ on process p when the expression on line 4 is evaluated and $\mathsf{nPhase} = j$, or 0 if p terminates before $\mathsf{nPhase} = j$. Define $\mathsf{in}_{p,j}[i]$, $\mathsf{send}_{p,j}[i]$, and $\mathsf{recv}_{p,j}[i]$ similarly. We show for all $j \geq 0$, the following all hold:

1. $M_p \geq j$ $(0 \leq p < n)$,
2. $\mathsf{in}_{p,j}[i] = \mathsf{out}_{i,j}[p]$ $(0 \leq i, p < n)$,
3. $\mathsf{send}_{p,j}[i] = \mathsf{recv}_{i,j}[p]$ $(0 \leq i, p < n)$, and
4. if $j > 0$, any send request generated in phase $j-1$ is matched with a receive request generated in phase $j-1$.

This implies, in particular, $M_p = \infty$ for all p, i.e., E does not deadlock.

The proof is by induction on j. The case $j = 0$ is clear: the all-to-all operation guarantees the in and out variables correspond in the desired way, and the send and recv arrays are uniformly 0. Suppose $j > 0$ and the hypothesis holds for values less than j; we will show it holds for j.

Suppose process p enters phase $j-1$ and posts a send to process i in *computeMoves*. Then $\mathsf{out}_{p,j-1}[i] = \mathsf{in}_{i,j-1}[p] > 0$, by the induction hypothesis. So process i must enter phase $j-1$ and post a receive to p in *computeMoves*. Similarly, if p enters and posts a receive for process i then i must enter and post a send to p.

Since all requests from previous phases have been matched, each send in phase $j-1$ must be paired with the corresponding receive. All of these paired requests must eventually complete, so all processes which have not terminated will return from *computeMoves*. Moreover, since the message sent by process p to process i is exactly $\mathsf{recv}_{p,j}[i]$, and this message is received into $\mathsf{send}_i[p]$, we have $\mathsf{recv}_{p,j}[i] = \mathsf{send}_{i,j}[p]$.

In *executePhase*, the same argument shows that the sends and receives match up in the expected way. Moreover, the assignments on lines 23 and 26 guarantee

$$\mathsf{in}_{p,j}[i] = \mathsf{in}_{p,j-1}[i] - \mathsf{recv}_{p,j}[i] = \mathsf{out}_{i,j-1}[p] - \mathsf{send}_{i,j}[p] = \mathsf{out}_{i,j}[p],$$

which completes the inductive step. □

3 Termination and Modified Basic Algorithm

Though the Basic Algorithm cannot deadlock, it is possible for it to "livelock": some processes may loop forever without making progress. This may happen even if the input is feasible. An obvious case is where no process has any free space. Another example, considered in [4], is where two processes with no free space attempt to exchange data while there is free space on a third (Figure 2(a)).

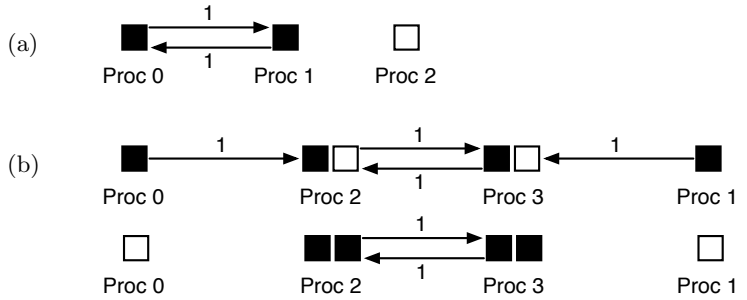


Fig. 2. Scenarios in which Basic Algorithm fails

In these cases the algorithm will always fail, regardless of the heuristic used to assign free spaces to incoming blocks.

What is less obvious is that there are situations where the algorithm can succeed if certain choices are made but will fail for other choices. Consider the example of Figure 2(b). If, in the first phase, process 2 chooses to allocate its one free space to process 0, and process 3 chooses to allocate its one free space to process 1, then the algorithm will not progress after the phase completes. (These are in fact the choices made by the first-fit heuristic.) If, on the other hand, process 2 first selects 3, and 3 selects 2, the algorithm completes normally.

