The MADRE Manual

Version 0.2

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

April 30, 2008

[1]Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark DE 19716, USA. *E-mail:* `siegel` at `cis` dot `udel` dot `edu`

[2]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne IL 60439, USA. *E-mail:* `siegela` at `mcs` dot `anl` dot `gov`

Contents

Chapter 1

Introduction

MADRE stands for Memory-Aware Data Redistribution Engine. The MADRE Library is a C library using the Message Passing Interface (MPI) to move blocks of data among processes in a parallel or distributed application.

The MADRE web page is `http://vsl.cis.udel.edu/madre`. The web page contains the latest distribution of the library, associated papers, and documentation (including this manual).

Chapter 2

Background

2.1 Motivation

The problem of data redistribution arises in a wide variety of parallel scientific applications. Examples include applications requiring extensive load-balancing, such as algorithms using adaptive mesh refinement or particle-based codes. A typical solution to a load-balancing problem consists of two parts. First, one must compute a map that specifies where each block of data is to be sent in order to achieve an appropriately balanced distribution. After this map has been determined, one must find a way to move the data blocks to their new locations as efficiently as possible, and without exceeding the memory available to the application on any process. There has been much research into the first problem, including, for example, algorithms for computing space-filling curves. MADRE addresses the second problem. This problem has received much less attention, though it is becoming increasingly clear that it is at least as hard as the first and is a major obstacle to scalability.

Many solutions to the redistribution problem used in production applications today are naïve for several reasons. For example, many solutions require each process to reserve a certain amount of application memory as buffer space for receiving and storing all incoming data blocks until all outgoing blocks have been sent out. This inefficient use of memory effectively reduces the overall amount of memory available to the application by a factor of two. Other solutions require the entire global map to be stored in the local memory of a single process. While this may work well on small architectures, it will not scale to systems consisting of hundreds of thousands of processors, which lie inevitably on the path to petascale computing. This is especially true if those systems provide only a small amount of memory per node, as is the case with the IBM Blue Gene/L.

The purpose of MADRE is to provide a variety of solutions to the redistribution problem. The algorithms differ in their memory-time tradeoffs and in other ways. Which is most effective depends on many factors. One of the goals of the MADRE project is to determine heuristics for deciding which algorithm is most effective in which contexts.

2.2 Description of Problem

The basic problem solved by MADRE can be explained as follows. First, the context: consider a message-passing application consisting of n processes executing in parallel. Each process has its own local memory; there is no shared memory. The processes are numbered $0, 1, \ldots, n-1$; this number is called the *rank* of the process.

Say the application produces and manipulates blocks of data, each block occupying m bytes

of memory. Assume the application allocates a fixed amount of memory on each process to store these data blocks; say the amount of memory allocated on process i for this purpose is a_i blocks, i.e., ma_i bytes. (Often a_i is the same for all i, but MADRE does not assume this.) We will assume the a_i memory blocks are numbered $0, 1, \ldots, a_i - 1$, and we will refer to this number as the *index* of a memory block on process i. For now, we also assume that the ma_i bytes occupy one contiguous region in the local memory of process i, though in a future release of MADRE we hope to relax that assumption.

Let $S = \{(i, j) \mid 0 \leq i < n, 0 \leq j < a_i\}$. This is the set of all *block addresses*. The first coordinate is the rank of the process containing the block, the second is the index of the block.

At any point in time during the execution of the application, some of the memory blocks may be *free*, that is, they are not currently holding useful data, and can therefore be used as scratch space, by MADRE or some other component of the application. Let us fix a point in time during execution, and suppose that at this point the number of *live* (i.e., non-free) blocks on process i is b_i. Of course, $b_i \leq a_i$ for all i. We do not assume that the live blocks occupy contiguous indices, though this is often the case.

