MADRE: The Memory-Aware Data Redistribution Engine*

Stephen F. Siegel Andrew R. Siegel

April 22, 2009

Abstract

We report on the development of a new computational framework for efficiently carrying out parallel data redistribution in a limited memory environment. This new library, MADRE (The Memory-Aware Data Redistribution Engine), is an open-source, C/MPI-based toolkit designed for quick and easy integration into application codes that have demanding data migration needs. At the same time, MADRE exposes a lower-level API that greatly facilitates the development and incorporation of new algorithms into the MADRE framework, thus serving as a potential organizing entity for continued research in this area. Finally, we develop, describe, and test in detail several new parallel redistribution algorithms that are incorporated into the MADRE distribution.

1 Introduction

A large class of massively parallel scientific applications are memory-bound. To achieve their scientific aims, considerable care must be taken to reduce their memory footprint via careful programming techniques, e.g. avoiding unnecessary data copies of main data structures, not storing global meta-data on a single processor, and so forth. The need for efficient many-to-many parallel data movement arises in a wide variety of such scientific applications (e.g., [8]). Adaptive algorithms, for example, require extensive load balancing to ensure an optimal distribution of mesh elements across processors (both in terms of equal balance and spatial locality) [2]. Time-dependent domain-decomposed particle codes are another good example, with frequent re-balancing of the main data structures as particles cross processor boundaries [6].

Typically, such load balancing operations consist of two parts: (1) computing the *map* that specifies the new destination for each block of data and (2) efficiently carrying out the movement of data blocks to their new location without exceeding the memory available to the application on any process. There has been much research into the first problem, resulting, for example, in numerous efficient algorithms for computing space-filling curves [1]. The second problem, though,

*Technical Report 2009/335, Department of Computer & Information Sciences, University of Delaware, April 2009. This research was supported by the National Science Foundation under Grants CCF-0733035 and CCF-0540948 and used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

has received little attention, especially in the critical case where memory resources are severely limited and memory use must be completely transparent to the application developer.

One of the few explorations into memory-efficient solutions to the data redistribution problem was undertaken by A. Pinar and B. Hendrickson [7]. They formulated a "phase"-based class of solution algorithms, showed that the problem of finding a minimal-phase solution is NP-hard, and studied a family of algorithms for approximating minimal solutions.

There remain a number of open issues, though. For example, one can envision many other solutions to the redistribution problem that are not based on the concept of migration phases. What are the basic properties of these algorithms and how will they perform relative to the phase-based methods? What is the impact of local data copies on overall algorithmic performance? What algorithms perform best for different areas of parameter space, including memory footprint, available scratch space, data block size, etc? Further theoretical work and experimentation is necessary to address these questions.

Our approach to addressing these questions includes several related goals: (1) to package the new algorithms in a way that is readily accessible to application-level developers with little interest in the underlying theory and (2) to provide a clear, open source software architecture together with the rudimentary algorithmic components to encourage the contribution of new methods by algorithm developers throughout the community. These requirements underpin the current design philosophy of MADRE (Memory-Aware Data Redistribution Engine) [9], an open source, C/MPI-based toolkit for the solution of the limited-memory data redistribution problem.

In this paper we describe the theory behind the multi-tiered MADRE API, focusing first on the fundamental properties of the local data redistribution problem, then extending this work to the parallel case. Two parallel algorithms are presented, their basic properties are studied in detail, and their performance is compared on a range of synthetic problems.

2 Local Redistribution

We begin with a precise analysis and description of the limited-memory redistribution problem on a single process. We do this both for clarity of exposition and, as we show, because efficient global (parallel) redistribution algorithms are based in large part on efficient solutions to the local problem. Providing access to a highly tuned local block solver is thus a key component of MADRE extensibility. An excerpt of the MADRE interface demonstrating these layers is shown in Figure 1.

The goal of the local redistribution problem is to minimize the number of data copies required to redistribute the blocks according to a given map. The problem is similar to an in-place redistribution of an array based on a given permutation. The maps we are considering, however, are more general than permutations. This is because, in our context, some blocks may be "free," i.e., they do not currently contain useful data. A redistribution map may indicate that a data block be moved to a free space, in which case there is no need to first copy the data in the free space to somewhere else.

The redistribution map is therefore not necessarily a permutation, but a more general sort of function we call a *transmutation*. Much of the standard theory on permutations generalizes to transmutations. The basic results on transmutations presented below have been known for some time in the field of *symmetric inverse semigroups* [4,5], but in Section 2.1 we give our own concise, self-contained presentation of the exact results needed for the local algorithm. The algorithm itself is described and analyzed in Section 2.2. Proofs of all results can be found in Appendix A.