To deal with these problems, and to reduce the total number of phases, Pinar and Hendrickson introduce the concept of *parking*. The idea is to move data blocks to temporary locations on processes that have extra free space, and later move these blocks to their final destinations. They describe a family of such algorithms and show that these will solve any feasible redistribution problem as long as there is at least one process with at least one free space. But as mentioned above, the parking algorithms also have many disadvantages.

The following establishes a sufficient condition for the successful completion of the Basic Algorithm:

Theorem 2 *Suppose the input to the Basic Algorithm satisfies* $\mathsf{free}'_p \geq 1$ *for all* p*. Then all processes will terminate normally.*

Proof. For any process p, whenever control enters the **while** loop,

$$\mathsf{free} - \sum_{i=0}^{n-1} \mathsf{in}[i] + \sum_{i=0}^{n-1} \mathsf{out}[i] = \mathsf{free}'_p \geq 1. \tag{2}$$

This is true when control first enters the loop by the definition of free'_p. It remains true on subsequent iterations because the value of the expression on the left-hand side of (2) is preserved by *computeMoves* and *executePhase*.

By Theorem 1, the algorithm does not deadlock. Suppose it livelocks. Then eventually a point is reached after which the values of in, out, and free never change. For $0 \leq p < n$, let in_p denote the final value of $\sum_i \mathsf{in}[i]$ on process

p. Define out_p similarly. Let $free_p$ be the final value of free on process p. Let $S = \{p \mid in_p > 0\}$. After the stable point has been reached, any process with incoming data must have no free space, i.e., $p \in S \Rightarrow free_p = 0$. By (2),

$$-in_p + out_p \geq 1 \qquad \text{for all } p \in S. \tag{3}$$

Summing (3) over all $p \in S$ yields

$$-\sum_{p \in S} in_p + \sum_{p \in S} out_p \geq |S|. \tag{4}$$

On the other hand, the total amount of incoming data must equal the total amount of outgoing data:

$$\sum_{p=0}^{n-1} in_p = \sum_{p=0}^{n-1} out_p$$

Hence

$$\sum_{p \in S} in_p - \sum_{p \in S} out_p = \sum_{p=0}^{n-1} in_p - \sum_{p=0}^{n-1} out_p + \sum_{p \notin S} out_p = \sum_{p \notin S} out_p \geq 0 \tag{5}$$

Adding (4) and (5) yields $0 \geq |S|$, i.e., $S = \varnothing$. This means $in_p = 0$ for all p, and hence $out_p = 0$ for all p, so all processes terminate normally, a contradiction. □

Theorem 2 suggests a way to modify the Basic Algorithm so that it will complete normally on any feasible input: simply allocate one extra block on every process. The existence of this block is not revealed to the user, so that a feasible map from the user's point of view will always result in at least one free block on every process post-redistribution. The extra blocks will initially be free. Theorem 2 guarantees the algorithm will complete and there will be at least one free block on each process. After all blocks are moved to their final positions, the extra block will again be free. We call this algorithm the Modified Basic Algorithm (MBA). For brevity, we will elide the details concerning the extra block, and just assume the input satisfies the hypothesis of Theorem 2.

4 Local Copy Efficient Algorithm

To this point we have ignored the question of local data movement. In [5], it was seen that in many examples, the time to perform local data copies could dominate the inter-process communication time. We now show how the MBA can be altered to deal efficiently with local data copying.

