MapReduce−MPI Library Users Manual

http://www.sandia.gov/~sjplimp/mapreduce.html

Table of Contents

Table of Contents

MapReduce−MPI (MR−MPI) Library Documentation

Version info:

The MR−MPI "version" is the date when it was released, such as 1 May 2010. MR−MPI is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of MR−MPI contains all the features and bug−fixes up to and including that version date. The version date is printed to the screen every time you run a program that uses MR−MPI. It is also in the file src/version.h and in the MR−MPI directory name created when you unpack a tarball.

- If you browse the HTML doc pages on the MR−MPI WWW site, they always describe the most current version of MR−MPI.
- If you browse the HTML doc pages included in your tarball, they describe the version you have.
- The PDF file on the WWW site or in the tarball is updated about once per month. This is because we don't want it to be part of every patch.

The MapReduce−MPI (MR−MPI) library is open−source software that implements the MapReduce operation popularized by Google on top of standard MPI message passing.

The library is designed for parallel execution on distributed−memory platforms, but will also operate on a single processor. It requires no additional software to build and run, except linking with an MPI library if you wish to perform MapReduces in parallel. Similar to the original Google design, a user performs a MapReduce by writing a small program that invokes the library. The user typically provides two application−specific functions, a "map()" and a "reduce()", that are called back from the library when a MapReduce operation is executed. "Map()" and "reduce()" are serial functions, meaning they are invoked independently on individual processors on portions of your data when performing a MapReduce operation in parallel.

The MR−MPI library is written in C++ and is callable from hi−level langauges such as C++, C, Fortran. A Python wrapper is also included, so MapReduce programs can be written in Python, including map() and reduce() user callback methods. A hi−level scripting interface to the MR−MPI library, called OINK, is also included which can be used to develop and chain MapReduce algorithms together in scripts with commands that simplify data management tasks. OINK has its own manual and doc pages.

The goal of the MR−MPI library is to provide a simple and portable interface for users to create their own MapReduce programs, which can then be run on any desktop or large parallel machine using MPI. See the Background section for features and limitations of this implementation.

The distrubution includes a few examples of simple programs that illustrate the use of MR−MPI.

Source code for the library and OINK is freely available for download from the MR−MPI web site and is licensed under the modified Berkeley Software Distribution (BSD) License. This basically means they can be used by anyone for any purpose. See the LICENSE file provided with the distribution for more details.

The authors of the MR−MPI library are Steve Plimpton and Karen Devine who can be contacted via email: sjplimp,kddevin at sandia.gov.

The MR−MPI documentation is organized into the following sections. If you find errors or omissions in this manual or have suggestions for useful information to add, please send an email to the developers so we can improve the MR−MPI documentation.

Once you are familiar with MR−MPI, you may want to bookmark this page at interface_c++.html, since it gives quick access to documentation for all the MR−MPI library methods.

PDF file of the entire manual, generated by htmldoc

Background

MapReduce is the programming paradigm popularized by Google researchers Dean and Ghemawat. Their motivation was to enable analysis programs to be rapidly developed and deployed within Google to operate on the massive data sets residing on their large distributed clusters. Their paper introduced a novel way of thinking about certain kinds of large−scale computations as "map" operations followed by "reduces". The power of the paradigm is that when cast in this way, a traditionally serial algorithm now becomes two highly parallel application−specific operations (requiring no communication) sandwiched around an intermediate operation that requires parallel communication, but which can be encapsulated in a library since the operation is independent of the application.

The Google implementation of MapReduce was a C++ library with communication between networked machines via remote procedure calls. They allow for fault tolerance when large numbers of machines are used, and can use disks as out−of−core memory to process huge data sets. Thousands of MapReduce programs have since been written by Google researchers and are part of the daily compute tasks run by the company.

While I had heard about MapReduce, I didn't appreciate its power for scientific computing on a monolithic distributed−memory parallel machine, until reading a SC08 paper by Tu, et al of the D.E. Shaw company. They showed how to think about tasks such as the post−processing of simulation output as MapReduce operations. In this context it can be useful for computations that would normally be thought of as serial, such as reading in a large data set and scanning it for events of a desired kind. As before, the computation can be formulated as a highly parallel "map" followed by a "reduce". The encapsulated parallel operation in the middle requires all−to−all communication to reorgnanize the data, a familiar MPI operation.

Tu's implementation of MapReduce was in parallel Python with communication between processors via MPI, again allowing disks to be used for out−of−core operations.

This MapReduce−MPI (MR−MPI) library is a very simple and lightweight implementation of the basic MapReduce functionality, borrowing ideas from both the Dean and Sanjay and Tu, et al papers. It has the following features:

- C++ library using MPI for inter−processor communication. This allows precise control over the memory allocated during a large−scale MapReduce.
- C++ and C and Python interfaces provided. A C++ interface means that one or more MapReduce objects can be instantiated and invoked by the user's program. A C interface means that the library can also be called from C or other hi−level languages such as Fortran. A Python interface means the library can be called from a Python script, allowing you to write serial map() and reduce() functions in Python. If your machine can run Python in parallel, you can also run a parallel MapReduce in that manner.
- Small, portable. The entire library is a few thousand lines of C++ code in a handful of C++ files which can be built on any machine with a C++ compiler. For parallel operation, you link with MPI, a standard message passing library available on all distributed memory machines. For serial operation, a dummy MPI library can be substituted, which is provided. The Python wrapper can be installed on any machine with a version of Python that includes the ctypes module, typically Python 2.5 or later.
- In−core or Out−of−core operation. Each MapReduce object created allocates per−processor "pages" of memory, where the page size is determined by the user. Typical MapReduce operations can be performed using just a few such pages. If your data set (key/value pairs) fits in a single page, then the library performs its operations in−core. If your data set exceeds the page size, then processors write to temporary disk files as needed and subsequently read from them. This allows processing of data sets that are larger than will fit in the aggregate memory of all the processors.

This library also has the following limitation:

- No fault tolerance. Current MPI implementations do not enable easy detection of a dead processor. So like most MPI programs, a MapReduce operation will hang or crash if a processor goes away.

Finally, I call attention to recent work by Alexander Gray and colleagues at Georgia Tech. They show that various kinds of scientific computations such as N−body forces via multipole expansions, k−means clustering, and machine learning algorithms, can be formulated as MapReduce operations. Thus there is an expanding set of data−intense or compute−intense problems that may be amenable to solution using a MapReduce library such as this.

(Dean) J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04 conference (2004); J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, 51, p 107−113 (2008).

(Tu) T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Kelpeis, P. Maragakis, P. Miller, K. A. Stafford, D. E. Shaw, "A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Trajectories", SC08 proceedings (2008).

(Gray) A. Gray, Georgia Tech, http://www.cc.gatech.edu/~agray

What is a MapReduce?

The canonical example of a MapReduce operation, described in both the Dean and Sanjay and Tu, et al papers, is counting the frequency of words in a collection of text files. Imagine a large corpus of text comprising Gbytes or Tbytes of data. To count how often each word appears, the following algorithm would work, written in Python:

```
dict = {}
for file in sys.argv[1:]:
 text = open(file,'r').read()
 words = text.split()
 for word in words:
   if word not in dict: dict[word] = 1
   else: dict[word] += 1
unique = dict.keys()
for word in unique:
 print dict[word],word
```

Dict is a "dictionary" or associative array which is a collection of key/value pairs where the keys are unique. In this case, the key is a word and its value is the number of times it appears in any text file. The program loops over files, and splits the contents into words (separated by whitespace). For each word, it either adds it to the dictionary or increments its associated value. Finally, the resulting dictionary of unique words and their counts is printed.

The drawback of this implementation is that it is inherently serial. The files are read one by one. More importantly the dictionary data structure is updated one word at a time.

A MapReduce formulation of the same task is as follows:

```
array = []
for file in sys.argv[1:]:
 array += map(file)
newarray = collate(array)
unique = []
for entry in newarray:
 unique += reduce(entry)
for entry in unique:
 print entry[1],entry[0]
```

Array is now a linear list of key/value pairs where a key may appear many times (not a dictionary). The map() function reads a file, splits it into words, and generates a key/value pair for each word ialignn the file. The key is the word itself and the value is the integer 1. The collate() function reorganizes the (potentially very large) list of key/value pairs into a new array of key/value pairs where each unique key appears exactly once and the associated value is a concatenated list of all the values associated with the same key in the original array. Thus, a key/value pair in the new array would be ("dog",[1,1,1,1,1]) if the word "dog" appeared 5 times in the text corpus. The reduce() function takes a single key/value entry from the new array and returns a key/value pair that has the word as its key and the count as its value, ("dog",5) in this case. Finally, the elements of the unique array are printed.

As written, the MapReduce algorithm could be executed on a single processor. However, there is now evident parallelism. The map() function calls are independent of each other and can be executed on different processors simultaneously. Ditto for the reduce() function calls. In this scenario, each processor would accumulate its own local "array" and "unique" lists of key/value pairs.

Also note that if the map and reduce functions are viewed as black boxes that produce a list of key/value pairs (in the case of map) or convert a single key/value pair into a new key/value pair (in the case of reduce), then they are

the only part of the above algorithm that is application−specific. The remaining portions (the collate function, assignment of map or reduce tasks to processors, combining of the map/reduce output across processors) can be handled behind the scenes in an application−independent fashion. That is the portion of the code that is handled by the MR−MPI (or other) MapReduce library. The user only needs to provide a small driving program to call the library and serial functions for performing the desired map() and reduce() operations.

(Dean) J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04 conference (2004); J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, 51, p 107−113 (2008).

(Tu) T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Kelpeis, P. Maragakis, P. Miller, K. A. Stafford, D. E. Shaw, "A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Trajectories", SC08 proceedings (2008).

Getting Started

Once you have downloaded the MapReduce MPI (MR−MPI) library, you should have the tarball mapreduce.tar.gz on your machine. Unpack it with the following commands:

```
gunzip mapreduce.tar.gz
tar xvf mapreduce.tar
```

which should create a mapreduce directory containing the following:

- README
- LICENSE
- doc
- examples
- mpistubs
- oink
- oinkdoc
- python
- src
- user

The doc directory contains this documentation. The oink and oinkdoc directories contain the OINK scripting interface to the MR−MPI library and its separate documentation. The examples directory contains a few simple MapReduce programs which call the MR−MPI library. These are documented by a README file in that directory and are discussed below. The mpistubs directory contains a dummy MPI library which can be used to build a MapReduce program on a serial machine. The python directory contains the Python wrapper files needed to call the MR−MPI library from Python. The src directory contains the files that comprise the MR−MPI library. The user directory contains user−contributed MapReduce programs. See the README in that directory for further details.

To build the library for use by a C++ or C program, go to the src directory and type

```
make
```

You will see a list of machine names, each of which has their own Makefile.machine file in the src/MAKE directory. You can choose one of these and attempt to build the MR−MPI library by typing

```
make machine
```

If you are successful, this will produce the file "libmrmpi_machine.a" which can be linked by other programs. If not, you will need to create a src/MAKE/Makefile.machine file compatible with your platform, using one of the existing files as a template.

The only settings in a Makefile.machine file that need to be specified are those for the compiler and the MPI library on your machine. If MPI is not already installed, you can install one of several free versions that work on essentially all platforms. MPICH and OpenMPI are the most common.

Within Makefile.machine you can either specify via −I and −L switches where the MPI include and library files are found, or you can use a compiler wrapper provided with MPI, like mpiCC or mpic++, which will know where those files are.

You can also build the MR−MPI library without MPI, using the dummy MPI library provided in the mpistubs directory. In this case you can only run the library on a single processor. To do this, first build the dummy MPI library, by typing "make" from within the mpistubs directory. Again, you may need to edit mpistubs/Makefile for your machine. Then from the src directory, type "make serial" which uses the src/MAKE/Makefile.serial file.

Both a C++ and C interface are part of the MR−MPI library, so it should be usable from any hi−level language. To use the library from Python, you don't need to build a *.a file from the src directory. Instead, you build it as a dynamic library from the python directory. Instructions are given in the Python interface section.

The MapReduce programs in the examples directory can be built by typing

```
make −f Makefile.machine
```

from within the examples directory, where Makefile.machine is one of the Makefiles in the examples directory. Again, you may need to modify one of the existing ones to create a new one for your machine. Some of the example programs are provided as a C++ program, a C program, as a Python script, or as an OINK input script. Once you have built OINK, the latter can be run as, for example,

```
oink_linux <in.rmat
```

When you run one of the example MapReduce programs or your own, if you get an immediate error about the MRMPI_BIGINT data type, you will need to edit the file src/mrtype.h and re−compile the library. Mrtype.h and the error check insures that your MPI will perform operations on 8−byte unsigned integers as required by the MR−MPI library. For the MPI on most machines, this is satisfied by the MPI data type MPI_UNSIGNED_LONG_LONG. But some machines do not support the "long long" data type, and you may need a different setting for your machine and installed MPI, such as MPI_UNSIGNED_LONG.