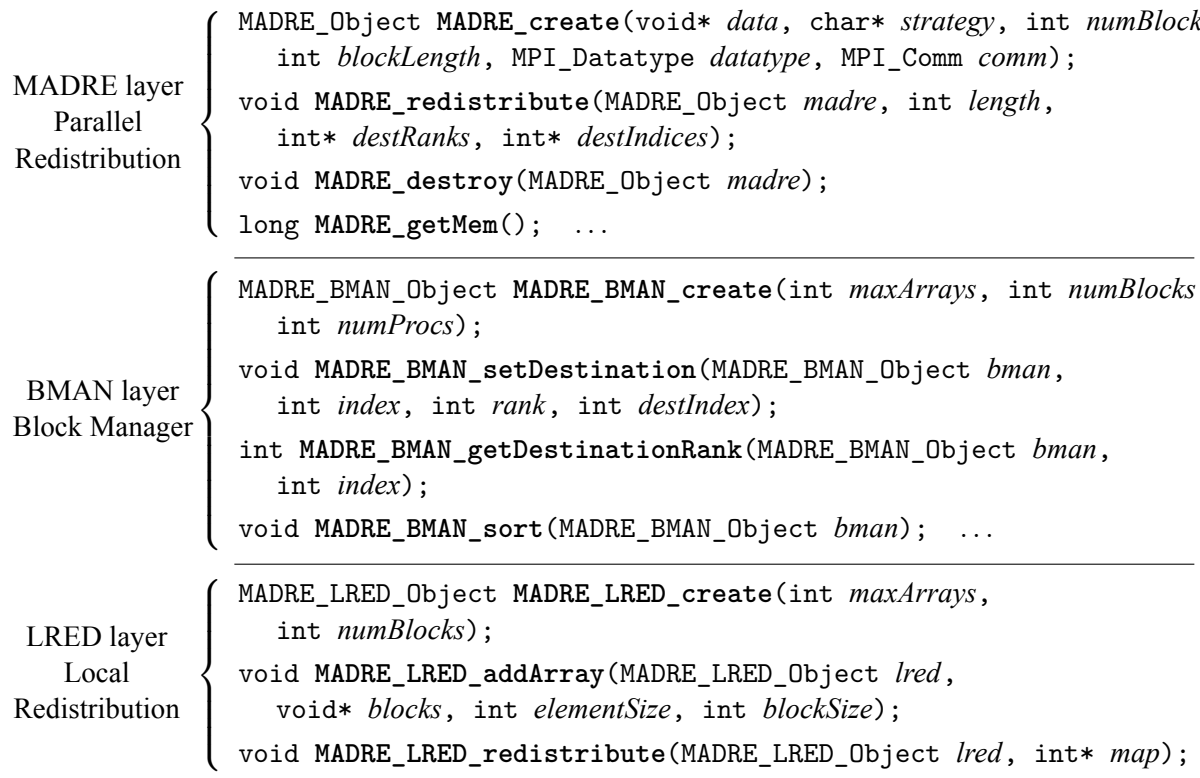


Figure 1: MADRE software layers

2.1 Transmutations

Let X be a finite set. Intuitively, we are interested in injective functions π from X to X which are not necessarily defined at every point. The following definition just makes this notion precise:

Definition 2.1. A *transmutation* of X is an ordered pair $\pi = (S, f)$ where $S \subseteq X$ and f is an injective function from S to X. The set of all transmutations of X is denoted $\mathsf{Trans}(X)$.

If we think of π in the intuitive way as a function on X, then S is just the set of points on which π is defined.

Define a product on $\mathsf{Trans}(X)$ as follows: given transmutations $\pi = (S, f)$ and $\sigma = (T, g)$, let $\pi\sigma = (U, h)$, where $U = \{x \in T \mid g(x) \in S\}$ and $h(x) = f(g(x))$ for $x \in U$. On the intuitive level, $\pi\sigma$ is the functional composition of π and σ, but is only defined at those points where the composition makes sense.

The transmutation (X, id_X), where id_X is the identity function on X, will be denoted simply 1_X. For any $\pi, \sigma, \tau \in \mathsf{Trans}(X)$, we have

1. $1_X\pi = \pi 1_X$ (the *identity* property), and
2. $(\pi\sigma)\tau = \pi(\sigma\tau)$ (the *associativity* property).

A set M with a binary operation and identity element e satisfying the identity and associativity properties is known as a *monoid* [3]. An element a of a monoid M is said to be *invertible* if there exists $b \in M$ such that $ab = e = ba$. The set of invertible elements of M forms a group. For the

monoid $\mathsf{Trans}(X)$, the set of invertible elements is $\{(X, f) \mid f\colon X \to X \text{ is a 1-1 correspondence}\}$. This group is isomorphic to the group of permutations of X.

Although not every element of $\mathsf{Trans}(X)$ has an inverse, we can define the *pseudo-inverse* of any element as follows. Given $\pi = (S, f) \in \mathsf{Trans}(X)$, let $\sigma = (f(S), g)$, where g is defined by $g(f(x)) = x$ for all $x \in S$. This is well-defined since f is injective. Then $\sigma\pi = (S, \mathsf{id}_S)$ and $\pi\sigma = (f(S), \mathsf{id}_{f(S)})$. We call σ the *pseudo-inverse* of π.

We next define two basic kinds of transmutations, which form the "building blocks" of $\mathsf{Trans}(X)$.

Definition 2.2. Let $n \geq 1$, and let $x_1, \ldots, x_n$ be n distinct elements of X. Define $f\colon X \to X$ by

$$f(x) = \begin{cases} x_{i+1} & \text{if } x = x_i \text{ for some } 1 \leq i < n \\ x_1 & \text{if } x = x_n \\ x & \text{if } x \notin \{x_1, \ldots, x_n\}. \end{cases}$$

The transmutation $\pi = (X, f)$ is called a *cycle*, and is denoted $(x_1, \ldots, x_n)$. The *length* of the cycle is n, and is denoted $\mathsf{len}(\pi)$.

Note there are many different ways to denote the same cycle, e.g., $(x_1, x_2, x_3) = (x_2, x_3, x_1)$. However, the length of a cycle is well-defined.

A cycle is said to be *trivial* if it has length 1. Clearly, a trivial cycle equals 1_X.

Definition 2.3. Let $n \geq 1$, and let $x_1, \ldots, x_n$ be n distinct elements of X. Let $S = X \setminus \{x_n\}$. Define $f\colon S \to X$ by

$$f(x) = \begin{cases} x_{i+1} & \text{if } x = x_i \text{ for some } 1 \leq i < n \\ x & \text{if } x \notin \{x_1, \ldots, x_n\}. \end{cases}$$

The transmutation $\pi = (S, f)$ is called a *shift* and is denoted $[x_1, \ldots, x_n]$. The *length* of the shift is n, and is denoted $\mathsf{len}(\pi)$.

We now turn to the problem of factoring a transmutation.