In each phase, the MBA sorts the blocks so that the data to be sent to a given process will be contiguous and can therefore be sent using a single MPI operation. The sort also makes the free space contiguous to facilitate the receives. Our implementation of MBA accomplishes this by sorting the blocks by increasing destination rank and placing all free blocks at the end. While this is a simple

```
1  procedure main is
2  |  Alltoall(out, 1, INT, in, 1, INT); nPhase ← 0;
3  |  while ∑_i(out[i] + in[i]) > 0 do computePhase();
4  |  sort blocks in phase order;
5  |  for i ← 0 to nPhase − 1 do executePhase(sched[i]);
6  |  move blocks to final positions;
7  procedure computePhase is
8  |  R ← ∅; S ← ∅; r ← free;
9  |  for i ← 0 to n − 1 do
10 |  |  recv[i] ← min{r, in[i]}; r ← r − recv[i];
11 |  |  if recv[i] > 0 then R ← R ∪ {(i, recv[i])};
12 |  foreach {i | in[i] > 0} do
13 |  |  post send of recv[i] to proc i; in[i] ← in[i] − recv[i]; free ← free − recv[i];
14 |  foreach {i | out[i] > 0} do post recv into send[i] for proc i;
15 |  wait for all requests to complete;
16 |  foreach {i | out[i] > 0 ∧ send[i] > 0} do
17 |  |  S ← S ∪ {(i, send[i])}; free ← free + send[i]; out[i] ← out[i] − send[i];
18 |  if R ≠ ∅ ∨ S ≠ ∅ then { sched[nPhase] ← (R, S); nPhase ← nPhase + 1; }
19 procedure executePhase(R, S) is
20 |  foreach (i, q) ∈ R do post recv for q blocks from proc i;
21 |  foreach (i, q) ∈ S do post send of q blocks to proc i;
22 |  wait for all requests to complete;
```

Fig. 3. Local Copy Efficient Algorithm. A schedule sched is computed completely before it is executed. Each entry in sched is an ordered pair (R, S). R encodes the number of blocks to receive from each process in the phase; S the number to send.

solution, it can lead to an explosion in local data copies. An an example, in one of the maps considered in [5], all processes have the same number of blocks m, there are no free spaces, and the solution requires m phases. Each phase involves sending and receiving one block, and the local sort turns out to be the worst possible case, involving approximately m calls to `memcpy`. Hence the total number of copies performed by a process is approximately m^2.

We now describe the Local Copy Efficient (LCE) Algorithm (Figure 3). This algorithm performs at most $O(m)$ local data copies on each process.

The first change to the earlier algorithm involves separating the process of schedule *computation* from that of schedule *execution*. (The same idea was employed in [5] to a very different algorithm, the "cyclic scheduler.") In the new algorithm, *computePhase* is called repeatedly and the result accumulated in a *schedule*. After the entire schedule has been computed, *executePhase* is called repeatedly to carry out the actual sending and receiving of blocks.

A schedule is an array of *phase structures*. Each phase structure specifies the number of blocks to send to each process and the number to receive from each process, in that phase. Careful consideration must be given to the data

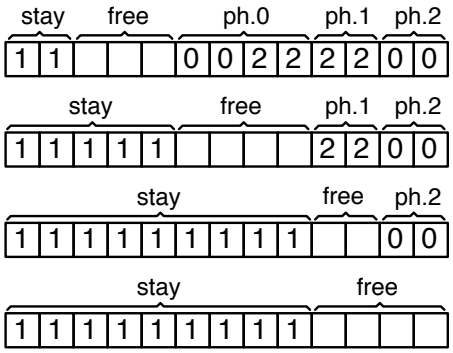


Fig. 4. Data layout to avoid local movement during schedule execution. Blocks are initially sorted in *phase order*. During phase i, blocks are received into the free region and sent from the region marked ph.i. No local data movement occurs between phases.