Suppose at this point in time, the application desires to redistribute the data blocks, and permit the use of its free memory blocks for this purpose. This may entail copying data blocks to new processes or to new addresses on the same process. The particular redistribution problem may be specified as an injective function f from a subset T of S to S. The elements of $S \setminus T$ are the addresses of the free memory blocks. To say that f is injective means that $f(x_1) \neq f(x_2)$ whenever $x_1 \neq x_2$. The injective requirement is certainly necessary for a meaningful redistribution problem, because it would not make sense to copy the contents of two distinct memory locations into the same memory location.

The requirements of a solution to the redistribution problem can now be stated simply as follows:

For all $x \in T$, the contents of the memory at address $f(x)$ after redistribution should equal the contents of the memory at address x before redistribution.

This is the only requirement. In particular, if $y \in T \setminus f(S)$, the contents of the memory at address y after redistribution is undefined.

2.3 The MADRE Library

In MADRE, each process specifies the destinations for the blocks on that process. Hence the global map f is not "known" to any single process. This information is instead distributed among the processes.

The destinations are specified by providing two integers for each block: the destination rank and destination index. The value -1 is used for the destination rank to indicate that a block is free.

To use MADRE, each MPI process must first invoke `MADRE_create` in order to create a `MADRE_Object`. This function requires a pointer to the region of memory containing that process' data, the number of blocks of data, the number of elements per block, the type of each element, a string identifying the algorithm to be used, and an MPI communicator handle.

To carry out a redistribution, the `MADRE_redistribute` function is called with a reference to this MADRE object and two arrays containing the destination information for the blocks on that process. This is a collective function that should be called by all processes in the communicator associated with the object. The MADRE object can be re-used as many times as one likes to carry out repeated redistributions.

When the application is finished with the MADRE object, the object is destroyed by calling

the `MADRE_destroy` function. Additional functions provide ways to get information about the `MADRE_Object` and the memory usage of the MADRE library.

Chapter 3

Download and Installation

You should already have an MPI implementation installed.

1. Download the latest version of MADRE from `http://vsl.cis.udel.edu/madre`. This will come in the form of a gzipped tar archive `madre.tgz`.
2. Untar/gunzip `madre.tgz`. You should see a directory `madre`.
3. (Optional). Change into the `madre/include` directory. Create a file with a name like `local-myPlatformName.mk` based on the default local configuration file `local.mk`.
4. Change into the `madre` directory. Type `./setup` (to use the default local configuration file) or `./setup myPlatformName` (to use your own local configuration file). This just copies the appropriate local configuration file from `include` into the `madre` directory.
5. Type `make`. Everything should compile without warnings. In the end you should have a file `lib/libmadre.a`.

To use the MADRE library in your application, add the line

```
#include "madre.h"
```

in your source files, at some point after `stdlib.h`, `stdio.h`, and `mpi.h` are included. (MADRE requires those three standard C libraries.) You can compile and link in the usual way. For example, to compile you might execute the following command:

```
mpicc -c -IpathToMadre/madre/include application.c
```

To link, you will execute something like

```
mpicc -LpathToMadre/madre/lib -lmadre -o application application.o ...
```

Chapter 4

Interface

In this section we give the syntax for each of the exported MADRE functions. These are all declared in the file `include/madre.h`. We also give a brief description of the semantics of each function.

4.1 `MADRE_create`: Create a MADRE object

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

`data`	pointer to memory region containing data blocks
`strategy`	name of the redistribution strategy, e.g., "cycleShiftBred"
`numBlocks`	number of blocks in data region
`blockLength`	number of data elements per block
`datatype`	MPI datatype of an element
`comm`	the MPI communicator handle, e.g., `MPI_COMM_WORLD`

Creates a MADRE object with given parameters. This object can be used repeatedly to carry out data redistribution. The communicator `comm` specifies the processes that will take part in the redistributions carried out with this MADRE object. Each process belonging to `comm` should create a MADRE object. The arguments `strategy`, `datatype`, `blockLength`, and `comm` should be the same on all processes. Only `data` and `numBlocks` can differ from process to process. The pointer `data` should point to a region of memory containing `numBlocks*blockLength` elements of type `datatype`. The communicator used by MADRE is actually a clone of the one passed in to this function; this ensures that no messages sent by MADRE functions can ever "leave" the MADRE library.