Definition 2.4. Let $\pi = (S, f) \in \mathsf{Trans}(X)$. The *support* of π is the set

$$\mathsf{supp}(\pi) \equiv \{s \in S \mid f(s) \neq s\} \cup (X \setminus S).$$

Hence the support of π consists of the points that are either "moved" by f or where f is not defined.

Definition 2.5. Let $\pi, \sigma \in \mathsf{Trans}(X)$. We say π and σ are *disjoint* if $\mathsf{supp}(\pi) \cap \mathsf{supp}(\sigma) = \varnothing$.

The following is an easy exercise in the definitions:

Proposition 2.6. *If π and σ are disjoint transmutations of X then*

1. $\pi\sigma = \sigma\pi$, *and*
2. $\mathsf{supp}(\pi\sigma) = \mathsf{supp}(\pi) \cup \mathsf{supp}(\sigma)$.

```
1  procedure localRedistribute is
       input: n: Integer, p: array 0..n − 1 of Integer, blocks: array 0..n − 1 of Block;
2      q: array 0..n − 1 of Integer;  /* used to store pseudo-inverse of p */
3      tmp: Block;  /* temporary space to hold one block when redistributing a cycle */
4      for i ← 0 to n − 1 do q[i] ← −1;
5      for i ← 0 to n − 1 do  /* compute pseudo-inverse and check p is injective */
6          j ← p[i];
7          if j ≥ 0 then {assert q[j] < 0; q[j] ← i;}
8      for k ← 0 to n − 1 do
9          while k < n ∧ q[k] < 0 do k ← k + 1;
10         if k = n then break;
11         i ← k; j ← p[i];
12         while j ≥ 0 ∧ j ≠ k do {i ← j; j ← p[i];}
13         if j < 0 then  /* shift */
14             j ← i; i ← q[j];
15             while i ≥ 0 do {copy(blocks[j], blocks[i]); q[j] ← −1; j ← i; i ← q[j];}
16         else /* cycle */
17             if i ≠ k then  /* non-trivial cycle */
18                 copy(tmp, blocks[j]);
19                 while i ≠ k do {copy(blocks[j], blocks[i]); q[j] ← −1; j ← i; i ← q[j];}
20                 copy(blocks[j], tmp);
21             q[j] ← −1;
```

Figure 2: Local Redistribution Algorithm

Note that the support of a non-trivial cycle $(x_1, \ldots, x_n)$ is $\{x_1, \ldots, x_n\}$ ($n \geq 2$). The support of a shift $[x_1, \ldots, x_n]$ is $\{x_1, \ldots, x_n\}$ ($n \geq 1$). Hence two of these primitive transmutations are disjoint if the two sets of symbols occurring in their representations do not intersect. The following is proved in Appendix A:

Theorem 2.7 (Lipscomb, [4,5])**.** *Every transmutation can be expressed as a product of non-trivial disjoint cycles and shifts. This factorization is unique, up to order.*

Example 2.8. Let $X = \{1, 2, 3, 4, 5\}$, $S = \{1, 2, 3\}$, $f(1) = 2$, $f(2) = 1$, $f(3) = 4$, and $\pi = (S, f)$. Then $\pi = (12)[34][5]$ is the unique factorization of π as a product of non-trivial disjoint cycles and shifts.

Note. The identity transmutation is considered to have the trivial factorization consisting of no cycles or shifts.

2.2 Local algorithm

The local redistribution problem can be formulated as follows. We are given an array of data blocks indexed from 0 to $n - 1$, and a transmutation $\pi = (S, f)$ on the set $X = \{0, 1, \ldots n - 1\}$.

An algorithm solves the redistribution problem if for all $i \in S$, the contents of block $f(i)$ after redistribution equals the contents of block i before redistribution. We assume the algorithm works by issuing a sequence of instructions of the form $copy(s, t)$, which copies the contents of block t to block s, and that additional "scratch" blocks can be allocated.

The MADRE local redistribution algorithm is given in Figure 2. The transmutation π is represented as an integer array p of length n. A negative value for $p[i]$ indicates that i is not in S, i.e., that block i is free. The algorithm begins by checking that f is injective and computing the pseudo-inverse of π, storing this in array q (lines 4–7). A non-negative value for $q[i]$ indicates that block i is waiting to receive the contents of block $q[i]$; a negative value indicates that block i should not be updated. As soon as the correct data is copied into block i, $q[i]$ is set to -1, so that at termination all entries of q will be -1.

The algorithm proceeds by discovering the factors in the unique decomposition of π guaranteed by Theorem 2.7. The factors are discovered by iterating over the entries of q, looking for the first non-negative value. The index of this entry is k. We must next discover whether k belongs to a cycle or a shift. This is accomplished by repeatedly applying p until either we return to k or we reach a point i for which $p[i] < 0$; in the first case we have discovered a cycle, in the second a shift.

The redistribution for a shift $[x_1, \ldots, x_m]$ ($m \geq 1$) is handled by copying data from x_{m-1} to x_m, then from x_{m-2} to x_{m-1}, and so on, finally copying from x_1 to x_2. The number of copies used to execute the shift is $m - 1$.

The redistribution for a non-trivial cycle $(x_1, \ldots, x_m)$ ($m \geq 2$) is accomplished by first copying x_m to a temporary space, then copying x_{m-1} to x_m, and so on, and finally copying the temporary space to x_1. The number of copies is $m + 1$.

The total number of copies performed by the algorithm is

$$\mathsf{ncpy}(\pi) \equiv \sum_{\sigma \in \mathsf{Shift}(\pi)} (\mathsf{len}(\sigma) - 1) + \sum_{\tau \in \mathsf{Cyc}(\pi)} (\mathsf{len}(\tau) + 1),$$

where $\mathsf{Shift}(\pi)$ is the set of shift factors in the unique decomposition of π, and $\mathsf{Cyc}(\pi)$ is the set of non-trivial cyclic factors.