Writing a MapReduce program

The usual way to use the MR−MPI library is to write a small main program that calls the library. In C++, your program includes two library header files and uses the MapReduce namespace:

```
#include "mapreduce.h"
#include "keyvalue.h"
using namespace MAPREDUCE_NS
```

Follow these links for info on using the library from a C program or from a Python program.

Arguments to the library's map() and reduce() methods include function pointers to serial "mymap" and "myreduce" functions in your code (named anything you wish), which will be "called back to" from the library as it performs the parallel map and reduce operations.

A typical simple MapReduce program involves these steps:

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD);   // instantiate an MR object
mr->map(nfiles,                            // parallel map
mr->collate()                                      // collate keys
mr->reduce(                              // parallel reduce
delete mr;                                         // delete the MR object
```

The main program you write may be no more complicated than this. The API for the MR−MPI library is a handful of methods which are components of a MapReduce operation. They can be combined in more complex sequences of calls than listed above. For example, one map() may be followed by several reduce() operations to massage your data in a desired way. Output of final results is typically performed as part of a myreduce() function you write which executes on one or more processors and writes to a file(s) or the screen.

The MR−MPI library operates on "keys" and "values" which are generated and manipulated by your mymap() and myreduce() functions. A key and a value are simply byte strings of arbitrary length which are logically associated with each other, and can thus represent anything you wish. For example, a key can be a text string or a particle or grid cell ID. A value can be one or more numeric values or a text string or a composite data structure that you create.

C++ Interface to the MapReduce−MPI Library

This mutiple−page section discusses how to call the MR−MPI library from a C++ program and gives a description of all its methods and variable settings. Use of the library from a C program (or other hi−level language) or from Python is discussed in other sections of the manual.

All the library methods operate on two basic data structures stored within the MapReduce object, a KeyValue object (KV) and a KeyMultiValue object (KMV). When running in parallel, these objects are stored in a distributed fashion across multiple processors.

A KV is a collection of key/value pairs. The same key may appear many times in the collection, associated with values which may or may not be the same.

A KMV is also a collection of key/value pairs. But each key in the KMV is unique, meaning it appears exactly once (see the clone() method for a possible exception). The value associated with a KMV key is a concatenated list (a multi−value) of all the values associated with the same key in the original KV.

More details about how KV and KMV objects are stored are given in the Technical Details section.

Here is an overview of how the various library methods operate on KV and KMV objects. This is useful to understand, since this determines how the various operations can be chained together in your program.

add()	KV −> KV	add pairs from one KV to another	serial	2 pages
aggregate()	KV −> KV	pairs are aggregated onto procs	parallel	7 pages
broadcast()	KV −> KV	send pairs from one proc to all procs	parallel	2 pages
clone()	KV −> KMV	each KV pair becomes a KMV pair	serial	2 pages
close()	KV	allows one MapReduce object to add KV pairs to another	serial	0 pages
collapse()	KV −> KMV	all KV pairs become one KMV pair	serial	2 pages
collate()	KV −> KMV	aggregate + convert	parallel	4+ pages
compress()	KV −> KV	calls back to user program to compress duplicate keys	serial	4+ pages
convert()	KV −> KMV	duplicate KV keys become one KMV key	serial	4+ pages
gather()	KV −> KV	collect pairs on many procs to few procs	parallel	2 pages
map()	create or add to a KV	calls back to user program to generate pairs	serial	1 page
reduce()	KMV −> KV	calls back to user program to process KMV pairs	serial	3 pages
open()			serial	

	create or add to a KV	allows one MapReduce object to add KV pairs to another		0 pages
print()	KV or KMV	print KV or KMV pairs to screen or file(s)	serial	1 page
scan()	KV or KMV	calls back to user program to process KV or KMV pairs	serial	1 page
scrunch()	KV −> KMV	gather + collapse	parallel	3 pages
sort_keys()	KV −> KV	calls back to user program to sort pairs by key	serial	5 pages
sort_values()	KV −> KV	calls back to user program to sort pairs by value	serial	5 pages
sort_multivalues()	KMV −> KMV	calls back to user program to sort multi−values within each pair	serial	4 pages
kv_stats()	KV	print stats about a KV	serial	0 pages
kmv_stats()	KMV	print stats about a KMV	serial	0 pages

Note that each MapReduce object contains a single KV or KMV object (or neither) when its method is called. (Some methods operate on 2 or more MapReduce objects.) When the method completes, the MapReduce object also contains a single KV or KMV object. Thus if a method creates a new KV or KMV object, the old one is deleted, if it existed. The KV object is also deleted if a KMV object is produced, and vice versa.

The methods flagged as "serial" perform their operation on the portion of a KV or KMV owned by an individual processor. They involve only local computation (performed simultaneously on all processors) and no parallel comuunication. The methods flagged as "parallel" involve communication between processors.

The listed page counts are the number of memory pages that method requires. See the *memsize* setting for a discussion of what memory pages are and how their size is set. The methods whose page count is listed as 4+ all perform a convert() operation internally. The minimum number of pages this requires is 4. Depending on the page size and the characteristics of the KV pairs being converted to KMV pairs, more pages can be required. See the out−of−core discussion in this section for more details.

MapReduce add() method

```
uint64_t MapReduce::add(MapReduce *mr2)
```

This calls the add() method of a MapReduce object, to add the KeyValue pairs contained in a second MapReduce object mr2, to the KeyValue object of the first MapReduce object, which is created if one does not exist. This is useful if multiple MapReduce objects have been created and populated with key/value pairs and you wish to combine them before performing further operations, such as a collate() and reduce().

For example, this sequence of calls:

```
MapReduce *mr1 = new MapReduce(MPI_COMM_WORLD);
mr1->map(ntasks,
MapReduce *mr2 = mr1->copy();
mr2->collate(NULL);
mr2->reduce(
mr1->add(mr2);
delete mr2;
mr1->collate(NULL);
mr1->reduce(
```

would generate one set of key/value pairs from the initial map() operation, then make a copy of them, which are then collated and reduced to a new set of key/value pairs. The new set of key/value pairs are added to the original set produced by the map() operation to form an augmented set of key/value pairs, which could be further processed.

Related methods: copy, map()

MapReduce aggregate() method

```
uint64_t MapReduce::aggregate(int (*myhash)(char *, int))
```

This calls the aggregate() method of a MapReduce object, which reorganizes a KeyValue object across processors into a new KeyValue object. In the original object, duplicates of the same key may be stored on many processors. In the new object, all duplicates of a key are stored by the same processor. The method returns the total number of key/value pairs in the new KeyValue object, which will be the same as the number in the original object.

A hashing function is used to assign keys to processors. Typically you will not care how this is done, in which case you can specify a NULL, i.e. mr−>aggregate(NULL), and the MR−MPI library will use its own internal hash function, which will distribute them randomly and hopefully evenly across processors.

On the other had, if you know the best way to do this for your data, then you should provide the hashing function. For example, if your keys are integer IDs for particles or grid cells, you might want to use the ID (modulo the processor count) to choose the processor it is assigned to. Ideally, you want a hash function that will distribute keys to processors in a load−balanced fashion.

In this example the user function is called myhash() and it must have the following interface:

```
int iproc = myhash(char *key, int keybytes)
```

Your function will be passed a key (byte string) and its length in bytes. Typically you want to return an integer such that 0 <= iproc < P, where P is the number of processors. But you can return any integer, since the MR−MPI library uses the result in this manner to assign the key to a processor:

```
int iproc = myhash(key,keybytes) % P;
```

Because the aggregate() method will, in general, reassign all key/value pairs to new processors, it incurs a large volume of all−to−all communication. However, this is performed concurrently, taking advantage of the large bisection bandwidth most large parallel machines provide.

The aggregate() method should load−balance key/value pairs across processors if they are initially imbalanced.

Related methods: collate()

MapReduce broadcast() method

```
uint64_t MapReduce::broadcast(int root)
```

This calls the broadcast() method of a MapReduce object, which delete the key/value pairs of a KeyValue object on all processors except root, and then broadcasts the key/value pairs owned by the root processor to all the other processors. The end result is that all processors have a copy of the key/value pairs initially owned by the root processor.

The resulting set of distributed key/value pairs will have P copies of each entry, where P = the # of processors. This will in general not be useful for further MapReduce operations, but it can be useful after a gather() before doing a final reduce() where you want to give each processor access to the entire gathered result and let it make a local copy of the datums.

This method requires parallel communication as processors send their key/value pairs to other processors.

Related methods: gather()

MapReduce clone() method

```
uint64_t MapReduce::clone()
```

This calls the clone() method of a MapReduce object, which converts a KeyValue object directly into a KeyMultiValue object. It simply turns each key in KeyValue object into a key in the new KeyMultiValue object, with the same value. The method returns the total number of key/value pairs in the KeyMultiValue object, which will be the same as the number in the KeyValue object.

This method essentially enables a KeyValue object to be passed directly to a reduce operation, which requires a KeyMultiValue object as input. Typically you would only do this if the keys in the KeyValue object are already unique, to avoid the extra overhead of an aggregate() or convert() or collate(), but this is not required. If they are not, then there will also be duplicate keys in the KeyMultiValue object.

Note that one of the map() methods allows an existing KeyValue object to be passed as input to a user mymap() function, generating a new Keyvalue object in the process. Thus there is typically no need to invoke clone() followed by reduce().

This method is an on−processor operation, requiring no communication. When run in parallel, the key/value pairs of the new KeyMultiValue object are stored on the same processor which owns the corresponding KeyValue pairs.

Related methods: collapse(), collate, convert()

MapReduce collapse() method

```
uint64_t MapReduce::collapse(char *key, int keybytes)
```

This calls the collapse() method of a MapReduce object, which collapses a KeyValue object into a KeyMultiValue object with a single new key, given as an argument with its length in bytes. The single new value in the KeyMultiValue object is a concatentated list of all the keys and values in the KeyValue object. The method returns the total number of key/value pairs in the KeyMultiValue object, which will be 1 for each processor owning pairs.

For example, if the KeyValue object contains these key/value pairs:

```
("dog",3), ("me",45), ("parallel",1)
```

then the new KeyMultiValue object will contain a single key/value pair:

```
(key,["dog",3,"me",45,"parallel",1])
```

This method can be used to collect a set of key/value pairs to use in a reduce() method so that it can all be passed to a single invocation of your myreduce() function for output. See the Technical Details section for details on how the collapse() method affects the alignment of keys and values that may eventually be passed to your myreduce() function via the reduce() method.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor collapses the key/value pairs it owns into a single key/value pair. Thus each processor will assign the same key to its new pair. See the gather() and scrunch() methods for ways to collect all key/value pairs on to one or a few processors.

Related methods: clone(), collate, convert()

MapReduce collate() method

```
uint64_t MapReduce::collate(int (*myhash)(char *, int))
```

This calls the collate() method of a MapReduce object, which aggregates a KeyValue object across processors and converts it into a KeyMultiValue object. This method is exactly the same as performing an aggregate() followed by a convert(). The method returns the total number of unique key/value pairs in the KeyMultiValue object.

The hash argument is used by the aggregate() portion of the operation and can be specified as NULL. See the aggregate() doc page for details.

Note that if your map operation does not produce duplicate keys, you do not typically need to perform a collate(). Instead you can convert a KeyValue object into a KeyMultiValue object directly via the clone() method, which requires no communication. Or you can pass it directly to another map() operation. One exception would be if your map operation produces a KeyValue object which is highly imbalanced across processors. The aggregate() portion of the operation should redistribute the key/value pairs more evenly.

This method is a parallel operation (aggregate()), followed by an on−processor operation (convert()).

Related methods: aggregate(), clone, collapse(), compress(), convert()

MapReduce compress() method

MapReduce multivalue_blocks() method

MapReduce multivalue_block() method

```
uint64_t MapReduce::compress(void (*mycompress)(char *, int, char *, int, int *, KeyValue *, void *)

uint64_t MapReduce::multivalue_blocks()

int MapReduce::multivalue_block(int iblock, char **ptr_multivalue, int **ptr_valuesizes)
```

This calls the compress() method of a MapReduce object, passing it a function pointer to a mycompress function you write. This method compresses a KeyValue object with duplicate keys into a new KeyValue object, where each key appears once (on that processor) and has a single new value. The new value is a combination of the values associated with that key in the original KeyValue object. The mycompress() function you provide generates the new value, once for each unique key (on that processor). The method returns the total number of key/value pairs in the new KeyValue object.

This method is used to compress a large set of key/value pairs produced by the map() method into a smaller set before proceeding with the rest of a MapReduce operation, e.g. with a collate() and reduce().

You can give this method a pointer (void *ptr) which will be returned to your mycompress() function. See the Technical Details section for why this can be useful. Just specify a NULL if you don't need this.

In this example the user function is called mycompress() and it must have the following interface, which is the same as that used by the reduce() method:

```
void mycompress(char *key, int keybytes, char *multivalue, int nvalues, int *valuebytes, KeyValue *k
```

A single key/multi−value (KMV) pair is passed to your function from a temporary KeyMultiValue object created by the library. That object creates a multi−value for each unique key in the KeyValue object which contains a list of the nvalues associated with that key. Note that this is only the values on this processor, not across all processors.

There are two possibilities for a KMV pair returned to your function. The first is that it fits in one page of memory allocated by the MapReduce object, which is the usual case. See the *memsize* setting for details on memory allocation.

In this case, the char *multivalue argument is a pointer to the beginning of the multi−value which contains all nvalues, packed one after the other. The int *valuebytes argument is an array which stores the length of each value in bytes. If needed, it can be used by your function to compute an offset into char *values for where each individual value begins. Your function is also passed a kv pointer to a new KeyValue object created and stored internally by the MapReduce object.

If the KMV pair does not fit in one page of memory, then the meaning of the arguments passed to your function is changed. Your function must call two additional library functions in order to retrieve a block of values that does fit in memory, and process them one block at a time.

In this case, the char *multivalue argument will be NULL and the nvalues argument will be 0. Either of these can be tested for within your function. If you know huge multi−values will not occur, then the test is not needed. The meaning of the kv and ptr arguments is the same as discussed above. However, the int *valuebytes argument is changed to be a pointer to the MapReduce object. This is to allow you to make the following two kinds of calls back to the library:

```
MapReduce *mr = (MapReduce *) valuebytes;
int nblocks;
uint64_t nvalues_total = mr->multivalue_blocks(nblocks);
for (int iblock = 0; iblock <nblocks; iblock++)
  int nv = mr->multivalue_block(iblock,ebytes);
  for (int i = 0; i  <nv; i++)
    process each value within the block of values
```

The call to multivalue_blocks() returns both the total number of values (as an unsigned 64−bit integer), and the number of blocks of values in the multi−value. Each call to multivalue_block() retrieves one block of values. The number of values in the block is returned, as nv in this case. The multivalue and valuebytes arguments are pointers to a char * and int * (i.e. a char ** and int **), which will be set to point to the block of values and their lengths respectively, so they can then be used just as the multivalue and valuebytes arguments in the myreduce() callback itself (when the values do not exceed available memory).

The call to multivalue_blocks() returns the number of blocks of values in the multi−value. Each call to multivalue_block() retrieves one block of values. The number of values in the block (nv in this case) is returned. The multivalue and valuebytes arguments are pointers to a char * and int * (i.e. a char ** and int **), which will be set to point to the block of values and their lengths respectively, so they can then be used just as the multivalue and valuebytes arguments in the mycompress() callback itself (when the values do not exceed available memory).

Note that in this example we are re−using (and thus overwriting) the original multivalue and valuebytes arguments as local variables.

Also note that your mycompress() function can call multivalue_block() as many times as it wishes and process the blocks of values multiple times or in any order, though looping through blocks in ascending order will typically give the best disk I/O performance.

Your mycompress() function should typicaly produce a single key/value pair which it registers with the MapReduce object by calling the add() method of the KeyValue object. The syntax for this call is described on the doc page for the KeyValue add() methd. For example, if the set of nvalues were integers, the compressed value might be the sum of those integers.

See the Settings and Technical Details sections for details on the byte−alignment of keys and values that are passed to your mycompress() function and on those you register with the KeyValue add() methods. Note that only the first value of a multi−value (or of each block of values) passed to your mycompress() function will be aligned to the *valuealign* setting.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor operates only on the key/value pairs it stores. Thus you are NOT compressing all values associated with a particular key across all processors, but only those currently owned by one processor.

Related methods: collate()

MapReduce convert() method

```
uint64_t MapReduce::convert()
```

This calls the convert() method of a MapReduce object, which converts a KeyValue object into a KeyMultiValue object. It does this by finding duplicate keys (stored only by this processor) and concatenating their values into a list of values which it associates with the key in the KeyMultiValue object. The method returns the total number of key/value pairs in the KeyMultiValue object, which will be the number of unique keys in the KeyValue object.

This operation creates a hash table to find duplicate keys efficiently. More details are given in the Technical Details section.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor converts only the key/value pairs it owns into key/multi−value pairs. Thus, this operation is typically performed only after the aggregate() method has collected all duplicate keys to the same processor. The collate() method performs an aggregate() followed by a convert().

Related methods: collate()

Copy a MapReduce object

```
MapReduce *MapReduce::copy()
```

This calls the copy() method of a MapReduce object, which creates a second MapReduce object which is an exact copy of the first, including all settings, and returns a pointer to the new copy.

If the original MapReduce object contained a KeyValue or KeyMultiValue object, as discussed here, then the new MapReduce object will contain a copy of it. This means that all the key/value and/or key/multivalue pairs contained in the first MapReduce object are copied into the new MapReduce object. Thus the first MapReduce object could be subsequently deleted without affecting the new MapReduce object.

This is useful if you wish to retain a copy of a set of key/value pairs before processing it further. See the add() method for how to merge the key/value pairs from two MapReduce objects into one. For example, this sequence of calls:

```
MapReduce *mr1 = new MapReduce(MPI_COMM_WORLD);
mr1->map(ntasks,
MapReduce *mr2 = mr1->copy();
mr2->collate(NULL);
mr2->reduce(
mr1->add(mr2);
delete mr2;
mr1->collate(NULL);
mr1->reduce(
```

would generate one set of key/value pairs from the initial map() operation, then make a copy of them, which are then collated and reduced to a new set of key/value pairs. The new set of key/value pairs are added to the original set produced by the map() operation to form an augmented set of key/value pairs, which could be further processed.

Related methods: create, add()

Create a MapReduce object

```
MapReduce::MapReduce(MPI_Comm comm)
MapReduce::MapReduce()
MapReduce::MapReduce(double dummy)
```

You can create a MapReduce object in any of the three ways shown, as well as via the copy() method. The three creation methods differ slightly in how MPI is initialized and finalized.

In the first case, you pass an MPI communicator to the constructor. This means your program should initialize (and finalize) MPI, which creates the MPI_COMM_WORLD communicator (all the processors you are running on). Normally this is what you pass to the MapReduce constructor, but you can pass a communicator for a subset of your processors if desired. You can also instantiate multiple MapReduce objects, giving them each a communicator for all the processors or communicators for a subset of processors.

The second case can be used if your program does not use MPI at all. The library will initialize MPI if it has not already been initialized. It will not finalize MPI, but this should be fine. Worst case, your program may complain when it exits if MPI has not been finalized.

The third case is the same as the second except that the library will finalize MPI when the last instance of a MapReduce object is destructed. Note that this means your program cannot delete all its MapReduce objects in a early phase of the program and then instantiate more MapReduce objects later. This limitation is why the second case is provided. The third case is invoked by passing a double to the constructor. If this is done for any instantiated MapReduce object, then the library will finalize MPI. The value of the double doesn't matter as it isn't used. The use of a double is simply to make it different than the first case, since MPI_Comm is often implemented by MPI libraries as a type cast to an integer.

As examples, any of these lines of code will create a MapReduce object, where "mr" is either a pointer to the object or the object itself:

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
MapReduce *mr = new MapReduce();
MapReduce *mr = new MapReduce(0.0);
MapReduce mr(MPI_COMM_WORLD);
MapReduce mr();
MapReduce mr;
MapReduce mr(0.0);
```

Related methods: destroy, copy()

Destroy a MapReduce object

```
MapReduce::~MapReduce()
```

This destroys a previously created MapReduce object, freeing all the memory it allocated internally to store keys and values.

If you created the MapReduce object in this manner:

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
```

then you should destroy it with

```
delete mr
```

If you created the MapReduce object in this manner:

```
MapReduce mr(MPI_COMM_WORLD);
```

then it will be destroyed automatically when the "mr" variable goes out of scope.

Related methods: create

MapReduce gather() method

```
uint64_t MapReduce::gather(int nprocs)
```

This calls the gather() method of a MapReduce object, which collects the key/value pairs of a KeyValue object spread across all processors to form a new KeyValue object on a subset (nprocs) of processors. Nprocs can be 1 or any number smaller than P, the total number of processors. The gathering is done to the lowest ID processors, from 0 to nprocs−1. Processors with ID >= nprocs end up with an empty KeyValue object containing no key/value pairs. The method returns the total number of key/value pairs in the new KeyValue object, which will be the same as in the original KeyValue object.

This method can be used to collect the results of a reduce() to a single processor for output. See the collapse() and scrunch() methods for related ways to collect key/value pairs for output. A gather() may also be useful before a reduce() if the number of unique key/value pairs is small enough that you wish to perform the reduce tasks on fewer processors.

This method requires parallel point−to−point communication as processors send their key/value pairs to other processors.

Related methods: scrunch(), broadcast()

KeyValue add() method

```
void KeyValue::add(char *key, int keybytes, char *value, int valuebytes)
void KeyValue::add(int n, char *keys, int keybytes, char *values, int valuebytes)
void KeyValue::add(int n, char *keys, int *keybytes, char *values, int *valuebytes)
```

The methods are called by the mymap(), mycompress(), and myreduce() functions in your program to register key/value pairs with the KeyValue object stored by the MapReduce object whose map(), compress(), or reduce() method was invoked. The first version registers a single key/value pair. The second version registers N key/value pairs, where the keys are all the same length and the values are all the same length. The third version registers a set of N key/value pairs where the length of each key and of each value is specified.

As explained here, from the perspective of the MR−MPI library, keys and values are variable−length byte strings. To register such strings, you must specify their length in bytes. This is done via the keybytes and valuebytes arguments, either as a single length or as a vectors of lengths. Note that if your key or value is a text string, it should typically include a trailing "0" to terminate the string.

See the Settings and Technical Details sections for details on the byte−alignment of keys and values you register with these add methods.

MapReduce map() method

```
Variant 1:
uint64_t MapReduce::map(int nmap, void (*mymap)(int, KeyValue *, void *), void *ptr)
uint64_t MapReduce::map(int nmap, void (*mymap)(int, KeyValue *, void *), void *ptr, int addflag)

Variant 2:
uint64_t MapReduce::map(int nstr, char **strings, int self, int recurse, int readfile, void (*mymap)
uint64_t MapReduce::map(int nstr, char **strings, int self, int recurse, int readfile, void (*mymap)

Variant 3:
uint64_t MapReduce::map(int nmap, int nstr, char **strings, int recurse, int readfile, char sepchar,
uint64_t MapReduce::map(int nmap, int nstr, char **strings, int recurse, int readfile, char sepchar,

Variant 4:
uint64_t MapReduce::map(int nmap, int nstr, char **strings, int recurse, int readfile, char *sepstr,
uint64_t MapReduce::map(int nmap, int nstr, char **strings, int recurse, int readfile, char *sepstr,

Variant 5:
uint64_t MapReduce::map(MapReduce *mr2, void (*mymap)(uint64_t, char *, int, char *, int, KeyValue *
uint64_t MapReduce::map(MapReduce *mr2, void (*mymap)(uint64_t, char *, int, char *, int, KeyValue *
```

This calls the map() method of a MapReduce object. A function pointer to a mapping function you write is specified as an argument. This method either creates a new KeyValue object to store all the key/value pairs generated by your mymap function, or adds them to an existing KeyValue object. The method returns the total number of key/value pairs in the KeyValue object.

There are several variants of the map() methods to allow for different ways to process input data. This also induces variants of the callback mymap() function.

For the first set of variants (with or without addflag) you simply specify a total number of map tasks *nmap* to perform across all processors. The index of a map task is passed back to your mymap() function. The MapReduce library assigns map tasks to processors; see more details below.

For the second set of variants, you specify *nstr* and *strings* which are file and/or directory names. Using these strings, a list of filenames is generated. Each filename in the list is passed back to your mymap() function which can open the file and process it.

If *self* is 0, then only processor 0 generates the list of filenames, and the MapReduce library assigns files to processors; see more details below. If *self* is 1, then each processor generates its own list of filenames and those files are assigned to that processor. Note that in the *self* = 0 case, it is assumed that every processor can read any file that is assigned to it. Also note, that with *self* = 1 you can assign files to a processor that reside on a disk local to a processor, or with a parallel disk system you can pass different strings to different processors so that each processor reads from different set of files/directories.

The list of filenames is generated in the following manner. Each of the *strings* is checked for whether it is a file or directory. If it is a file, it is added to the list of files. If it is a directory, the directory is opened and all the files in it are added to the list of files. If the *recurse* flag is set to 1, then if sub−directories are found in the directory, they are opened and the files in them are also added to the list of files (and so forth, recursively).

The *readfile* setting adds one additional wrinkle. If *readfile* is 1, then instead of adding each filename to the list, each file is opened, and filenames are read from that file and added to the list. In this mode, each file should contain contain one filename per line. Blank lines are not allowed. Leading and trailing whitespace around each

filename is OK.

The number of files that are generated and processed can be accessed after the map() method is invoked, but the variable mapfilecount, e.g.