4.2 `MADRE_getDataPointer`: Get data pointer

```
void* MADRE_getDataPointer(MADRE_Object madre)
```

`madre`	MADRE object

Returns the data pointer for this MADRE object.

4.3 `MADRE_getBlockLength`: Get the number of elements per block

```
int MADRE_getBlockLength(MADRE_Object madre)
```

`madre` MADRE object

Returns block length (number of data elements per block).

4.4 `MADRE_getNumBlocks`: Get the number of blocks in data region

```
int MADRE_getNumBlocks(MADRE_Object madre)
```

`madre` MADRE object

Returns the number of blocks in data region.

4.5 `MADRE_getDatatype`: Get the MPI datatype

```
MPI_Datatype MADRE_getDatatype(MADRE_Object madre)
```

`madre` MADRE object

Returns MPI datatype of each data element.

4.6 `MADRE_getComm`: Get the MPI communicator

```
MPI_Comm MADRE_getComm(MADRE_Object madre)
```

`madre` MADRE object

Returns the MPI communicator handle used by MADRE object. This is a clone of the original communicator passed to the constructor of the MADRE object.

4.7 `MADRE_getBlockBytes`: Get the number of bytes per block

```
int MADRE_getBlockBytes(MADRE_Object madre)
```

`madre` MADRE object

Returns the number of bytes occupied by one block.

4.8 `MADRE_setDataPointer`: Set the pointer to the data region

```
void MADRE_setDataPointer(MADRE_Object madre, void *data)
```

`madre`	MADRE object
`data`	new pointer to data region

Changes the data pointer to the given pointer. This is the only thing you can change once the MADRE object has been created.

4.9 `MADRE_destroy`: Destroy the MADRE object

```
void MADRE_destroy(MADRE_Object madre)
```

`madre`	MADRE object

Destroys the MADRE object and deallocates all resources (memory, MPI communicators, etc.) that had been allocated for it.

4.10 `MADRE_redistribute`: Redistribute data

```
void MADRE_redistribute(MADRE_Object madre,
                        int length,
                        int* destRanks,
                        int* destIndices )
```

`madre`	the MADRE object
`length`	length of `destRanks` and `destIndices` arrays
`destRanks`	destination ranks for blocks
`destIndices`	destination indices for blocks

Redistributes the blocks according to the map specified by the given arguments. This is a collective communication, it must be called by all processes in the communicator associated to the MADRE object. The arrays `destRanks` and `destIndices` have length `length`. We must have $0 \leq$ `length` $\leq$ `numBlocks`. `destRanks[`i`]` is the rank of the process to which block i is to be moved, or -1 if block i is free. `destIndices[`i`]` is the position to which block i is to be moved on the destination process (this integer is ignored if `destRanks[`i`]` is -1). The blocks in positions `length`, `length` $+ 1$, ..., `numBlocks` $- 1$ are assumed to be free. Note that one must specify the destination rank and index of a block that is to stay in its current location.

4.11 `MADRE_print`: Print state of MADRE object

```
void MADRE_print(FILE *stream, MADRE_Object madre)
```

`stream`	output stream, e.g., `stdout`
`madre`	MADRE object

Prints abundant information on the state of the MADRE object. This is really intended for debugging and small test cases only.

4.12 MADRE_getAllocationCount: Get current number of allocated objects

```
long MADRE_getAllocationCount()
```

This function returns the current total number of heap-allocated objects generated by MADRE on this process. It takes no arguments. This number is the number of calls to `malloc` for which there has not yet occurred subsequent call to `free`. This has been used primarily for debugging, to help ensure there are no memory leaks.

4.13 MADRE_getMem: Get current amount of heap-allocated memory

```
long MADRE_getMem()
```

Returns the total number of bytes of heap-allocated data currently in use by the MADRE library on the calling process.