Theorem 2.9. *For any $\pi \in \mathsf{Trans}(X)$, $\mathsf{ncpy}(\pi)$ is the minimal number of copies that can occur in a redistribution solution for π.*

We now consider the complexity of the algorithm as a function of the number of blocks n. We assume the size of a block and the time required to execute *copy* are constant.

Theorem 2.10. *The functions for worst-case time and memory consumed by the local redistribution algorithm are both $O(n)$.*

3 Distributed Algorithms

We now formally describe the full parallel redistribution problem that was introduced in Section 1. The distributed data redistribution problem generalizes the local problem described in Section 2.2. For the distributed problem we assume a distributed-memory parallel program consisting of N processes. Each process maintains in its local memory a contiguous, fixed-sized array of data blocks of arbitrary fixed size. Let m_i be the number of blocks maintained by process i ($0 \leq i <$

N). A block is specified by its *address* (i, j), where i is the process rank and j ($0 \leq j < m_i$) is the array index. Let A denote the set of all addresses. A *distributed redistribution problem* is specified by a transmutation $\pi = (B, f) \in \mathsf{Trans}(A)$. As in the local case, a redistribution algorithm solves the problem if for all $b \in B$, the contents of $f(b)$ after redistribution equal the contents of b before redistribution. The distributed problem may be thought of as an "in-place" version of MPI_ALLTOALLV.

The algorithms implemented in MADRE are expected to satisfy certain additional requirements. The first is that each should be a complete algorithmic solution to the data redistribution problem, i.e., it should terminate in finite time with the correct result on any redistribution problem, even one with no free blocks on any process. Second, the additional memory required by the algorithm should depend only linearly on the problem size. To be precise, there must be (reasonably small) constants c_1, c_2, and c_3 such that the memory required by the algorithm on process i is at most $c_1 N + c_2 m_i + c_3 l$. Algorithms with a quadratic dependence on the problem size will not scale on current high-end machines. In particular, *no single process can store the entire global map π; it is only given the destination addresses for its own blocks*. Finally, each algorithm should consume other resources frugally; e.g., the number of outstanding MPI communication requests on a process must stay within reasonably small bounds.

The MADRE library includes a module for the management of blocks on a single process. This module maintains the destination addresses for each block and provides a procedure to sort the blocks on a single process by increasing destination rank (with all free spaces at the end). The sorting procedure uses the local redistribution algorithm of Section 2.2. The distributed algorithms depend heavily on these services, in particular because it is often necessary to create contiguous send and receive buffers.

Finally, the algorithms discussed here are restricted to a certain subset of MPI functions (referred to as "permissible" in [10]) that guarantees deterministic behavior. As shown in [10], the advantage gained by this is that the final state arrived at in any execution of the program depends only on the inputs to the program (e.g., the map), and not on any choices made by the MPI implementation. This makes the programs much easier to test and debug.

3.1 A Pinar-Hendrickson Parking Algorithm

Our first algorithm is an instance of the family of "parking" algorithms described in [7]. Pseudo-code for the algorithm is given in Figure 3. We first describe a more basic algorithm from [7], and then describe the modifications to the basic version which yield the parking algorithm.

The basic algorithm proceeds in a series of global phases. Within each phase, each process receives as much data as possible into its free space while sending out as much data as possible to other processes. When this communication completes, the space occupied by the sent data is reclaimed as free, the blocks are re-sorted, and the next phase begins. This continues until all blocks have arrived at their final destinations.

The protocol for determining how many blocks a process p will send and receive to/from other processes in a phase proceeds as follows. First, p informs each process q of the number of blocks it has to send to q. These can be thought of as "requests to send" on the part of p. At the same time, p receives similar requests from the processes that wish to send it data. If p does not have sufficient free space to receive all the incoming data, it uses some heuristic to apportion its free space among its sources. In any case, p sends each source a message stating how much of the request will

symbol	meaning
N	number of processes
$\mathsf{out}[i]$	number of blocks remaining to send to proc i
$\mathsf{in}[i]$	number of blocks remaining to receive from proc i
$\mathsf{send}[i]$	number of blocks to send to i in current phase
$\mathsf{recv}[i]$	number of blocks to receive from i in current phase
$\mathsf{park}[i]$	number of parking blocks to send to i in current phase
$\Delta[i]$	number parking spaces ($-$) or blocks to park ($+$) for proc i (root only)
free	number of free spaces on this proc

```
1  procedure main is
2      while true do
3          update out;
4          done ← (∑_i out[i] = 0 ? 1 : 0);
5          Allreduce(done, 1, INT, LAND);
6          if done then break;
7          computeDirectMoves();
8          addParking();
9          executePhase();
10 procedure computeDirectMoves is
11     Alltoall(out, 1, INT, in, 1, INT);
12     r ← free;
13     for i ← 0 to N − 1 do
14         recv[i] ← min{r, in[i]};
15         r ← r − recv[i];
16     Alltoall(recv, 1, INT, send, 1, INT);
17 procedure executePhase is
18     sortBlocks();
19     foreach {i | recv[i] > 0} do
20         post recv for recv[i] blocks from proc i
21     foreach {i | send[i] > 0} do
22         post send of send[i] blocks to proc i
23     free ← free + ∑_i(send[i] − recv[i]);
24     wait for all requests to complete;
25 procedure addParking is
26     foreach i do park[i] ← 0;
27     d ← (∑_i in[i]) − free;
28     s ← ∑_i send[i];
29     if d < 0 then δ ← d;
30     else if d > s then δ ← d − s;
31     else δ ← 0;
32     Gather(δ, 1, INT, Δ, 1, INT, root);
33     if myRank = root then
34         foreach {i | Δ[i] > 0} do
35             foreach j do
36                 t[j] ← num. parking blocks
                   proc i will send to proc j
37             if i ≠ root then
38                 Send(t, N, INT, i);
39             else
40                 foreach j do park[j] ← t[j];
41     else
42         if δ > 0 then
43             Recv(park, N, INT, root)
44     foreach i do
45         send[i] ← send[i] + park[i]
46     Alltoall(send, 1, INT, recv, 1, INT);
```

Figure 3: A Pinar-Hendrickson parking algorithm

be granted, i.e., the number of blocks it will receive from that source in the current phase. (Our implementation uses the *first-fit* heuristic of [7].) At the same time, p receives similar granting messages from its targets. At this point, p knows how many blocks it will send and receive to/from each process. The data is then transferred by initiating nonblocking send and receive operations for the specified quantities.