```
MapReduce *mr = new MapReduce();
mr->map(nstr,strings,1,0,1,mymap,NULL);
int ntotalfiles = mr->mapfilecount;
```

The third set of variants allows large file(s) to be broken into chunks and one or more sections to be passed back to your mymap() function as a string so it can process it. *Nmap* is the number of chunks to generate from all the files in aggregate (not *nmap* chunks per file). As with the previous variant, you also specify *nstr*, *strings*, *recurse*, and *readfile*. This generates a list of filenames, the same as in the previous variant. The only difference is that no *self* setting is allowed, because only processor 0 does this. The specified *nmap* should be >= the number of files in the generated list; it is reset to the number of files if that is not the case.

For the third set of variants you specify a separation character *sepchar*. For the fourth set of variants, you specify a separation string *sepstr*. The files in the generated list of files are split into *nmap* chunks with roughly equal numbers of bytes in each chunk. Think of all the files concatenated together and then split into *nmap* chunks. For each call to your mymap() function, a chunk is read from a particular file, and passed to your function as a string, so your code does not read the file. See details below about the splitting methodology and the delta input parameter.

For the fifth set of variants, you specify an existing MapReduce object mr2 with key/value pairs, which can either be this MapReduce object or another one. The key/value pairs from mr2 are passed back to your mymap() function, one key/value at a time, allowing you to generate new key/value pairs from an existing set.

You can give any of the map() methods a pointer (void *ptr) which will be returned to your mymap() function. See the Technical Details section for why this can be useful. Just specify a NULL if you don't need this.

The meaning of the final *addflag* argument is as follows.

For all but the last variant, if *addflag* is omitted or is specified as 0, then map() will create a new KeyValue object, deleting any existing KeyValue object. If addflag is non−zero, then KV pairs generated by your mymap() function are added to an existing KeyValue object, which is created if needed.

For the last variant, if the source of KeyValue pairs (mr2) is different than the MapReduce object mr, then the KV pairs in mr2 are not altered or deleted, regardless of the addflag setting. If addflag is 0, then the KeyValue object in mr is deleted, and newly generated KV pairs are added to a new KeyValue object. If addflag is 1, then newly generated KV pairs are added to the existing KeyValue object in mr.

For the last variant, if the source of KeyValue pairs (mr2) is the same as MapReduce object mr, there are two possibilities. If addflag is 1, then newly generated KV pairs are added to the existing KeyValue object. If addflag is 0, then the existing KeyValue object is effectively replaced by the newly generated KV pairs. Note that the addflag=1 option requires the KeyValue object to first be copied. If your mymap() function will not generate any new KV pairs, then it is more efficient to use the scan() method, which simply allows you to iterated over the existing KV pairs.

In these examples the user function is called mymap() and it has one of four interfaces depending on which variant of the map() method is invoked:

```
void mymap(int itask, KeyValue *kv, void *ptr)
void mymap(int itask, char *file, KeyValue *kv, void *ptr)
```

```
void mymap(int itask, char *str, int size, KeyValue *kv, void *ptr)
void mymap(uint64_t itask, char *key, int keybytes, char *value, int valuebytes, KeyValue *kv, void
```

In all cases, the final 2 arguments passed to your function are a pointer to a KeyValue object (kv) stored internally by the MapReduce object, and the original pointer you specified as an argument to the map() method, as void *ptr.

In the first mymap() variant, itask is passed to your function with a value 0 <= itask < *nmap*, where *nmap* was specified in the map() call. For example, you could use itask to select a file from a list stored by your application. Your mymap() function could open and read the file or perform some other operation.

In the second mymap() variant, itask will have a value 0 <= itask < nfiles, where nfiles is either the number of filenames in the list of files that was generated. Your function is also passed a single filename, which it will presumably open and read.

In the third mymap() variant, itask will have a value from 0 <= itask < *nmap*, where *nmap* was specified in the map() call and is the number of file segments generated. It is also passed a string of bytes (str) of length size read from one of the files. Size includes a trailing '\0' that is appended to the string.

For map() methods that take files and a separation criterion as arguments, you must specify *nmap* >= nfiles, so that there is one or more map tasks per file. For files that are split into multiple chunks, the split is done at occurrences of the separation character or string. You specify a delta of how many extra bytes to read with each chunk that will guarantee the splitting character or string is found within that many bytes. For example if the files are lines of text, you could choose a newline character '\n' as the sepchar, and a delta of 80 (if the longest line in your files is 80 characters). If the files are snapshots of simulation data where each snapshot is 1000 lines (no more than 80 characters per line), you could choose the first line of each snapshot (e.g. "Snapshot") as the sepstr, and a delta of 80000. Note that if the separation character or string is not found within delta bytes, an error will be generated. Also note that there is no harm in choosing a large delta so long as it is not larger than the chunk size for a particular file.

If the separation criterion is a character (sepchar), the chunk of bytes passed to your mymap() function will start with the character after a sepchar, and will end with a sepchar (followed by a '\0'). If the separation criterion is a string (sepstr), the chunk of bytes passed to your mymap() function will start with sepstr, and will end with the character immediately preceeding a sepstr (followed by a '\0'). Note that this means your mymap() function will be passed different byte strings if you specify sepchar = 'A' vs sepstr = "A".

In the fourth mymap() variant, itask will have a value from 0 <= itask < nkey, where nkey is a unsigned 64−bit int and is the number of key/value pairs in the specified MapReduce object. Key and value are the byte strings for a single key/value pair and are of length keybytes and valuebytes respectively.

The MapReduce library assigns map tasks to processors. Options for how it does this can be controlled by MapReduce settings. Basically, *nmap*/P tasks are assigned to each processor, where P is the number of processors in the MPI communicator you instantiated the MapReduce object with.

Typically, your mymap() function will produce key/value pairs which it registers with the MapReduce object by calling the add() method of the KeyValue object. The syntax for registration is described on the doc page of the KeyValue add() method.

See the Settings and Technical Details sections for details on the byte−alignment of keys and values you register with the KeyValue add() methods or that are passed to your mymap() function.

Aside from the assignment of tasks to processors, this method is really an on−processor operation, requiring no communication. When run in parallel, each processor generates key/value pairs and stores them, independently of

other processors.

Related methods: Keyvalue add(), reduce()

MapReduce open() method

MapReduce close() method

```
void MapReduce::open()
void MapReduce::open(int addflag)
uint64_t MapReduce::close()
```

These call the open() and close() methods of a MapReduce object. This is only necessary when you will be performing a map() or reduce() that generates key/value pairs, and you wish to add pairs not only to the MapReduce object which is invoking the map() and reduce(), but also to one or more other MapReduce objects. In order to do this, you need to invoke the open() and close() methods on the other MapReduce object(s), so that they can accumulate new key/value pairs properly. The close() method returns the total number of key/value pairs in the KeyValue object.

Here is an example of how this is done:

```
MapReduce *mr = new MapReduce()
MapReduce *mr2 = new MapReduce()
mr2->open()
mr->map(1000,mymap,mr2->kv);
mr2->close()

void mymap(int itask, KeyValue *kv, void *ptr) {
  ...
  kv->add(key1,key1bytes,value1,value1bytes);
  KeyValue *kv2 = (KeyValue *) ptr;
  kv2->add(key2,key2bytes,value2,value2bytes);
}
```

The mymap() function is being called from the "mr" MapReduce object, and can add key/value pairs to "mr" in the usual way, via the kv->add() function call. But it can also add key/value pairs to the "mr2" MapReduce object via the kv2->add() function call. To do this, 3 things were necessary:

- call the open() method of mr2 before the map() was invoked
- pass a pointer to the map() which allows mymap() to retrieve the pointer to mr2's internal KeyValue object
- call the close() method of mr2 after the map() was invoked

The second bullet point was accomplishsed by passing mr2->kv directly to the map() method, but other variations are possible. For example, a pointer to a data structure could be passed, which contains pointers to several other MapReduce objects. In this case, the open() and close() methods for each of the other MapReduce objects would need to be called appropriately before and after the map() method, assuming they would each have key/value pairs added to them by the mymap() function.

You can call open() and close() as many times as needed, but note calls to open() and close() should always come in pairs. You should not call close() when an open() has not been invoked. And you should not open() a second time without calling close() first.

Related methods: map(), reduce

MapReduce print() method

```
void MapReduce::print(int proc, int nstride, int kflag, int vflag)
void MapReduce::print(char *file, int fflag, int proc, int nstride, int kflag, int vflag)
```

This calls the print() method of a MapReduce object. The first variant prints out the KeyValue or KeyMultiValue pairs to the screen. The second variant prints to one or more files. This can be useful for debugging purposes.

If *proc* < 0, then all processors print their information, one processor at a time. If *proc* >= 0, then only the specified *proc* prints its information.

For printing to files, if *fflag* = 0, then all processors print in succession to the names file. If *fflag* = 1, then each processor writes to file.P, where P = 0 to Nprocs−1.

Each processor prints every Nth of its pairs, where N = *nstride*. Thus if *nstride* = 1, all pairs are printed.

The *kflag* and *vflag* setting control the format of the printed output. Only a limited set of choices is available. If these choices do not match the format of your keys and values, you will need to pass your data to map() or reduce() function you write yourself to print them. These can be invoked by the map() or reduce() methods.

These are the recognized *kflag* and *vflag* settings:

- flag = 0 for NULL
- flag = 1 for 32−bit positive integer (int)
- flag = 2 for 64−bit unsigned integer (uint64_t)
- flag = 3 for 32−bit floating point value (float)
- flag = 4 for 64−bit floating point value (double)
- flag = 5 for a NULL−terminated string
- flag = 6 for a pair of 32−bit positive integers (int int)
- flag = 7 for a pair of 64−bit unsigned integers (uint64_t uint64_t)

For example, using *kflag* = 1 and *vflag* = 7, would be appropriate for keys that are 32−bit integers, and values that are a pair of 64−bit integers.

For KeyMultiValue pairs, the *vflag* setting is used to format each output value in the multi−value.

Related methods: collate()

MapReduce reduce() method

MapReduce multivalue_blocks() method

MapReduce multivalue_block() method

MapReduce multivalue_block_select() method

```
uint64_t MapReduce::reduce(void (*myreduce)(char *, int, char *, int, int *, KeyValue *, void *), vo

uint64_t MapReduce::multivalue_blocks()

int MapReduce::multivalue_block(int iblock, char **ptr_multivalue, int **ptr_valuesizes)

void MapReduce::multivalue_block_select(int which)
```

This calls the reduce() method of a MapReduce object, passing it a function pointer to a myreduce function you write. It operates on a KeyMultiValue object, calling your myreduce function once for each unique key/multi−value (KMV) pair owned by that processor. A new KeyValue object is created which stores all the key/value pairs generated by your myreduce() function. The method returns the total number of new key/value pairs stored by all processors.

You can give this method a pointer (void *ptr) which will be returned to your myreduce() function. See the Technical Details section for why this can be useful. Just specify a NULL if you don't need this.

In this example the user function is called myreduce() and it must have the following interface, which is the same as that used by the compress() method:

```
void myreduce(char *key, int keybytes, char *multivalue, int nvalues, int *valuebytes, KeyValue *kv,
```

A single KMV pair is passed to your function from the KeyMultiValue object stored by the MapReduce object. The key is typically unique to this reduce task and the multi−value is a list of the nvalues associated with that key in the KeyMultiValue object.

There are two possibilities for a KMV pair returned to your function. The first is that it fits in one page of memory allocated by the MapReduce object, which is the usual case. See the *memsize* setting for details on memory allocation.

In this case, the char *multivalue argument is a pointer to the beginning of the multi−value which contains all nvalues, packed one after the other. The int *valuebytes argument is an array which stores the length of each value in bytes. If needed, it can be used by your function to compute an offset into char *values for where each individual value begins. Your function is also passed a kv pointer to a new KeyValue object created and stored internally by the MapReduce object.

If the KMV pair does not fit in one page of memory, then the meaning of the arguments passed to your function is changed. Your function must call two additional library functions in order to retrieve a block of values that does fit in memory, and process them one block at a time.

In this case, the char *multivalue argument will be NULL and the nvalues argument will be 0. Either of these can be tested for within your function. If you know that no KMV pair will overflow one page of memory, then the test

is not needed. The meaning of the kv and ptr arguments is the same as discussed above. However, the int *valuebytes argument is changed to be a pointer to the MapReduce object. This is to allow you to make the following two kinds of calls back to the library:

```
MapReduce *mr = (MapReduce *) valuebytes;
int nblocks;
uint64_t nvalues_total = mr->multivalue_blocks(nblocks);
for (int iblock = 0; iblock <nblocks; iblock++) {
  int nv = mr->multivalue_block(iblock,ebytes);
  for (int i = 0; i  <nv; i++) {
    process each value within the block of values
  }
}
```

The call to multivalue_blocks() returns both the total number of values (as an unsigned 64−bit integer), and the number of blocks of values in the multi−value. Each call to multivalue_block() retrieves one block of values. The number of values in the block is returned, as nv in this case. The multivalue and valuebytes arguments are pointers to a char * and int * (i.e. a char ** and int **), which will be set to point to the block of values and their lengths respectively, so they can then be used just as the multivalue and valuebytes arguments in the myreduce() callback itself (when the values do not exceed available memory).

Note that in this example we are re−using (and thus overwriting) the original multivalue and valuebytes arguments as local variables.

Also note that your myreduce() function can call multivalue_block() as many times as it wishes and process the blocks of values multiple times or in any order, though looping through blocks in ascending order will typically give the best disk I/O performance.

If you need to load and process two blocks of values simultaneously (e.g. in a double loop), then the multivalue_block_select() function can be called with which = 1 or 2 to specify a page of memory to read a block of values into. This should be set just before the call to multivalue_block(), to insure one block of values is not overwritten by reading a second block.

Your myreduce() function can produce key/value pairs (though this is not required) which it registers with the MapReduce object by calling the add() method of the KeyValue object. The syntax for registration is described on the doc page of the KeyValue add() method. Alternatively, your myreduce() function can write information to an output file.

See the Settings and Technical Details sections for details on the byte−alignment of keys and values that are passed to your myreduce() function and on those you register with the KeyValue add() methods. Note that only the first value of a multi−value (or of each block of values) passed to your myreduce() function will be aligned to the *valuealign* setting.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor performs a myreduce() on each of the key/value pairs it owns and stores any new key/value pairs it generates.

Related methods: Keyvalue add(), map()

MapReduce scan() method