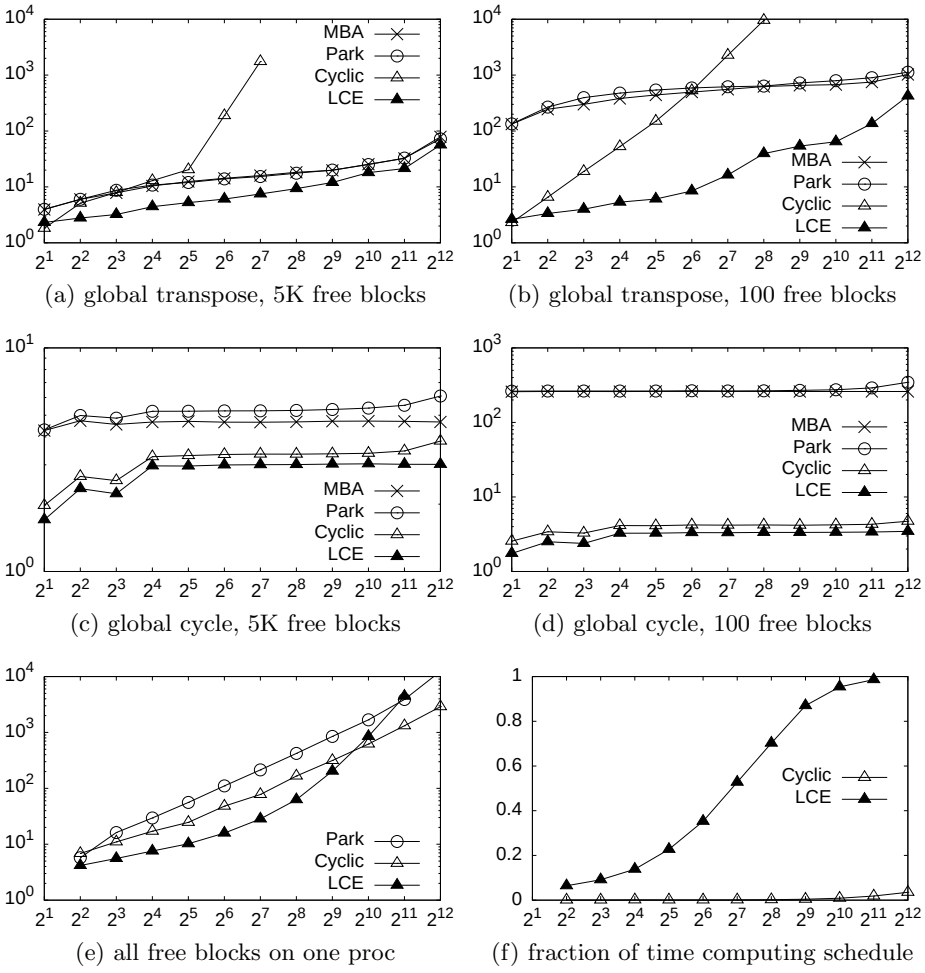
structures used in a schedule. We have already seen it is possible for the number of phases to approach m, the number of blocks on one process. If each phase requires storing an array of length n then the memory consumed by a schedule can approach nm, an unacceptable level. In contrast, the worst-case memory requirements of the Basic Algorithm, and all algorithms discussed in [5, 6], are linear combinations of n, m, and the size of one block.

We therefore use a sparse format to represent a phase structure. It consists of a set R of pairs (i, in_i) for which $in_i > 0$, for the incoming data, and a set S of pairs (j, out_j) for which $out_j > 0$, for the outgoing data. In the worst case, the total number of blocks sent and the total number received are both m. Since each pair in the schedule represents the send or receive of at least one block, the total number of pairs is at most $2m$. Furthermore, a phase in which there is no incoming or outgoing data is simply not recorded, so the number of phase structures on a process is at most $2m$. Hence the memory consumed by the schedule is $O(m)$.

Once the schedule has been computed, the blocks are sorted in *phase order*: all blocks staying on the process appear first, followed by all free blocks, then all blocks departing in phase 0, followed by all blocks departing in phase 1, and so on. For each i, the outgoing blocks for phase i are further arranged by increasing destination rank, to create contiguous send buffers. (See Figure 4.)

Once the blocks are arranged in phase order, no local data movement is needed until the entire schedule has been executed. This is because incoming blocks are received into the free region while outgoing blocks are sent from the region immediately following the free region. Once a phase completes, the new free region is contiguous, consisting of perhaps part of the old free region (if not all free spaces were used to receive incoming data) and all of the old send region.

After schedule execution, a final local redistribution is required to move all blocks to their final positions. Hence the entire algorithm requires two local redistributions: one at the beginning to sort the blocks in phase order, and



(a) global transpose, 5K free blocks

(b) global transpose, 100 free blocks

(c) global cycle, 5K free blocks

(d) global cycle, 100 free blocks

(e) all free blocks on one proc

(f) fraction of time computing schedule

Fig. 5. Time to redistribute data. Each process maintains 25,000 blocks of 16,000 bytes each. In (a)–(e), x-axis is number of processes, y-axis is the time in seconds to complete a data redistribution.

one at the end. An efficient local redistribution algorithm is described in [6]. That algorithm performs precisely the minimum number of copies possible; in particular, each block is moved at most 2 times. So the LCE algorithm performs at most $O(m)$ local copies.