There are situations in which the basic algorithm described above does not terminate. This can happen, for example, if a cyclic dependency occurs among a set of processes with no free space.

Moreover, when it does complete it may take many more phases than necessary. For these reasons, Pinar and Hendrickson introduce "parking." The idea is that if p does not have enough free space to receive all of its incoming data in the next phase it can temporarily "park" data on processes with extra free space. The parked data will be moved to its final destination when space becomes available. A root process is used in each phase to match processes with data to park with those with extra free space. A parking algorithm will always complete, as long as there is at least one free space on at least one process. (Our implementation allocates a single free space if there are no free spaces at all so that the algorithm will complete in all cases.) Parking can also significantly reduce the number of phases, approximating to within a factor of 1.5 a minimal-phase solution, though this does not always reduce the run-time.

3.2 Cyclic Scheduler

The parking algorithm described above exhibits several potential weaknesses. First, the total quantity of data moved is greater than necessary, because parking blocks are moved more than once. Second, the division into phases, with an effective barrier between each phase, may limit the possible overlap of communication with communication. For example, if some processes require very little time to complete their work in a phase, they will block, waiting for other processes to complete the phase, when they could be working on the next phase. Third, the number of phases may blow up as the total number of free spaces decreases, and there is significant overhead required to execute each phase.

Like the parking algorithm, the cyclic scheduler algorithm uses a root process to help schedule actions on other processes. However, it differs in several ways. First, all blocks are moved directly to their final destinations. Second, the cyclic algorithm separates the process of *schedule creation* from the process of *schedule execution*. The schedule is created in the first stage of the algorithm, which relies heavily on the root. Once that stage completes, each process has the complete sequence of instructions it must follow in order to complete the redistribution. In the schedule execution stage, each process executes these instructions. In this stage, the root plays no special role, there are no effective barriers, and minimal overhead is required. Since the length of a schedule is bounded above by m_i, the memory overhead required to store the schedule is within our required limits. In all of our experiments to date, the time required to complete the schedule creation stage has been insignificant compared to the time required for the execution stage.

A *schedule* is a sequence of actions for a process to follow. There are three types of actions. The first has the form "Send q blocks to process p_1 in increments of c blocks," meaning that the process should send a message to p_1 containing c blocks, then another message of c blocks, and so on, until a total of q blocks have been sent. (The final message will consist of $q\%c$ blocks if c does not divide q.) The other action types are "Receive q blocks from p_2 in increments of c" and "Send q blocks to p_1 and receive q blocks from p_2 in increments of c." The last form requires that c blocks be sent and c received, and after those operations have completed, another c are sent and received, and so on. The blocks must be sorted once before an action begins, but do not have to be sorted again until the next action is executed. This is an important point which reduces the overhead associated to executing an action.

To create the schedule, the root essentially performs a depth-first search of the *transmission graph*. This is the weighted directed graph in which the nodes are the process ranks, and there is an edge from i to j of weight $k > 0$ if i has k blocks to send to j. It is not possible to store the

symbol	meaning
degree$[i]$	number of remaining edges departing from proc i in transmission graph
stack$[i]$	i^{th} element in DFS stack containing nodes of transmission graph
stackSize	current size of DFS stack
weight$[i]$	weight of current edge departing from proc i
dest$[i]$	destination node for current edge departing from i
free$[i]$	current number of free spaces on proc i

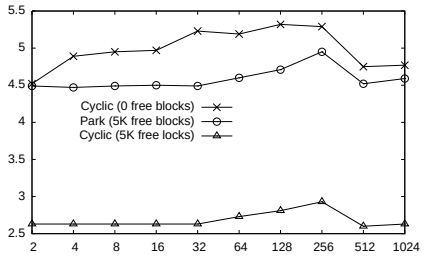
```
1  procedure main is
2      for i ← 0 to N − 1 do
3          while degree[i] > 0 do
4              push(i);
5              while stackSize > 0 do
6                  r ← peek();
7                  if degree[r] = 0 then
8                      scheduleShift();
9                  else
10                     d ← dest[r];
11                     p ← stackPos[d];
12                     if p < 0 then push(d);
13                     else scheduleCycle(p);
14 procedure scheduleCycle(s) is
15     q ← min_{i≥s}{weight[stack[i]]};
16     c ← min{q, max{1, min_{i≥s}{free[stack[i]]}}};
17     l ← peek();
18     while stackSize > s do
19         p_1 ← pop();
20         p_2 ← dest[p_1];
21         if stackSize > s then p_0 ← peek();
22         else p_0 ← l;
23         schedule send-recv on p_1 with source p_0,
           dest p_2, quantity q, and increment c;
24         weight[p_1] ← weight[p_1] − q;
25         if weight[p_1] = 0 then nextEdge(p_1);
```

Figure 4: The cyclic scheduler root scheduling algorithm.