```
uint64_t MapReduce::scan(void (*myscan)(char *, int, char *, int, void *), void *ptr)
uint64_t MapReduce::scan(void (*myscan)(char *, int, char *, int, int *, void *), void *ptr)
```

This calls the scan() method of a MapReduce object, passing it a function pointer to a myscan function you write. Depending on whether you pass it a function for processing key/value (KV) or key/multi−value (KMV) pairs, it will call your myscan function once for each KV or KMV pair owned by that processor. The KV or KMV pairs stored by the MapReduce object are not altered by this operation, nor are you allowed to emit any new KV pairs. Thus your myscan function is not passed a KV pointer. This is a useful way to simply scan over the existing KV or KMV pairs and process them in some way, e.g. for debugging or statistics generation or output.

Contrast this method with the map() method variant that takes a MapReduce object as input and returns KV pairs to your mymap() function. If that MapReduce object is the same as the caller and if the addflag parameter is set to 0, your existing KV pairs are deleted by this action. If the addflag parameter is set to 1, and you emit no new KV pairs, then your existing KV pairs are unchanged. However a copy of all your KV pairs is first performed to insure this outcome. The scan() method avoids this copy.

Also contrast this method with the reduce() method which returns KMV pairs to your myreduce() function. Your existing KMV pairs are deleted by this action, and replaced with new KV pairs which you generate.

You can give this method a pointer (void *ptr) which will be returned to your myscan() function. See the Technical Details section for why this can be useful. Just specify a NULL if you don't need this.

In this example the user function is called myscan() and it must have one of the two following interfaces, depending on whether the MapReduce object currently contains KV or KMV pairs:

```
void myscan(char *key, int keybytes, char *value, int valuebytes, void *ptr)
void myscan(char *key, int keybytes, char *multivalue, int nvalues, int *valuebytes, void *ptr)
```

Either a single KV or KMV pair is passed to your function from the KeyValue or KeyMultiValue object stored by the MapReduce object. In the case of KMV pairs, the key is typically unique to this scan task and the multi−value is a list of the nvalues associated with that key in the KeyMultiValue object.

There are two possibilities for a KMV pair returned to your function. The first is that it fits in one page of memory allocated by the MapReduce object, which is the usual case. Or it does not, in which case the meaning of the arguments passed to your function is changed. This behavior is identical to that of the reduce() method, including the meaning of the arguments returned to your myscan() function, and the 3 additional library functions you can call to retrieve additional values in the KMV pair, namely:

```
uint64_t MapReduce::multivalue_blocks()
int MapReduce::multivalue_block(int iblock, char **ptr_multivalue, int **ptr_valuesizes)
void MapReduce::multivalue_block_select(int which)
```

See the reduce() method doc page for details.

See the Settings and Technical Details sections for details on the byte−alignment of keys and values that are passed to your myscan() function. Note that only the first value of a multi−value (or of each block of values) passed to your myscan() function will be aligned to the *valuealign* setting.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor

performs a myscan() on each of the KV or KMV pairs it owns.

Related methods: map(), reduce()

MapReduce scrunch() method

```
uint64_t MapReduce::scrunch(int nprocs, char *key, int keybytes)
```

This calls the scrunch() method of a MapReduce object, which gathers a KeyValue object onto nprocs and collapses it into a KeyMultiValue object. This method is exactly the same as performing a gather() followed by a collapse(). The method returns the total number of key/value pairs in the KeyMultiValue object which should be one for each of the nprocs.

The nprocs argument is used by the gather() portion of the operation. See the gather() doc page for details. The key and keybytes arguments are used by the collapse() portion of the operation. See the collapse() doc page for details.

Note that if nprocs > 1, then the same key will be assigned to the collapsed key/multi−value pairs on each processor.

This method can be used to collect a set of key/value pairs to use in a reduce() method so that it can all be passed to a single invocation of your myreduce() function for output.

This method is a parallel operation (gather()), followed by an on−processor operation (collapse()).

Related methods: collapse(), gather()

Settings and defaults

These are internal library variables that can be set by your program:

- mapstyle = 0 (chunk) or 1 (stride) or 2 (master/slave)
- all2all = 0 (irregular communication) or 1 (use MPI_Alltoallv)
- verbosity = 0 (none) or 1 (summary) or 2 (histogrammed)
- timer = 0 (none) or 1 (summary) or 2 (histogrammed)
- memsize = N = number of Mbytes per page of memory
- minpage = N = # of pages to pre−allocate per processor
- maxpage = N = max # of pages allocatable per processor
- freepage = 1 if memory pages are freed in between operations, 0 if held
- outofcore = 1 if even 1−page data sets are forced to disk, 0 if not, −1 if cannot write to disk
- zeropage = 1 if zero out every allocated page, 0 if not
- keyalign = N = byte−alignment of keys
- valuealign = N = byte−alignment of values
- fpath = string

All the settings except *fpath* are set in the following manner from C++:

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
mr->verbosity = 1;
```

Because *fpath* takes a string argument, it is set with the following function:

```
mr->set_fpath(char *string);
```

See the C interface and Python interface doc pages for how to set the various settings from C and Python.

As documented below, some of these settings can be changed at any time. Others only have effect if they are changed before the MapReduce object begins to operate on KeyValue and KeyMultiValue objects.

The *mapstyle* setting determines how the N map tasks are assigned to the P processors by the map() method.

A value of 0 means split the tasks into "chunks" so that processor 0 is given tasks from 0 to N/P, proc 1 is given tasks from N/P to 2N/P, etc. Proc P−1 is given tasks from N − N/P to N.

A value of 1 means "strided" assignment, so proc 0 is given tasks 0,P,2P,etc and proc 1 is given tasks 1,P+1,2P+1,etc and so forth.

A value of 2 uses a "master/slave" paradigm for assigning tasks. Proc 0 becomes the "master"; the remaining processors are "slaves". Each is given an initial task by the master and reports back when it is finished. It is then assigned the next available task which continues until all tasks are completed. This is a good choice if the CPU time required by various mapping tasks varies greatly, since it will tend to load−balance the work across processors. Note however that proc 0 performs no mapping tasks.

This setting can be changed at any time.

The default value for *mapstyle* is 0.

The *all2all* setting determines how point−to−point communication is done when the aggregate() method is invoked, either by itself or as part of a collate().

A value of 0 means custom routines for irregular communication are used. A value of 1 means the MPI_Alltoallv() function from the MPI library is used. The results should be identical. Which is faster depends on the MPI library implementation of the MPI standard on a particular machine.

This setting can be changed at any time.

The default value for *all2all* is 1.

The *verbosity* setting determines how much diagnostic output each library call prints to the screen. A value of 0 means "none". A value of 1 means a "summary" of the results across all processors is printed, typically a count of total key/value pairs and the memory required to store them. A value of 2 prints the summary results and also a "histogram" of these quantities by processor, so that you can detect memory usage imbalance.

This setting can be changed at any time.

The default value for *verbosity* is 0.

The *timer* setting prints out timing information for each call to the library. A value of 0 means "none". A value of 1 invokes an MPI_Barrier() at the beginning and end of the operation and prints the elapsed time, which will be the same on all processors. A value of 2 invokes no MPI_Barrier() calls and prints a one−line summary of timing results across all processors and also a "histogram" of the time on each processor, so that you can detect computational imbalance.

This setting can be changed at any time.

The default value for *timer* is 0.

The *memsize* setting determines the page size (in Mbytes) of each page of memory allocated by the MapReduce object to perform its operations. The number of pages required by different methods varies; 1 to 7 is typical. The *freepage* setting (see below) determines whether pages are freed or not between operations, once allocated. See this section for a summary of memory page requirements.

The minimum allowed value for the *memsize* setting is 1, meaning 1 Mb pages.

IMPORTANT NOTE: The maximum value is unlimited, but you should insure the total memory consumed by all pages allocated by all the MapReduce objects you create, does not exceed the physical memory available (which may be shared by several processors if running on a multi−core node). If you do this, then many systems will allocate virtual memory, which will typically cause MR−MPI library operations to run very slowly and thrash the disk.

If the data owned by a processor in its collection of KeyValue or KeyMultiValue pairs fits within one page, then no disk I/O is performed; the MR−MPI library runs in−core. If data exceeds the page size, then it is written to temporary disk files and read back in for subsequent operations; the MR−MPI library runs out−of−core. See this section for more discussion of out−of−core operations. These files are created on a per−processor basis and are deleted when no longer needed. Thus if you delete all MapReduce objects that you have instantiated, no such files should exist at the end of the user program. If you should need to clean them up yourselves (e.g. your program crashes), see the discussion of the *fpath* setting which describes how they are named and where they reside.

If you set *memsize* small, then processing a large data set will induce many reads and writes to disk. If you make it large, then the reads and writes will happen in large chunks, which generally yields better I/O performance. However, past a few MBytes in size, there may be little gain in I/O performance.

This setting can only be changed before the first KeyValue or KeyMultiValue object is created by the MapReduce object. If changed after that, it will have no effect.

The default value for *memsize* is 64, meaning 64 Mbyte pages.

The default value can be changed by a compiler setting when the MR−MPI library is built. Using this flag for the compilation of the src/mapreduce.cpp file:

```
−DMRMPI_MEMSIZE=n
```

where n = 16, for example, will build the library with the default set to 16 Mbyte pages, instead of 64.

The *minpage* setting determines how many memory pages each processor pre−allocates as a block of contiguous memory when the MapReduce object performs its first operation. *Minpage* can be set to a number >= 0.

Note that if the *freepage* setting is 1 then memory pages will be freed after each MapReduce operation. This will include the initial *minpage* block of pages if none of them are in use.

This setting can only be changed before the first KeyValue or KeyMultiValue object is created by the MapReduce object. If changed after that, it will have no effect.

The default value for *minpage* is 0.

The *maxpage* setting determines the maximum number of pages a processor can ever allocate when performing MapReduce operations. Normally this will be no more than 7; see the discussion in this section for more details. *Maxpage* can be set to a number >= 0. A value of 0 means there is no limit; new pages are allocated whenever they are needed.

This setting can be changed at any time, though previously−allocated pages are not deleted if *maxpage* is set to a smaller number.

The default value for *maxpage* is 0.

The *freepage* setting determines whether or not the MapReduce ojbect frees unused memory pages after each operation is completed. If *freepage* is set to 0, then once allocated, pages are never deallocated until the MapReduce object itself is deleted. In this case pages are reused by successive operations performed by the library. If *freepage* is set to 1, then after each operation, pages used by the operation are freed, and then reallocated (as needed) by the next operation.

The default *freepage* setting of 1 is useful to limit memory use, particularly if your code uses several MapReduce objects or you are running in parallel on a multi−core node where all the cores share the same physical memory. If memory is not an issue, setting *freepage* to 0 may be somewhat faster, since memory pages will not be repeatedly allocated and freed. See the *zeropage* setting for an additional source of overhead when pages are repeatedly freed and allocated.

If the *outofcore* setting is 1, then setting *freepage* to 1 means that all memory pages will be released after each MapReduce operation. If *outofcore* is set to 0, and data fits in a single page, then the MapReduce object will always hold onto a single page of memory for that data even if *freepage* is set to 1.

This setting can be changed at any time.

The default value for *freepage* is 1.

The *outofcore* setting determines whether data that could fit in a single page of memory, within a KeyValue or KeyMultiValue object, will still be written to disk. If the data does not fit in a single page, it is always written to disk. If *outofcore* is 1, then disk files will be written. If *outofcore* is 0, then disk files are not written if not needed. If *outofcore* is −1, then disk files cannot be created and an error will result if they are needed. The latter setting is a way to insure that your data set fits in memory.

Note that if the *freepage* setting and the *outofcore* setting are both 1, then all memory pages will be released after each MapReduce operation. This can be useful to insure if your application uses many MapReduce objects and wants to limit its memory use.

This setting can be changed at any time.

The default value for *outofcore* is 0.

The *zeropage* setting determines whether newly allocated pages are filled with 0 bytes when allocated by the MapReduce object. Note that this does not apply to reused pages that were not freed. A setting of 1 means zero each page. A setting of 0 leaves them uninitialized.