4.14 MADRE_getPeakMem: Get peak memory usage

```
long MADRE_getPeakMem()
```

Returns the *peakMem*, the maximum value achieved so far (from beginning of execution or last call to `resetMemMonitor`) of total number of bytes allocated (but not freed) by MADRE on this process.

4.15 MADRE_computeMemStats: Compute collective memory stats

```
void MADRE_computeMemStats(MPI_Comm comm)
```

`comm` the MPI communicator over which collective computation takes place

Using the given communicator, this functions computes the *peakMemSum*, *peakMemMax*, *peakMemMin*, and *peakMemMean*. These quantities are, respectively, the sum, maximum, minimum, and mean of the *peakMem* over all processes in the communicator. These quantities can then be retrieved using the functions `MADRE_getPeakMemSum`, `MADRE_getPeakMemMax`, `MADRE_getPeakMemMin`, and `MADRE_getPeakMemMean`. This is a collective operation over `comm`.

4.16 MADRE_getPeakMemSum: Get the sum of peak memory over all procs

```
long MADRE_getPeakMemSum()
```

Returns the sum over all processes of *peakMem*. This should only be called after the function `MADRE_computeMemStats` has been invoked.

4.17 MADRE_getPeakMemMax: Get the max of peak memory over all procs

```
long MADRE_getPeakMemMax()
```

Returns the max over all processes of *peakMem*. This should only be called after the function `MADRE_computeMemStats` has been invoked.

4.18 MADRE_getPeakMemMin: Get the min of peak memory over all procs

```
long MADRE_getPeakMemMin()
```

Returns the min over all processes of *peakMem*. This should only be called after the function `MADRE_computeMemStats` has been invoked.

4.19 MADRE_getPeakMemMean: Get the mean of peak memory over all procs

```
long MADRE_getPeakMemMean()
```

Returns the mean over all processes of *peakMem*. This should only be called after the function `MADRE_computeMemStats` has been invoked.

4.20 MADRE_resetMemMonitor: Reset memory count to 0.

```
void MADRE_resetMemMonitor()
```

Resets all current memory statistic to 0. This can only be called when there are no live memory alloations, i.e., when `MADRE_getAllocationCount` returns 0. Thus, the next computation of peak memory values will be from the invoation of this function.

4.21 MADRE_printPeakMem: Print current *peakMem* for this process

```
void MADRE_printPeakMem(FILE *stream)
```

`stream` output stream

Prints a one-line message to `stream` giving current value of *peakMem* for this process.

4.22 MADRE_printMemStats: Print current memory statistics

```
void MADRE_printMemStats(FILE *stream)
```

`stream` output stream

Print the statistics *peakMemSum*, *peakMemMax*, *peakMemMin*, and *peakMemMean* to the `stream`. This should only be called after the collective operation `MADRE_computeMemStats`.

Chapter 5

Examples

A simple example of a complete C/MPI program using MADRE is shown in Figure 5.1. The `Makefile` used to compile, link, and execute the program is shown in Figure 5.2. The output produced by executing the program on 4 processes is shown in Figure 5.3.

In this program, there is a `data` array on each process. This array consists of `NUMBLOCKS` blocks, each block comprising `BLOCKLENGTH` `double`s. The blocks will be redistributed using the `phBred` algorithm. (This is the standard Pinar-Hendrickson algorithm, without parking). The map simply moves each block on process i to the same positions on process $(i+1)\%n$, where n is the number of processes in `MPI_COMM_WORLD`. The redistribution function is called 3 times using this map. The complete data is printed out once at the beginning, and once after all 3 redistributions have completed.