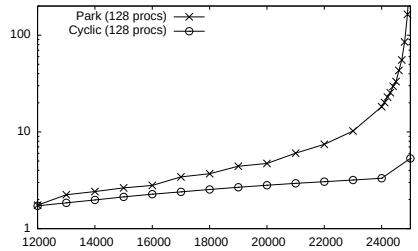
entire transmission graph on the root in linear memory. Instead, each process p sends the root one outgoing edge, i.e., the rank r of a single process to which p has data to send and the number of blocks p has to send to r. As soon as the root has received an edge from each process, it begins scheduling actions. This involves decrementing edge weights, and sending actions back to the non-root processes. When the weight of an edge from p reaches 0, the root requests a new edge from p. Hence edges are continually flowing in to the root and actions are continually flowing out, in a pipelined manner, which is how the memory required by the root is bounded by a constant times the number of processes.

The main part of the scheduling algorithm on the root is shown in Figure 4. The root uses a stack to perform the search of the graph. The stack holds nodes in the graph, i.e., process ranks. If p_0 and p_1 are two consecutive elements in the stack then there is an edge from p_0 to p_1. The stack grows until either (a) a cycle is reached, i.e., the destination rank of the edge departing from the last node on the stack is already on the stack, or (b) a node is reached with no remaining outgoing edges (i.e., no data to send). In case (a), an action is scheduled on each process in the cycle, while in case (b), an action is scheduled on each process in the stack. The algorithm for case (a) is shown. The functions for sending actions and retrieving edges are not shown; these involve parameters, such as the number of actions/edges to include in a single message, that provide some control over the memory/time trade-off.

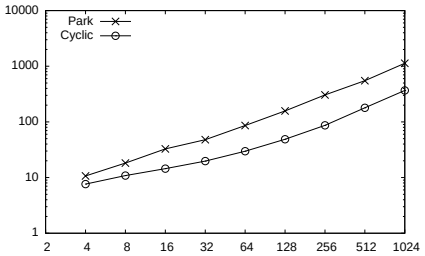
In case (a), the number of blocks q to be sent and received by each process is the minimum weight of an edge in the cycle. The increment is usually the minimum number of free spaces for



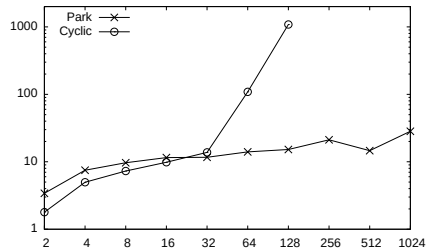
(a) Simple global cycle



(b) Effect of decreasing free memory



(c) All free space on one process



(d) Global transpose, 5,000 free blocks/proc

Figure 5: Time to redistribute data. In all cases, each process maintains 25,000 memory blocks of 16,000 bytes each and the y-axis is the time in seconds to complete a data redistribution. In (a), (c), and (d), the x-axis shows the number N of processes in logarithmic scale. In (b), (c), and (d) the time is logarithmic.

a process in the cycle. However, if there is a process with no free space, the increment is set to 1: in this case an extra free space reserved for this situation will be used to receive each incoming increment. When the cycle has been executed the extra space will again be free. This is the essential fact that guarantees the algorithm will always terminate, though it does require the allocation of an additional block on each process. The actions scheduled in case (b) are similar, though the first element in the stack performs only a send and the last performs only a receive.

The algorithm is most effective when it finds many long cycles or shifts with large quantities. The idea is that all of the actions comprising a single cycle/shift can execute in parallel and should take approximately the same amount of time. Moreover, as pointed out above, no local redistribution needs to take place during the execution of an action.

4 Experiments

We report here on a few experiments comparing the performance of the different algorithms. The experiments are designed to test the algorithms in situations of very high data movement and very limited memory. All experiments were executed on the 1024-node IBM Blue Gene/L at Argonne National Laboratory. In each experiment, each process maintains an array of 25,000 blocks, and each block consists of 16,000 bytes. Thus 400 MB are consumed by data, which is a significant portion of the 512 MB total RAM available on each node.

In the first experiment, each process has no free space, and wishes to send all 25K blocks to the next process in the cyclic ordering $i \mapsto (i+1)\%N$, where N is again the number of processes. The

data redistribution was timed for various values of N. The second experiment is similar, except that each process has 5K free spaces and sends 20K blocks. The results of these two experiments are shown in graph (a) of Figure 5. The time for the parking algorithm to complete the first experiment is not shown because it did not terminate after 30 minutes; the reasons for this will be explained below. Otherwise, the times range from 2.5 to 5.5 seconds and remain relatively constant as the number of processes is scaled in each case, suggesting near-perfect parallelism in the data transfer.

The third experiment is similar to the two above, except that the number of processes was fixed at 128 and the number of blocks to be transferred was scaled from 12K to 25K. (Equivalently, the number of free spaces was decreased from 13K to 0.) The results are shown in graph (b) of Figure 5, and illustrate how the time of the parking algorithm blows up with the decreasing amount of free space, in contrast with the cyclic algorithm. Our performance analyses reveal that this behavior is not due to the communication patterns, which are essentially the same in both cases. Rather, the difference arises because the parking algorithm requires that data be re-sorted at the end of each phase. In this experiment, the distribution of the blocks on process $N-1$ at the end of each phase presents a worst case scenario for the sort: the entire array of blocks must be shifted, requiring on the order of 25K calls to `memcpy` (of 16 KB) as the free space approaches 0. Moreover, the number of phases approaches 25K as the free space approaches 0, and so the time dedicated to sorting quickly blows up. For the cyclic algorithm, each process need only execute a single send-receive action of 25K blocks with increment equal to the number of free spaces. Because a sort is only required at the end of each action, the cyclic algorithm performs only one sort for the entire execution.

In the fourth experiment, one process has 25K blocks of free space and all others have no free space. Each of the processes with data wishes to send an approximately equal portion of its data to all other processes. The times are shown in Figure 5(c). Both algorithms complete for every process count, though the time clearly grows exponentially for both.