Normally it should not be necessary to zero out allocated memory, and it only consumes time, especially if large pages are being used and are freed and allocated often (e.g. with *freepage* set to 1). But it can be useful when debugging with memory checkers, which may flag certain bytes within pages as uninitialized, even when this doesn't matter. This is because the byte−alignment rules for keys and values (discussed below) can skip over bytes in the page when data is written to the page.

This setting can be changed at any time.

The default value for *zeropage* is 0.

The *keyalign* and *valuealign* settings determine the byte alignment of keys and values generated by the user program when they are stored inside the library and passed back to the user program. A setting of N means N−byte alignment. N must always be a power of two.

As explained in this section, keys and values are variable−length strings of bytes. The MR−MPI library knows nothing of their contents and simply treats them as contiguous chunks of bytes. This section explains why it may be important to insure proper alignment of numeric data such as integers and floating point values.

Because keys are stored following integer lengths, keys are always at least 4−byte aligned. A larger alignment value can be specified if desired.

Because they follow keys, which may be of arbitrary length (e.g. a string), values can be 1−byte aligned. Note that if all keys are integers, then values will also be 4−byte aligned. A larger alignment value can be specified if desired.

When a multi−value is returned to the user program, e.g. by the callback of a reduce() method, only the first value in the multi−value is aligned to the *valuealign* setting. Subsequent values are packed one after the other. If all values are the same data−type, e.g. integers, then they will all have the same alignment. However, if the values are mixed data types (e.g. strings and integers), then you may need to insure each value is aligned properly before using it in your myreduce() function. See the Technical Details for more discussion of data alignment.

These settings can only be changed before the first KeyValue or KeyMultiValue object is created by the MapReduce object. If changed after that, they will have no effect.

The default value for *keyalign* and *valuealign* is 4, meaning 4−byte alignment of keys and values.

The *fpath* setting determines the pathname for all disk files created by the MR−MPI library when it runs in out−of−core mode. Note that it is not a pathname for user data files read by the map() method. Those should be specified directly as part of the filename.

Out−of−core disk files are created with names like "fpath/mrmpi.kv,N,M,P" where "kv" is an file−type string ("kv", or "kmv" or "sort" or "part" or "set"), N is a number unique to each MapReduce object, M is a file counter, and P is the processor ID. fpath/mrmpi.kmv.N.P. Sort files are created by the sorting methods. Part and set files are created by collate() or convert() methods.

Setting *fpath* may be useful for specifying a disk local to each processor, or for a parallel file system that each processor can access.

This setting can only be changed before the first KeyValue or KeyMultiValue object is created by the MapReduce object. If changed after that, it will have no effect.

The default value for *fpath* is ".", which means the current working directory.

The default value can be changed by a compiler setting when the MR−MPI library is built. Using this flag for the compilation of the src/mapreduce.cpp file:

```
−DMRMPI_FPATH=foo
```

where foo is the desired pathname, will build the library with the default fpath set to foo, instead of the current working directory.

MapReduce sort_keys() method

```
uint64_t MapReduce::sort_keys(int (*mycompare)(char *, int, char *, int))
uint64_t MapReduce::sort_keys(int flag)
```

This calls the sort_keys() method of a MapReduce object, which sorts a KeyValue object by its keys to produce a new KeyValue object.

For the first variant, you provide a mycompare() function which compares pairs of keys for the sort, since the MapReduce object does not know how to interpret the content of your keys. The method returns the total number of key/value pairs in the new KeyValue object which will be the same as in the original.

For the second variant, you can select one of several pre−defined compare functions, so you do not have to write the compare function yourself:

flag = +/− 1	compare 2 integers
flag = +/− 2	compare 2 64−bit unsigned integers
flag = +/− 3	compare 2 floats
flag = +/− 4	compare 2 doubles
flag = +/− 5	compare 2 NULL−terminated strings via strcmp()
flag = +/− 6	compare 2 arbitrary strings via strncmp()

If the flag is positive, the sorting is done is ascending order; if the flag is negative, the sorting is done is descending order.

For the flag = +/− 6 case, the 2 strings do not have to be NULL−terminated since only the first N characters are compared, where N is the shorter of the 2 string lengths.

This method is used to sort key/value pairs by key before a KeyValue object is transformed into a KeyMultiValue object, e.g. via the clone(), collapse(), or convert() methods. Note that these operations preserve the order of paires in the KeyValue object when creating a KeyMultiValue object, which can then be passed to your application for output, e.g. via the reduce() method. Note however, that sort_keys() does NOT sort keys across all processors but only sorts the keys on each processor within the KeyValue object. Thus if you gather() or aggregate() after performing a sort_keys(), the sorted order will be lost, since those methods move key/value pairs to new processors.

In this example for the first variant, the user function is called mycompare() and it must have the following interface

```
int mycompare(char *key1, int len1, char *key2, int len2)
```

Key1 and key2 are pointers to the byte strings for 2 keys, each of length len1 and len2. Your function should compare them and return a −1, 0, or 1 if key1 is less than, equal to, or greater than key2, respectively.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor operates only on the key/value pairs it stores.

Related methods: sort_values(), sort_multivalues()

MapReduce sort_multivalues() method

```
uint64_t MapReduce::sort_multivalues(int (*mycompare)(char *, int, char *, int))
uint64_t MapReduce::sort_multivalues(int)
```

This calls the sort_multivalues() method of a MapReduce object, which sorts the values for each key within a KeyMultiValue object to produce a new KeyMultiValue object.

For the first variant, you provide a mycompare() function which compares pairs of values for the sort, since the MapReduce object does not know how to interpret the content of your values. The method returns the total number of key/multi−value pairs in the new KeyMultiValue object which will be the same as in the original.

For the second variant, you can select one of several pre−defined compare functions, so you do not have to write the compare function yourself:

flag = 1	compare 2 integers
flag = 2	compare 2 64−bit unsigned integers
flag = 3	compare 2 floats
flag = 4	compare 2 doubles
flag = 5	compare 2 NULL−terminated strings via strcmp()
flag = 6	compare 2 arbitrary strings via strncmp()

For the flag = 6 case, the 2 strings do not have to be NULL−terminated since only the first N characters are compared, where N is the shorter of the 2 string lengths.

This method can be used to sort a set of multi−values within a key before they are passed to your application, e.g. via the reduce() method. Note that it typically only makes sense to use sort_multivalues() for a KeyMultiValue object created by the convert() or collate() methods, not KeyMultiValue objects created by the clone() or collapse() or scrunch() methods.

In this example for the first variant, the user function is called mycompare() and it must have the following interface

```
int mycompare(char *value1, int len1, char *value2, int len2)
```

Value1 and value2 are pointers to the byte strings for 2 values, each of length len1 and len2. Your function should compare them and return a −1, 0, or 1 if value1 is less than, equal to, or greater than value2, respectively.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor operates only on the key/multi−value pairs it stores.

Related methods: sort_keys(), sort_values()

MapReduce sort_values() method

```
uint64_t MapReduce::sort_values(int (*mycompare)(char *, int, char *, int))
uint64_t MapReduce::sort_values(int flag)
```

This calls the sort_values() method of a MapReduce object, which sorts a KeyValue object by its values to produce a new KeyValue object.

For the first variant, you provide a mycompare() function which compares pairs of values for the sort, since the MapReduce object does not know how to interpret the content of your values. The method returns the total number of key/value pairs in the new KeyValue object which will be the same as in the original.

For the second variant, you can select one of several pre−defined compare functions, so you do not have to write the compare function yourself:

flag = 1	compare 2 integers
flag = 2	compare 2 64−bit unsigned integers
flag = 3	compare 2 floats
flag = 4	compare 2 doubles
flag = 5	compare 2 NULL−terminated strings via strcmp()
flag = 6	compare 2 arbitrary strings via strncmp()

For the flag = 6 case, the 2 strings do not have to be NULL−terminated since only the first N characters are compared, where N is the shorter of the 2 string lengths.

This method is used to sort key/value pairs by value before a KeyValue object is transformed into a KeyMultiValue object, e.g. via the clone(), collapse(), or convert() methods. Note that these operations preserve the order of pairs in the KeyValue object when creating a KeyMultiValue object, which can then be passed to your application for output, e.g. via the reduce() method. Note however, that sort_values() does NOT sort values across all processors but only sorts the values on each processor within the KeyValue object. Thus if you gather() or aggregate() after performing a sort_values(), the sorted order will be lost, since those methods move key/value pairs to new processors.

In this example for the first variant, the user function is called mycompare() and it must have the following interface

```
int mycompare(char *value1, int len1, char *value2, int len2)
```

Value1 and value2 are pointers to the byte strings for 2 values, each of length len1 and len2. Your function should compare them and return a −1, 0, or 1 if value1 is less than, equal to, or greater than value2, respectively.

This method is an on−processor operation, requiring no communication. When run in parallel, each processor operates only on the key/value pairs it stores.

Related methods: sort_keys(), sort_multivalues()

MapReduce kv_stats() method

MapReduce kmv_stats() method

MapReduce cummulative_stats() method

```
uint64_t MapReduce::kv_stats(int level)
uint64_t MapReduce::kmv_stats(int level)
void MapReduce::cummulative_stats(int level, int reset)
```

Calling the kv_stats() method prints statistics about the KeyValue object stored within the MapReduce object. The total number of key/value pairs is returned. If level = 0 is specified, nothing else is done. If level = 1 is specified, a one−line summary is printed for all the key/value pairs across all processors. If a level = 2 is specified, per−processor information is also printed in a one−line histogram format.

Calling the kmv_stats() method prints statistics about the KeyMultiValue object stored within the MapReduce object. The total number of key/multi−value pairs is returned. If level = 0 is specified, nothing else is done. If level = 1 is specified, a one−line summary is printed for all the key/multi−value pairs across all processors. If a level = 2 is specified, per−processor information is also printed in a one−line histogram format.

Calling the cummulative_stats() method prints statistics about the cummulative memory allocation, inter−processor communication volume, and file I/O volume that has been performed by all MapReduce operations up to this point, by all MapReduce objects your program has instantiated. If level = 1 is specified, a brief summary is printed. If level = 2 is specified, per−processor information is also printed in a one−line histogram format.

If the *reset* flag is set to 1, then the counters for these quantities are reset to 0.

This cummulative_stats() method is called internally when your program destructs the last MapReduce object, using the verbosity setting for the level argument. If verbosity is set to 0, then the method is not called.

C interface to the MapReduce−MPI Library

The MR−MPI library can be called from a C program, using the interface defined in src/cmapreduce.h. This is a C file which should be included in your C program to define the API to the library:

```
#include "cmapreduce.h"
```

Note that the C interface should also be usable to call the MapReduce MPI library from Fortran or other hi−level languages, including scripting languages. See information below on how to do this from Python.

The C interface consists of the following functions. Their functionality and arguments are described in the C++ interface section.

```
void *MR_create(MPI_Comm comm);
void *MR_create_mpi();
void *MR_create_mpi_finalize();
void *MR_copy(void *MRptr);
void MR_destroy(void *MRptr);

uint64_t MR_add(void *MRptr);
uint64_t MR_aggregate(void *MRptr, int (*myhash)(char *, int));
uint64_t MR_broadcast(void *MRptr, int root);
uint64_t MR_clone(void *MRptr);
uint64_t MR_close(void *MRptr);
uint64_t MR_collapse(void *MRptr, char *key, int keybytes);
uint64_t MR_collate(void *MRptr, int (*myhash)(char *, int));
uint64_t MR_compress(void *MRptr,
                     void (*mycompress)(char *, int, char *, int, int *, void *KVptr, void *APPptr),
                     void *APPptr);
uint64_t MR_convert(void *MRptr);
uint64_t MR_gather(void *MRptr, int numprocs);

uint64_t MR_map(void *MRptr, int nmap,
                void (*mymap)(int, void *KVptr, void *APPptr),
                void *APPptr);
uint64_t MR_map_add(void *MRptr, int nmap,
                    void (*mymap)(int, void *KVptr, void *APPptr),
                    void *APPptr, int addflag);
uint64_t MR_map_file(void *MRptr, int nstr, char **strings,
                     int self, int recurse, int readfile,
                     void (*mymap)(int, char *,
                                   void *KVptr, void *APPptr),
                     void *APPptr);
uint64_t MR_map_file_add(void *MRptr, int nstr, char *strings,
                         int self, int recurse, int readfile,
                         void (*mymap)(int, char *,
                                       void *KVptr, void *APPptr),
                         void *APPptr, int addflag);
uint64_t MR_map_file_char(void *MRptr, int nmap, int nstr, char **strings,
                          int recurse, int readfile,
                          char sepchar, int delta,
                          void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                          void *APPptr);
uint64_t MR_map_file_char_add(void *MRptr, int nmap, int nstr, char **strings,
                              int recurse, int readfile,
                              char sepchar, int delta,
                              void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                              void *APPptr, int addflag);
```

```
uint64_t MR_map_file_str(void *MRptr, int nmap, int files, char **files,
                         char *sepstr, int delta,
                         void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                         void *APPptr);
uint64_t MR_map_file_str_add(void *MRptr, int nmap, int files, char **files,
                             char *sepstr, int delta,
                             void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                             void *APPptr, int addflag);
uint64_t MR_map_mr(void *MRptr, void *MRptr2,
                   void (*mymap)(uint64_t, char *, int, char *, int *, void *KVptr, void *APPptr),
                   void *APPptr);
uint64_t MR_map_mr_add(void *MRptr, void *MRptr2,
                  void (*mymap)(uint64_t, char *, int, char *, int *, void *KVptr, void *APPptr),
                  void *APPptr, int addflag);

void MR_open(void *MRptr, int addflag);
void MR_open_add(void *MRptr);
void MR_print(void *MRptr, int, int, int, int);
void MR_print_file(void *MRptr, char *, int, int, int, int, int);

uint64_t MR_reduce(void *MRptr,
                   void (*myreduce)(char *, int, char *, int, int *, void *KVptr, void *APPptr),
                   void *APPptr);
uint64_t MR_multivalue_blocks(void *MRptr);
void MR_multivalue_block_select(void *MRptr, int which);
int MR_multivalue_block(void *MRptr, int iblock,
                        char **ptr_multivalue, int **ptr_valuesizes);

uint64_t MR_scan_kv(void *MRptr,
                    void (*myscan)(uint64_t, char *, int, char *, int, void *),
                    void *APPptr);
uint64_t MR_scan_kmv(void *MRptr,
                     void (*myscan)(char *, int, char *, int, int *, void *),
                     void *APPptr);

uint64_t MR_scrunch(void *MRptr, int numprocs, char *key, int keybytes);

uint64_t MR_sort_keys(void *MRptr,
                      int (*mycompare)(char *, int, char *, int));
uint64_t MR_sort_keys_flag(void *MRptr, int);
uint64_t MR_sort_values(void *MRptr,
                        int (*mycompare)(char *, int, char *, int));
uint64_t MR_sort_values_flag(void *MRptr, int);
uint64_t MR_sort_multivalues(void *MRptr,
                             int (*mycompare)(char *, int, char *, int));
```

