MADRE: The Memory-Aware Data Redistribution Engine*

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. A new class of algorithms is presented for efficiently carrying out many-to-many parallel data redistribution in a memory-limited environment. Key properties of these algorithms are explored, and their performance is compared using idealized benchmark problems. These algorithms form part of a newly developed MPI-based library MADRE (Memory-Aware Data Redistribution Engine), an open source toolkit designed for easy integration with application codes to improve their performance, portability, and scalability.

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 locally, and so forth. The need for efficient many-to-many parallel data movement arises in a wide variety of such scientific applications (e.g., [1]). 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 particle-tracking codes are another good example, with frequent rebalancing of the main data structures as particles cross processor boundaries [3].

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 [4]. The second problem, though, has received little attention, especially in the critical case where memory

* 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.

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 [5]. They formulated a "phase"-based framework for the problem, showed that the problem of finding a minimal-phase solution is NP-hard, and studied a family of algorithms for approximating minimal solutions. Yet there are many very different solutions to the redistribution problem that do not necessarily fit into the phase-based framework and much experimentation will be necessary to compare their performance and ascertain their various qualities.

We describe here a new class of memory-aware parallel redistribution algorithms that extends the pioneering work of Pinar and Hendrickson. Properties of these algorithms are explored, and some initial performance tests are carried out. The algorithms presented form only a small part of the larger MADRE effort (Memory-Aware Data Redistribution Engine) [6], an open source, C/MPI-based toolkit that includes a wide range of strategies that allow the client to control tradeoffs between buffer space, performance, and scalability. Moreover, MADRE is architected as well to serve as a testbed for continued research in the area of parallel data redistribution, where developers can easily integrate new techniques/strategies within the framework. An excerpt of the MADRE interface is shown in Fig. 1.

2 Algorithms

In this section, we define the data redistribution problem precisely, and describe two of the algorithms implemented in MADRE for solving it.

Assume we are given a distributed-memory parallel program consisting of n processes. Each process maintains in its local memory a contiguous, fixed-sized array of data blocks, each block comprising l bytes. 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 *redistribution problem* is specified by a subset $B \subseteq A$ and an injective map $f\colon B \to A$. 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 problem may be thought of as an "in-place" version of MPI_ALLTOALLV. The blocks in $A \setminus B$ are said to be *free*; there is no requirement that their data be preserved, so the algorithm is free to use them as "scratch space."

The algorithms implemented in MADRE are expected to satisfy certain 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

```
MADRE_Object MADRE_create(void* data, char *strategy, int numBlocks,
    int blockLength, MPI_Datatype datatype, MPI_Comm comm);

/* Redistributes the blocks based on the given map information.
 * destRanks and destIndices are both integer arrays of the given
 * length.  We must have 0 <= length <= madre->numBlocks.
 * destRanks[i] is the rank of the proc to which block i is to be
 * moved, or -1 if block i is dead.  destIndices[i] is the position to
 * which block i is to be moved (this integer is ignored if
 * destRanks[i] is -1).  The blocks in positions length, length+1,
 * ..., madre->numBlocks-1 are assumed to be dead. */
void MADRE_redistribute(MADRE_Object madre, int length,
    int* destRanks, int* destIndices);

void MADRE_destroy(MADRE_Object madre);

void MADRE_setDataPointer(MADRE_Object madre, void* data);

/* Current number of bytes allocated by MADRE */
long MADRE_getMem();
```

Fig. 1. Excerpt of MADRE interface, file `madre.h`

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 "knows" the global map; 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 function to sort the blocks on a single process by increasing destination rank (with all free spaces at the end). All of the MADRE redistribution algorithms depend heavily on these services, in particular because it is often necessary to create contiguous send and receive buffers. For the most part, however, we will elide these details in our descriptions of the algorithms.

2.1 A Pinar-Hendrickson Parking Algorithm

Our first algorithm is an instance of the family of "parking" algorithms described in [5]. We will briefly summarize the algorithm and refer the reader to [5] for details.

The 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 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 [5].) 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.