In the fifth experiment, each process has 5K free spaces 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.) The results are shown in Figure 5(d). In this case the cyclic algorithm blows up with process count, and cannot scale beyond 128 nodes without exceeding our 30-minute limit. In contrast, the parking algorithm scales reasonably well and completes the 1024-node redistribution in only 28 seconds.

Further investigation revealed the reason for the discrepancy. For this redistribution problem, the transmission graph is the complete graph in which all edges have approximately equal weight. Moreover, in our implementation each process orders its outgoing edges by increasing destination rank. The result is that all of the actions scheduled by the root are cycles of length 2. These are scheduled in the "dictionary order"

$$\{0,1\},\{0,2\},\ldots,\{0,N-1\},\{1,2\},\{1,3\},\ldots,\{1,N-1\},\ldots,\{N-1,N\}.$$

The execution of one of these pairs involves the complete exchange of data between the two processes. If all exchanges take approximately the same time, execution will proceed in $2N+3$ phases: in phase k, the exchanges for all pairs $\{i,j\}$, where $i+j=k$ and $i \neq j$, will take place. This means that many processes will block when they could be working. For example, in phase 1 only processes 0 and 1 will exchange data, while all other processes wait. In contrast, the parking algorithm completes in 4 stages for any N. Hence, in this case, the parking algorithm achieves much greater overlap of communication.

5 Conclusion

For a certain class of scientific applications, the limited-memory data redistribution problem will become an increasingly important component of overall performance and scalability as we move toward the petascale. Solutions to this problem are difficult and subtle. Solution algorithms can behave in ways that are difficult or impossible to predict using purely analytical means, so experimentation is essential for ascertaining their qualities. We have implemented a practical framework with multiple solutions to the problem, two of which are explored in this paper. A series of simple experiments helped us identify some potential weaknesses in a variant of the Pinar-Hendrickson parking algorithm. We addressed these in a new algorithm, only to discover different situations in which that algorithm also performs poorly.

We continue to refine the algorithms presented here and to explore entirely new ones. In addition, we are exploring heuristics capable of predicting which algorithm will perform well on a given problem. It may also be possible to combine different algorithms, so that in certain states the algorithm will be switched dynamically, in the middle of the redistribution. Finally, we are preparing another set of experiments in which we integrate MADRE directly into scientific applications, as opposed to the synthetic experiments reported on here.

Acknowledgements. We are grateful to Ali Pinar for answering questions about the parking algorithm and making the code used in [7] available to us. We also thank Anthony Chan for assistance with the Jumpshot performance visualization tool [11].

A Proofs

Proof of of Theorem 2.7. We first establish some notation. Let $S \subseteq X$ and $f\colon S \to X$ an injective function. For any integer i, we will define a function f^i from a subset of X to X. For $i = 0$, $f^0 = \mathsf{id}_X$. For $i > 0$, $f^i(x) = f(f^{i-1}(x))$ and the domain of f^i consists of all x in the domain of f^{i-1} such that $f^{i-1}(x) \in S$. For $i < 0$, $f^i(x)$ is defined if there exists $y \in X$ such that $f^{-i}(y) = x$, in which case $f^i(x) = y$; this is well-defined since f is injective. Note that $f^{i+j}(x) = f^i(f^j(x))$ for any integers i and j and any x for which the expressions are defined. We define

$$f^*(x) = \{f^i(x) \mid i \in \mathbf{Z} \text{ and } x \text{ is in the domain of } f^i\}$$

Note that $x \in f^*(x)$, by taking $i = 0$.

The proof is by induction on $|\mathsf{supp}(\pi)|$. If $\mathsf{supp}(\pi) = \varnothing$ then $\pi = 1_X$, which has the trivial factorization. So suppose $|\mathsf{supp}(\pi)| \geq 1$ and that the theorem holds for any transmutation with support of size less than $|\mathsf{supp}(\pi)|$. We will show how to express π as product of disjoint transmutations σ and τ, where τ is a non-trivial cycle or shift and $\mathsf{supp}(\sigma) \subsetneq \mathsf{supp}(\pi)$. The inductive hypothesis can then be applied to σ, and Proposition 2.6(2) ensures the factors of σ will all be disjoint from τ, completing the proof.

Write $\pi = (S, f)$. Let $t \in \mathsf{supp}(\pi)$, and $T = f^*(t)$. Let $\sigma = (S \cup T, g)$, where

$$g(x) = \begin{cases} x & \text{if } x \in T \\ f(x) & \text{if } x \in S \setminus T. \end{cases}$$

The injectivity of g follows from the injectivity of f and the fact that if $x \in S \setminus T$ then $f(x) \notin T$. Hence $\sigma \in \mathsf{Trans}(X)$ and $\mathsf{supp}(\sigma) = \mathsf{supp}(\pi) \setminus T \subsetneq \mathsf{supp}(\pi)$ as $t \in T$.

Let $\tau = (X \setminus (T \setminus S), h)$, where

$$h(x) = \begin{cases} x & \text{if } x \in X \setminus T \\ f(x) & \text{if } x \in S \cap T. \end{cases}$$

The injectivity of h follows from that of f and the fact that if $x \in S \cap T$ then $f(x) \in T$. Hence $\tau \in \mathsf{Trans}(X)$ and $\mathsf{supp}(\tau) = (T \setminus S) \cup (T \cap \mathsf{supp}(\pi))$. Furthermore, it is not possible that $\tau = 1_X$. For if this were the case then we would have $T \setminus S = \varnothing$, i.e., $T \subseteq S$, and hence $t \in S$. The definition of h would then imply $f(t) = t$, which contradicts that assumption that $t \in \mathsf{supp}(\pi)$.