uint64_t MR_sort_multivalues_flag(void *MRptr, int);

```
void MR_kv_stats(void *MRptr, int level);
void MR_kmv_stats(void *MRptr, int level);

void MR_set_mapstyle(void *MRptr, int value);
void MR_set_verbosity(void *MRptr, int value);
void MR_set_timer(void *MRptr, int value);
void MR_set_memsize(void *MRptr, int value);
void MR_set_keyalign(void *MRptr, int value);
void MR_set_valuealign(void *MRptr, int value);

void MR_kv_add(void *KVptr, char *key, int keybytes,
               char *value, int valuebytes);
void MR_kv_add_multi_static(void *KVptr, int n,
                            char *key, int keybytes,
```

```
                            char *value, int valuebytes);
void MR_kv_add_multi_dynamic(void *KVptr, int n,
                            char *key, int *keybytes,
                            char *value, int *valuebytes);
```

These functions correspond one−to−one with the C++ methods described here, except that for C++ methods with multiple interfaces (e.g. map()), there are multiple C functions, with slightly different names. The MR_set() functions are added to the C interface to enable the corresponding library variables to be set.

Note that when you call MR_create() or MR_copy(), they return a "void *MRptr" which is a pointer to the MapReduce object created by the library. This pointer is used as the first argument of all the other MR calls. This means a C program can effectively instantiate multiple MapReduce objects by simply keeping track of the pointers returned to it.

The remaining arguments of each function call are the same as those used with the C++ methods. The only exceptions are several of the MR_kv_add() functions which take a KVptr as their first argument. This is a pointer to a KeyValue object. These calls are made from your program's mymap(), myreduce(), and mycompress() functions to register key/value pairs with the MR−MPI library. The KVptr is passed as an argument to your functions when they are called back from the MR−MPI library.

See the C programs in the examples directory for examples of how these calls are made from a C program. They are conceptually identical to the C++ programs in the same directory.

Python interface to the MapReduce−MPI Library

A Python wrapper for the MR−MPI library is included in the distribution. The advantage of using Python is how concise the language is, enabling rapid development and debugging of MapReduce programs. The disadvantage is speed, since Python is slower than a compiled language. Using the MR−MPI library from Python incurs two additional overheads, discussed in the Technical Details section.

Before using the MR−MPI library in a Python script, the Python on your machine must be "extended" to include an interface to the MR−MPI library. If your Python script will invoke MPI operations, you will also need to extend your Python with an interface to MPI itself.

Thus you should first decide how you intend to use the MR−MPI library from Python. There are 3 options:

(1) Use the library on a single processor running Python.

(2) Use the library in parallel, where each processor runs Python, but your application script does not use MPI.

(3) Use the library in parallel, where each processor runs Python, and your application also makes MPI calls through a Python/MPI interface.

Note that for (2) and (3) you will not be able to interact with Python interactively by typing commands and getting a response. This is because when you have multiple instances of Python running (e.g. on a parallel machine) they cannot all read what you type.

Working in mode (1) does not require your machine to have MPI installed. You should extend your Python with a serial version of the MR−MPI library and its dummy MPI library. See instructions below on how to do this.

Working in mode (2) requires your machine to have an MPI library installed, but your Python does not need to be extended with MPI itself. The MPI library must be a shared library (e.g. a *.so file on Linux) which is not typically created when MPI is built/installed. See instruction below on how to do this. You should extend your Python with the parallel MR−MPI library which will use the shared MPI system library. See instructions below on how to do this.

Working in mode (3) requires your machine to have MPI installed (as a shared library as in (2)). You must also extend your Python with the parallel MR−MPI library (same as in (2)) and with MPI itself, via one of several available Python/MPI packages. See instructions below on how to do the latter task.

The following sub−sections cover the rest of the Python setup discussion:

This sub−section describes the Python syntax used to invoke the MR−MPI library:

Extending Python with a serial version of the MR−MPI library

From the python directory, type

```
python setup_serial.py build
```

and then one of these commands:

```
sudo python setup_serial.py install
python setup_serial.py install −−home=~/foo
```

The "build" command should compile all the needed MR−MPI C++ files, including the dummy MPI library. The first "install" command will put the needed files in your Python's site−packages sub−directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site−packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the MR−MPI files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, an *mrmpi.py* and *_mrmpi_serial.so* file will be put in the appropriate directory.

Creating a shared MPI library

A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a". Such a shared library is normally not built if you installed MPI yourself, but it is easy to do. Here is how to do it for MPICH, a popular open−source version of MPI, distributed by Argonne National Labs. From within the mpich directory, type

```
./configure −−enable−sharedlib=gcc
make
make install
```

You may need to use "sudo make install" in place of the last line. The end result should be the file libmpich.so in /usr/local/lib. Note that if the file libmpich.a already existed in /usr/local/lib, you will now have both a static and shared MPICH library. This will be fine for Python MR−MPI since it only uses the shared library. But if you build other codes with libmpich.a, then those builds may fail if the linker uses libmpich.so instead, unless other dynamic libraries are also linked to.

Extending Python with a parallel version of the MR−MPI library

From the python directory, type

```
python setup.py build
```

and then one of these commands:

```
sudo python setup.py install
python setup.py install −−home=~/foo
```

The "build" command should compile all the needed MR−MPI C++ files, which will require MPI to be installed on your system. This means it must find both the header file mpi.h and a shared library file, e.g. libmpich.so if the MPICH version of MPI is installed. See the preceding section for how to create a build MPI as a shared library if

it does not exist.

The first "install" command will put the needed files in your Python's site−packages sub−directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.5/site−packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the MR−MPI files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, an *mrmpi.py* and *_mrmpi.so* file will be put in the appropriate directory.

Extending Python with MPI itself

There are several Python packages available that purport to wrap MPI and allow its functions to be called from Python.

These include

- pyMPI
- maroonmpi
- mpi4py
- myMPI
- Pypar

All of these except pyMPI work by wrapping the MPI library (which must be available on your system as a shared library, as discussed above), and exposing (some portion of) its interface to your Python script. This means they cannot be used interactively in parallel, since they do not address the issue of interactive input to multiple instances of Python running on different processors. The one exception is pyMPI, which alters the Python interpreter to address this issue, and (I believe) creates a new alternate executable (in place of python itself) as a result.

In principle any of these Python/MPI packages should work with the MR−MPI library. However, when I downloaded and looked at a few of them, their docuemtation was incomplete and I had trouble with their installation. It's not clear if some of the packages are still being actively developed and supported.

The one I recommend, since I have successfully used it with the MR−MPI library, is Pypar. Pypar requires the ubiqtuitous Numpy package be installed in your Python. After launching python, type

```
>>> import numpy
```

to see if it is installed. If not, here is how to install it (version 1.3.0b1 as of April 2009). Unpack the numpy tarball and from its top−level directory, type

```
python setup.py build
sudo python setup.py install
```

The "sudo" is only needed if required to copy Numpy files into your Python distribution's site−packages directory.

To install PyPar (version pypar−2.1.0_66 as of April 2009), unpack it and from its "source" directory, type

```
python setup.py build
```

```
sudo python setup.py install
```

Again, the "sudo" is only needed if required to copy PyPar files into your Python distribution's site−packages directory.

If you have successully installed Pypar, you should be able to run python serially and type

```
>>> import pypar
```

without error. You should also be able to run python in parallel on a simple test script

```
% mpirun −np 4 python test.script
```

where test.script contains the lines

```
import pypar
print "Proc %d out of %d procs" % (pypar.rank(),pypar.size())
```

and see one line of output for each processor you ran on.

Testing the MR−MPI library from Python

Before importing the MR−MPI library in a Python program, one more step is needed. The interface to the library is via Python ctypes, which loads the shared MR−MPI library via a CDLL() call, which in turn is a wrapper on the C−library dlopen(). This command is different than a normal Python "import" and needs to be able to find the MR−MPI shared library, which is either in the Python site−packages directory or in a local directory you specified in the "python setup.py install" command, as described above.

The simplest way to do this is add a line like this to your .cshrc or other shell start−up file.

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/lib/python2.5/site−packages
```

and then execute the file to insure the path has been updated. This will extend the path that dlopen() uses to look for shared libraries.

To test if the MR−MPI library has been successfully installed, launch python in serial and type

```
>>> from mrmpi import mrmpi
>>> mr = mrmpi()
```

If you get no errors, you're ready to use the library, as described below.

If you built the MR−MPI library for parallel use, launch python in parallel

```
% mpirun −np 4 python test.script
```

where test.script contains the lines

```
import pypar
from mrmpi import mrmpi
mr = mrmpi()
print "Proc %d out of %d procs has" % (pypar.rank(),pypar.size()), mr
pypar.finalize()
```

Again, if you get no errors, you're good to go.

Using the MR−MPI library from Python

The Python interface to the MR−MPI library consists of an "mrmpi" class which creates a "mrmpi" object, with a set of methods that can be invoked on that object. The sample code lines below assume you have first imported the "mrmpi" module as follows:

```
from mrmpi import mrmpi
```

Note that when your script imports the Pypar package (same with some other Python/MPI packages), it initializes MPI for you. Pypar does not, however, make the global MPI communicator (MPI_COMM_WORLD) visible to your program, so you can't pass it to the MR−MPI library. When using Pypar, the last line of your input script should thus be pypar.finalize(), to insure MPI is shut down correctly.

Some of the methods defined by the mrmpi class take callback functions as arguments, e.g. map() and reduce(). These are Python functions you define elsewhere in your script. When you register "keys" and "values" with the library, they can be simple quantities like strings or ints or floats. Or they can be Python data structures like lists or tuples.

These are the class methods defined by the mrmpi module. Their functionality and arguments are described in the C++ interface section.