Further examples can be found in the `examples` and `test` subdirectories of the main `madre` directory.

```
#include<stdlib.h>
#include<stdio.h>
#include "mpi.h"
#include "madre.h"
#define NUMBLOCKS 3
#define BLOCKLENGTH 2
#define NUMELEMENTS 6

double data[NUMELEMENTS];
int myRank, numProcs;

void printData() {
  int i, j, k;

  for (i = 0; i < numProcs; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (myRank == i) {
      printf("Process %d:\n", myRank);
      for (j = 0; j < NUMBLOCKS; j++) {
        printf("  Block %d: ", j);
        for (k = 0; k < BLOCKLENGTH; k++) printf("%2.2f ", data[j*BLOCKLENGTH+k]);
        printf("\n");
      }
      printf("\n");
      fflush(stdout);
    }
  }
}

int main (int argc, char *argv[]) {
  int destRanks[NUMBLOCKS], destIndices[NUMBLOCKS];
  MADRE_Object madre;
  int i, j, k;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  madre = MADRE_create(data, "phBred", NUMBLOCKS, BLOCKLENGTH,
                       MPI_DOUBLE, MPI_COMM_WORLD);
  for (i = 0; i < NUMELEMENTS; i++) data[i] = myRank*NUMELEMENTS + i;
  printData();
  for (j = 0; j < NUMBLOCKS; j++) {
    destRanks[j] = (myRank+1)%numProcs;
    destIndices[j] = j;
  }
  for (k = 0; k < 3; k++)
    MADRE_redistribute(madre, NUMBLOCKS, destRanks, destIndices);
  MADRE_destroy(madre);
  printData();
  MPI_Finalize();
}
```

Figure 5.1: `simple.c`: A simple C/MPI program using the MADRE library

```
MADRE_HOME = ../..
include $(MADRE_HOME)/local.mk
COMPILE = $(COMPILER) -c -I$(MADRE_HOME)/include
LINK = $(LINKER) -L$(MADRE_HOME)/lib -lmadre

all: run

simple.o: simple.c
        $(COMPILE) simple.c

simple: simple.o
        $(LINK) $(LIBS) -o simple simple.o

run: simple
        mpiexec -np 4 ./simple
```

Figure 5.2: `Makefile` for simple example

```
frederic:simple siegel$ make
mpiexec -np 4 ./simple
Process 0:
  Block 0: 0.00 1.00
  Block 1: 2.00 3.00
  Block 2: 4.00 5.00

Process 1:
  Block 0: 6.00 7.00
  Block 1: 8.00 9.00
  Block 2: 10.00 11.00

Process 2:
  Block 0: 12.00 13.00
  Block 1: 14.00 15.00
  Block 2: 16.00 17.00

Process 3:
  Block 0: 18.00 19.00
  Block 1: 20.00 21.00
  Block 2: 22.00 23.00

Process 0:
  Block 0: 6.00 7.00
  Block 1: 8.00 9.00
  Block 2: 10.00 11.00

Process 1:
  Block 0: 12.00 13.00
  Block 1: 14.00 15.00
  Block 2: 16.00 17.00

Process 2:
  Block 0: 18.00 19.00
  Block 1: 20.00 21.00
  Block 2: 22.00 23.00

Process 3:
  Block 0: 0.00 1.00
  Block 1: 2.00 3.00
  Block 2: 4.00 5.00

frederic:simple siegel$
```

Figure 5.3: Output from simple example

Chapter 6

Redistribution Strategies

The redistribution strategies are developing rapidly. Theh algorithms are somewhat complicated are beyond the scope of this manual. Eventually, we will add references to papers discussing each of the algorithms. For now, users are encouraged to just try different strategies to discover which work well for their particular problem.

Strategies currently available include:

1. `phBred`: the basic Pinar-Hendrickson redistribution algorithm
2. `parkBred`: the Pinar-Hendrickson redistribution algorithm with "parking"
3. `cyclicSchedulerBred`: the Cyclic Scheduler algorithm
4. `unitBred`: a "naïve" algorithm that may deadlock in some situations but is nevertheless quite effective in some situations
5. `phaseBred`: a phase-based algorithm that resorts with a cycle-basing used when memory gets tight

Appendix A

GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are

based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the

portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the

Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.