2.2 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 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 Fig. 2. 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

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

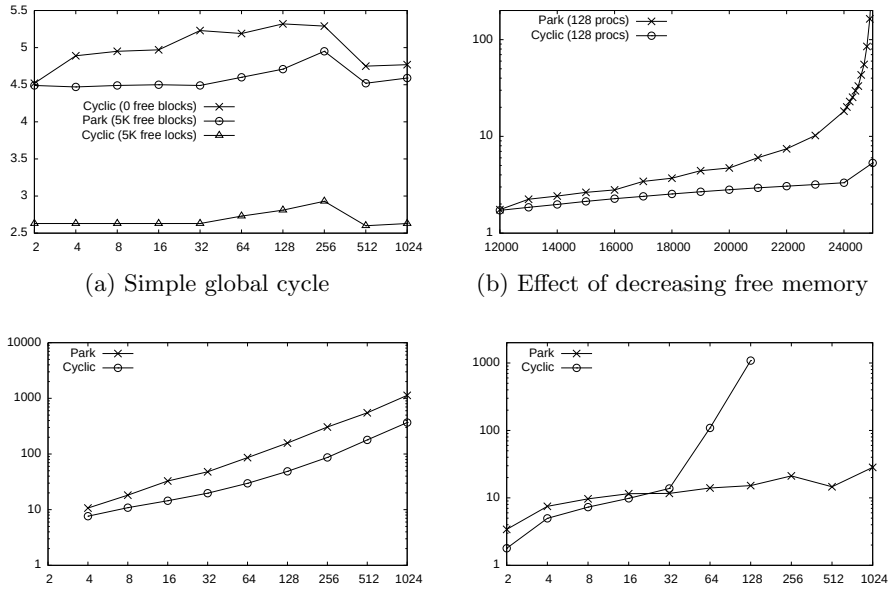
```
1  procedure main is
2      for i ← 0 to n − 1 do
3          while degree[i] > 0 do
4              push(i);
5              while stackSize > 0 do
6                  r ← peek();
7                  if degree[r] = 0 then
8                      scheduleShift();
9                  else
10                     d ← dest[r];
11                     p ← stackPos[d];
12                     if p < 0 then
13                         push(d)
14                     else
15                         scheduleCycle(p)

16 procedure scheduleCycle(s) is
17     q ← min_{i≥s}{weight[stack[i]]};
18     c ← min{q, max{1, min_{i≥s}{free[stack[i]]}}};
19     l ← peek();
20     while stackSize > s do
21         p1 ← pop();
22         p2 ← dest[p1];
23         if stackSize > s then
24             p0 ← peek()
25         else
26             p0 ← l
27         schedule send-recv on p1 with source
           p0, dest p2, quantity q, and
           increment c;
28         weight[p1] ← weight[p1] − q;
29         if weight[p1] = 0 then nextEdge(p1)
```

Fig. 2. The cyclic scheduler root scheduling algorithm.

the number of actions/edges to include in a single message, that provide some control over the memory/time tradeoff.

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 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.



(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

Fig. 3. 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.

3 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 $i \mapsto (i+1)\%n$. 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 Fig. 3. 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 Fig. 3, 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 Fig. 3(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 $m = 20\text{K}$. (The map is essentially a global transpose of the data matrix.) The results are shown in Fig. 3(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 m, the exchanges for all pairs $\{i,j\}$, where $i+j=m$ 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.

4 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 [5] available to us. We also thank Anthony Chan for assistance with the Jumpshot performance visualization tool [7].

References

1. Ricker, P.M., Fryxell, B., Olson, K., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Tufo, H.: FLASH: A multidimensional hydrodynamics code for modeling astrophysical thermonuclear flashes. Volume 31 of Bulletin of the American Astronomical Society. (December 1999) 1431
2. deCougny, H.L., Devine, K.D., Flaherty, J.E., Loy, R.M., Özturan, C., Shephard, M.S.: Load balancing for the parallel adaptive solution of partial differential equations. Appl. Numer. Math. **16**(1-2) (1994) 157–182
3. Nieter, C., Cary, J.R.: Vorpal: a versatile plasma simulation code. J. Comput. Phys. **196**(2) (2004) 448–473
4. Catalyurek, U., Boman, E., Devine, K., Bozdag, D., Heaphy, R., Riesen, L.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07), IEEE (2007)
5. Pinar, A., Hendrickson, B.: Interprocessor communication with limited memory. IEEE Transactions on Parallel and Distributed Systems **15**(7) (July 2004)
6. Siegel, S.F.: The MADRE web page. `http://vsl.cis.udel.edu/madre` (2008)
7. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. Int. J. High Perform. Comput. Appl. **13**(3) (1999) 277–288