5 Experiments and Conclusions

Experiments were run on Argonne's *Full Disclosure*, a SiCortex 5832 with 972 nodes, each with 6 cores and 4 GB RAM (512 MB of which is reserved for fabric use). Each MPI process was mapped to one core. In all experiments, each process managed 25K blocks of 16 KB each, for a total of 400 MB per process. Each

experiment was executed using $n = 2^k$ processes, where k was scaled to 12 or until execution time exceeded 10^4 seconds. This was repeated for each of four redistribution algorithms: (1) MBA, (2) Park: a Pinar-Hendrickson "parking" algorithm, (3) Cyclic: the "cyclic scheduler" algorithm, and (4) LCE. (See [5,6] for a detailed description of (2) and (3).) Timing results appear in Figure 5.

In experiment (a), each process has 5K free blocks and the map sends block j of process i to position $(mi + j)/n$ of process $(mi + j)\%n$, where $m = 20$K. (The map is essentially a global transpose of the data matrix.) Experiment (b) is similar but with only 100 free blocks per process. As explained in [5], the cyclic algorithm blows up on this map. In both cases, LCE performs better than the other algorithms for all $n > 2$, though it is not clear if this trend would continue for $n > 2^{12}$. The decreasing difference between the performance of LCE and MBA/Park in (b) appears to be due to the fact that the proportion of time devoted to schedule computation increases with n; e.g, schedule computation consumes 202 of the 422 seconds for $n = 2^{12}$. We have not shown data on the memory consumed by the algorithms, but the numbers are generally small and the differences not great, e.g., 1060 KB per process for LCE vs. 845 KB for MBA at $n = 2^{12}$ in (b).

In experiments (c) and (d) all data from process i is sent to process $(i+1)\%n$. For (c), each process has 5K free blocks; in (d), 100 free blocks. Again LCE proves faster than the other algorithms, with Cyclic not far behind. The inefficiency of MBA/Park in these experiments is due primarily to the large number of local data copies performed between phases, which both Cyclic and LCE avoid.

In experiment (e), $n - 1$ processes have no free space; the last process has all 25K blocks free. Each of the $n - 1$ "full" processes divides its data into $n - 2$ approximately equally sized slices and sends one slice to each of the other full processes. MBA cannot complete within 10^4 seconds (even for $n = 2$), due to the enormous number of local data copies. Park, in contrast, takes advantage of the free space on the last process to greatly reduce the number of phases. Cyclic makes no use at all of the free process, but has the advantage that its schedule is computed in a single phase (see [5]). The experiment reveals that LCE performs better through $n = 2^9$, but beyond this point LCE is surpassed by Cyclic, and at $n = 2^{11}$ by Park. The blowup of LCE time again appears to be due to the time required to compute the schedule (f). For any full process, each phase involves sending and receiving one block, so there are a total of 25K phases. Scheduling a phase requires essentially an all-to-all communication among the $n - 1$ full processes. In contrast, to execute a phase each process communicates with only two other processes.

Conclusion. The LCE algorithm performs better than the other MADRE algorithms in most, but not all, cases. Our future work will look for ways to reduce the communication cost of schedule computation when free space is very limited. Other planned improvements include developing a local copy efficient version of the parking algorithm and a multi-threaded version of LCE for use in hybrid MPI/threaded programs.

References

1. F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson, and K. B. Damevski. Data redistribution and remote method invocation in parallel component architectures. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 40.2, Washington, DC, USA, 2005. IEEE Computer Society.
2. F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley. An approach to parallel MxN communication. In *Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium*, Santa Fe, NM, October 2003.
3. N. Park, V. K. Prasanna, and C. Raghavendra. Efficient algorithms for block-cyclic array redistribution between processor sets. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.
4. A. Pinar and B. Hendrickson. Interprocessor communication with limited memory. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):606–616, July 2004.
5. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings*, volume 5205 of *LNCS*, pages 218–226. Springer, 2008.
6. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. Technical Report UDEL-CIS 2009/335, Department of Computer and Information Sciences, University of Delaware, 2009. To appear in *The International Journal of High Performance Computing Applications*.
7. R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *In Proc. of Scalable High Performance Computing Conference*, pages 309–316. IEEE, 1994.