It is clear that σ and τ are disjoint and that $\pi = \sigma\tau$. It remains to see that τ is either a cycle or a shift.

Case 1: $T \subseteq S$. We will show τ is a cycle. Since $T \subseteq S$, $f^i(t)$ is defined for every integer i. Since S is finite, there must exist $j > k \geq 0$ such that $f^j(t) = f^k(t)$. Since f is injective, $f^{j-k}(t) = t$. Let m be the least positive integer such that $f^m(t) = t$. Given any integer i, write $i = mq + r$, where $0 \leq r < m$. Then $f^i(t) = f^{mq+r}(t) = f^r(f^{mq}(t)) = f^r(t)$. This shows that $T = \{t, f(t), \ldots, f^{m-1}(t)\}$, from which it follows τ is the cycle $(t, f(t), \ldots, f^{m-1}(t))$.

Case 2: $T \not\subseteq S$. We will show τ is a shift. First, it is not possible that $f^i(t)$ is defined for all $i \geq 0$. For if this were the case, we could argue as in Case 1 to show that $f^m(t) = t$ for some m, and from there that $T \subseteq S$. So let m be the least positive integer such that t is not in the domain of f^m. Similarly, it is not possible that $f^i(t)$ be defined for all $i \leq 0$, so let n be the greatest negative integer such that t is not in the domain of f^n. It follows that τ is the shift $[f^{n+1}(t), \ldots, f^{-1}(t), t, f(t), \ldots, f^{m-1}(t)]$. □

Proof of Theorem 2.9. Let $\mu(\pi)$ denote the minimal number of copies that can occur in a solution for π. We first claim that if $\pi = \pi_1\pi_2$, where π_1 and π_2 are disjoint transmutations, then $\mu(\pi) = \mu(\pi_1) + \mu(\pi_2)$. To see this, note that any copy instruction in a redistribution solution involves copying data that originated in some block. Given any solution for π, the subsequence of copy instructions for data that originated in $\mathsf{supp}(\pi_1)$ will be a solution for π_1. A similar statement holds for π_2. Since the supports of π_1 and π_2 are disjoint, $\mu(\pi) \geq \mu(\pi_1) + \mu(\pi_2)$. On the other hand, suppose we are given solutions for π_1 and π_2. If the solution for π_1 uses any blocks in $\mathsf{supp}(\pi_2)$, replace these with temporary blocks to get another solution of the same length; likewise for π_2. Now a solution for π can be obtained by appending the solution for π_2 to the one for π_1, showing $\mu(\pi) \leq \mu(\pi_1) + \mu(\pi_2)$.

This means we only need to show that $\mathsf{ncpy}(\pi) = \mu(\pi)$ for π a cycle or a shift. If π is a shift of length m, there are precisely $m - 1$ blocks that must be moved to new locations, so clearly $\mu(\pi) \geq m - 1 = \mathsf{ncpy}(\pi)$, so $\mathsf{ncpy}(\pi) = \mu(\pi)$ in this case. If π is a cycle of length $m \geq 2$, there are m blocks that must be moved to new locations. Given any solution, consider the first copy from a block in $S \equiv \mathsf{supp}(\pi)$. The target of this copy cannot be in S, because that would overwrite data that has not yet been copied to another location. Hence the first copy must be into some block not in S. So after this first copy completes, there are still m blocks that need to be copied to their final destinations. Hence $\mu(\pi) \geq m + 1$, and we have $\mathsf{ncpy}(\pi) = \mu(\pi)$ in this case as well. □

Proof of Theorem 2.10. Space is clearly $O(n)$. To see that time is $O(n)$, we argue that the body of each loop is executed no more than n times. This is obvious for the loops on lines 4, 5, and

8. The loop on line 9 always begins with a value of k at least one larger than the final value of k from the previous iteration, so can iterate no more than n times. Now, X is the disjoint union of the sets $\mathsf{supp}(\sigma)$ ($\sigma \in \mathsf{Cyc}(\pi) \cup \mathsf{Shift}(\pi)$) and the sets $\{x\}$ (for all $x \in X$ such that $f(x) = x$). After any execution of the body of the loop of line 8, $q[x] = -1$ for all x in the partition containing k. Since control can only reach line 11 if $q[k] \geq 0$, this implies that the inner loops (lines 12, 15, 19) are executed at most once for each σ. Moreover, each inner loop iterates at most once over the elements of $\mathsf{supp}(\sigma)$. Hence the total number of iterations of each inner loop is at most n. □

References

[1] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.

[2] H. L. deCougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Özturan, and M. S. Shephard. Load balancing for the parallel adaptive solution of partial differential equations. *Appl. Numer. Math.*, 16(1-2):157–182, 1994.

[3] J. M. Howie. *Fundamentals of Semigroup Theory*. Oxford University Press, 1995.

[4] S. Lipscomb. Cyclic subsemigroups of symmetric inverse semigroups. *Semigroup Forum*, 34:243–248, 1986.

[5] S. Lipscomb. *Symmetric Inverse Semigroups*. American Mathematical Society, Providence, RI, 1996.

[6] C. Nieter and J. R. Cary. Vorpal: a versatile plasma simulation code. *J. Comput. Phys.*, 196(2):448–473, 2004.

[7] A. Pinar and B. Hendrickson. Interprocessor communication with limited memory. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.

[8] P. M. Ricker, B. Fryxell, K. Olson, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: A multidimensional hydrodynamics code for modeling astrophysical thermonuclear flashes. volume 31 of *Bulletin of the American Astronomical Society*, page 1431, Dec. 1999.

[9] S. F. Siegel. The MADRE web page. `http://vsl.cis.udel.edu/madre`, 2008.

[10] S. F. Siegel and G. S. Avrunin. Verification of halting properties for MPI programs using nonblocking operations. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 326–334. Springer, 2007.

[11] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, 1999.