```
mr = mrmpi()                # create an mrmpi object
mr = mrmpi(mpi_comm)        # ditto, but with a specified MPI communicator
mr = mrmpi(0.0)             # ditto, and the library will finalize MPI

mr2 = mr.copy()             # copy mr to create mr2

mr.destroy()                # destroy an mrmpi object, freeing its memory
                            # this will also occur if Python garbage collects

mr.add(mr2)
mr.aggregate()
mr.aggregate(myhash)        # if specified, myhash is a hash function
                            #   called back from the library as myhash(key)
                            # myhash() should return an integer (a proc ID)
mr.broadcast(root)
mr.clone()
mr.close()
mr.collapse(key)
mr.collate()
mr.collate(myhash)          # if specified, myhash is the same function
                            #   as for aggregate()

mr.compress(mycompress)     # mycompress is a function called back from the
                            #   library as mycompress(key,mvalue,mr,ptr)
                            #   where mvalue is a list of values associated
                            #   with the key, mr is the MapReduce object,
                            #   and you (optionally) provide ptr (see below)
                            # your mycompress function should typically
                            #   make calls like mr->add(key,value)
mr.compress(mycompress,ptr) # if specified, ptr is any Python datum
                            #    and is passed back to your mycompress()
                            # if not specified, ptr = None

mr.convert()
mr.gather(nprocs)
```

```
mr.map(nmap,mymap)          # mymap is a function called back from the
                            #   library as mymap(itask,mr,ptr)
                            #   where mr is the MapReduce object,
                            #   and you (optionally) provide ptr (see below)
                            # your mymap function should typically
                            #   make calls like mr->add(key,value)
mr.map(nmap,mymap,ptr)      # if specified, ptr is any Python datum
                            #    and is passed back to your mymap()
                            # if not specified, ptr = None
mr.map(nmap,mymap,ptr,addflag) # if addflag is specfied as a non-zero int,
                               #   new key/value pairs will be added to the
                               #   existing key/value pairs

mr.map_file(files,self,recurse,readfile,mymap)
                             # files is a list of filenames and dirnames
                             # mymap is a function called back from the
                             #   library as mymap(itask,filename,mr,ptr)
                             # as above, ptr and addflag are optional args
mr.map_file_char(nmap,files,recurse,readfile,sepchar,delta,mymap)
                             # files is a list of filenames and dirnames
                             # mymap is a function called back from the
                             #   library as mymap(itask,str,mr,ptr)
                             # as above, ptr and addflag are optional args
mr.map_file_str(nmap,files,recurse,readfile,sepstr,delta,mymap)
                             # files is a list of filenames and dirnames
                             # mymap is a function called back from the
                             #   library as mymap(itask,str,mr,ptr)
                             # as above, ptr and addflag are optional args
mr.map_mr(mr2,mymap)         # pass key/values in mr2 to mymap
                             # mymap is a function called back from the
                             #   library as mymap(itask,key,value,mr,ptr)
                             # as above, ptr and addflag are optional args

mr.open()
mr.open(addflag)
mr.print_screen(proc,nstride,kflag,vflag)
mr.print_file(file,fflag,proc,nstride,kflag,vflag)

mr.reduce(myreduce)          # myreduce is a function called back from the
                             #   library as myreduce(key,mvalue,mr,ptr)
                             #   where mvalue is a list of values associated
                             #   with the key, mr is the MapReduce object,
                             #   and you (optionally) provide ptr (see below)
                             # your myreduce function should typically
                             #   make calls like mr->add(key,value)
mr.reduce(myreduce,ptr)      # if specified, ptr is any Python datum
                             #    and is passed back to your myreduce()
                             # if not specified, ptr = None

mr.scan_kv(myscan)           # myscan is a function called back from the
                             #   library as myscan(key,value,ptr)
                             #   for each key/value pair
                             #   and you (optionally) provide ptr (see below)
mr.scan_kv(myscan,ptr)       # if specified, ptr is any Python datum
                             #    and is passed back to your myreduce()
                             # if not specified, ptr = None

mr.scan_kmv(myscan)          # myscan is a function called back from the
                             #   library as myreduce(key,mvalue,ptr)
                             #   where mvalue is a list of values associated
                             #   with the key,
                             #   and you (optionally) provide ptr (see below)
mr.scan_kmv(myscan,ptr)      # if specified, ptr is any Python datum
```

```
                              #    and is passed back to your myreduce()
                              # if not specified, ptr = None

mr.scrunch(nprocs,key)
mr.sort_keys(mycompare)
mr.sort_values(mycompare)
mr.sort_multivalues(mycompare) # compare is a function called back from the
                               #   library as mycompare(a,b) where
                               #   a and b are two keys or two values
                               # your mycompare() should compare them
                               #   and return a -1, 0, or 1
                 <b, or a == b, #or ai f a
mr.sort_keys_flag(flag)
mr.sort_values_flag(flag)
mr.sort_multivalues_flag(flag)

mr.kv_stats(level)
mr.kmv_stats(level)

mr.mapstyle(value)             # set mapstyle to value
mr.all2all(value)              # set all2all to value
mr.verbosity(value)            # set verbosity to value
mr.timer(value)                # set timer to value
mr.memsize(value)              # set memsize to value
mr.minpage(value)              # set minpage to value
mr.maxpage(value)              # set maxpage to value

mr.add(key,value)                 # add single key and value
mr.add_multi_static(keys,values)  # add list of keys and values
                                  # all keys are assumed to be same length
                                  # all values are assumed to be same length
mr.add_multi_dynamic(keys,values) # add list of keys and values
                                  # each key may be different length
                                  # each value may be different length
```

These class methods correspond one−to−one with the C++ methods described here, except that for C++ methods with multiple interfaces (e.g. map()), there are multiple Python methods with slightly different names, similar to the C interface.

There is no set function the the *keyalign* and *valuealign* settings. These are hard−wired to 1 for the Python interface, since no other values make sense, due to the pickling/unpickling that is performed in key and value data.

See the Python scripts in the examples directory for examples of how these calls are made from a Python program. They are conceptually identical to the C++ and C programs in the same directory.

OINK interface to the MapReduce−MPI Library

OINK is a C++ application that provdes a hi−level scripting interface to the MR−MPI library which it uses internally. These are three goals of OINK:

- (1) To allow MapReduce algorithms which call the MR−MPI library to be written with a minimum of extraneous code, to work with input/output in various forms, and to be chained together and driven via a simple, yet versatile scripting language.
- (2) To create an archive of map() and reduce() functions for re−use by different algorithms.
- (3) To provide a scripted interface to the lo−level MR−MPI library calls that can speed development/debugging of new algortihms before coding them up in C++ or another language.

OINK has its own manual and doc pages, so further details are not given here.

Technical Details

This section provides additional details about using the MapReduce library and how it is implemented. These topics are covered:

Length and byte−alignment of keys and values

As explained in this section, keys and values are variable−length strings of bytes. The MR−MPI library knows nothing of their contents and simply treats them as contiguous chunks of bytes.

When you register a key and value in your mymap() or mycompress() or myreduce() function via the KeyValue add() method, you specify their lengths in bytes. Keys and values are typically returned to your program for further processing or output, e.g. as arguments passed to your myreduce() function by the reduce() operation, as are their lengths.

Keys and values are passed as character pointers to your functions where you may need to convert the pointer to an appropriate data type and then correctly interpret the byte string. For example, either of these lines could be used:

```
int *iptr = (int *) key;
int myvalue = *((int *) key);
```

If the key or value is a variable−length text string, you may want to terminate it with a "0", and include the trailing "0" in the byte count, so that C−library−style string functions can later be invoked on it. If a key or value is a complex data structure, your function must be able to decode it.

A related issue with keys and values is the byte−alignment of integer or floating point values they include. For example, it is usually a bad idea to store an 8−byte double such that it is mis−aligned with respect to an 8−byte boundary in memory. The reason is that using a mis−aligned double in a computation may be slow.

If your keys or values are homogeneous (e.g. all integers), you can use the *keyalign* and *valuealign* settings, discussed here, to insure alignment of keys and values to desired byte boundaries. Since this may incur extra memory costs, you should not typically make these settings larger than needed.

Special care may need to be taken if your values are heterogeneous, e.g. a mixture of strings and integers. This is because the MR−MPI library packs values one after the other into one long byte string when it is returned to your program as a multi−value, e.g. as an argument to the callback of a reduce() method. Only the first value in the multi−value is aligned to the *valuealign* setting. Similarly, the collapse() method creates a multi−value that is sequence of key,value,key,value,etc from a KV. If the keys are variable−length text strings and the values are

integers, then the values will not be aligned on 4−byte boundaries.

Here are two ideas that can be used to insure alignment of heterogeneous data:

(a) Say your "value" is a 4−byte integer followed by an 8−byte double. You might think it can be stored and registered as 12 contiguous bytes. However, this would likely mean the double is mis−aligned. One solution is to convert the integer to a double before storing both quantities in a 16−byte value string. Another solution is to create a struct to store the integer and double and use the sizeof() function to determine the length of the struct and use that as the length of your "value". The compiler should then guarantee proper alignment of each structure member.

(b) Your callback function can always copy the bytes of a key or value into a local data structure with the proper alignment, e.g. using the C memcpy() function. E.g. in the collapse example above, these lines of code:

```
int myvalue;
memcpy(lue[offset],sizeof(int));
```

would load the 4 bytes of a particular value (at location offset) in the multi−value into the local integer "myvalue", where it can then be used for computation.

Memory requirements for KeyValue and KeyMultiValue objects

KeyValue and KeyMultiValue objects are described in this section. A MapReduce object contains either a single KeyValue object (KV) or a single KeyMultiValue object (KMV), depending on which methods you have invoked.

The memory cost for storing key/value pairs in a KV is as follows. The key and value each have a byte length. Two integers are also stored for the key and value length. There may also be additional bytes added to align the key and value on byte boundaries in memory; see the *keyalign* and *valuealign* settings, discussed in this section. Thus the total size of a KV is the memory for the key/value datums plus 2 integers per pair plus any extra alignment bytes.

A KMV contains key/multi−value pairs where the number of pairs is typically the number of unique keys in the original KV. The memory cost for storing key/multi−value pairs in a KMV is as follows. The key and multi−value each have a byte length. For the multi−value, this is the sum of individual value lengths. Again, there may also be additional bytes added to align the key and multi−value on byte boundaries in memory; see the *keyalign* and *valuealign* settings, discussed in this section. Three integers are also stored: the key and multi−value length, and the number of values N in the multi−value. An N−length array of integers is also stored for the length of each value in the multi−value. Thus the total size of a KMV is the memory for the key/multi−value datums plus 3 integers per pair plus 1 integer per value in the original KV plus any extra alignment bytes.

Note that memory for key data in a KMV is typically less than in the original KV, since the KMV only stores unique keys. The memory for multi−value data is the same as the value data in the original KV, since all the original KV values are contained in the multi−values.

Note that in parallel, for a KV or KMV, each processor stores the above data for only a fraction of key/value pairs it generated during a map() operation or acquired during other operations, like a collate(). If this is imbalanced, one processor may own and process datums more than other processors.

If KV or KMV data on a processor exceeds the page size determined by the *memsize* setting, discussed here, then data is written to temporary disk files, on a per−processor basis.

Out−of−core operation

If the KV or KMV pairs of a data set owned by a processor fit within a single page of memory, whose size is determined by the *memsize* setting, then the MR−MPI library operates on the data in−core; no disk files are written or read.

When the data on any single processor exceeds the page size, that processor will write data, one page at a time, to one or more temporary disk files, and later read it back in as needed, again one page at a time. Thus all the MR−MPI methods can be invoked on data sets larger than fit in the aggregate memory of the processors being used. The only real limitation in this case is available disk space.

All of the MR−MPI methods, except one, perform their operations within a fixed number of memory pages. This includes memory needed for message passing calls to the MPI library, e.g. buffers used to send and receive data. Any large data exchanges are performed with pre−posted receives (MPI_Irecv) into user−space memory, which do not require additional internal MPI library memory.

The number of required pages ranges from 1 to 7, and is listed on this page for each MR−MPI library method. This means, for example, that even if the page size is 1 Mb (smallest allowed value), and the data set size is 10 Gb per processor, and the sort_keys() method is invoked, which requires 5 pages per processor, that the operation will successfully complete, using only 5 Mb per processor. Of course, there may be considerable disk I/O performed along the way.

The one exception is the convert() method, also called by the collate() and commpress() methods, which performs an on−processor reorganization of the data in a KV to produce a KMV. For large data sets this requires breaking up the large KV data file into smaller files, each of which holds data that will contribute to one page of the eventual KMV file. Each smaller file requires an in−memory buffer to store data that is written to the file. The number of these smaller files, and hence the number of buffers, is hard to predict in advance or even bound. It depends on the page size and the characteristics of the KV pairs, e.g. how many unique keys there are. The number of extra allocated pages needed to store these buffers depends of the number of small files and the minimum buffer size, which is currently set at 16K bytes for reasonable disk I/O performance. If a very large number of small files are needed to partition the KV data and the page size is small, then several extra memory pages may need to be allocated. This is not normally the case, but the number of small files and number of allocated pages can be monitored if the *verbosity* setting is non−zero. Note that a larger page size will reduce the number of extra pages the convert() method needs to allocate.

IMPORTANT NOTE: You should choose a *memsize* setting that insures the total memory consumed by all pages allocated by all the MapReduce objects you create, does not exceed the physical memory available (which may be shared by several processors if running on a multi−core node). If you do this, then many systems will allocate virtual memory, which will typically cause MR−MPI library operations to run very slowly and thrash the disk.

Also note that in addition to "pages", there are numerous additional small allocations of memory made by the MR−MPI library. Here are two examples. The aggregate() method allocates vectors of length P = the number of processors. Out−of−core disk files are stored as "pages" of data. Each page requires some in−memory bookkeeping so it can be written and read. Thus if a file grows to 1000s of pages, the corresponding in−memory bookkeeping structure will also become larger. For normal page sizes as determined by the *memsize* setting, e.g. the 64 Mbyte default, these additional in−memory allocations should be small compared to the size of a single page.

Fundamemtal library limits

Even in out−of−core mode, the MR−MPI library has limitations on the data set sizes it can process. In practice, these are hopefully not restrictive limits.

Define:

- INTMAX = 2^31 − 1 = largest 32−bit signed int
- UINT64MAX = 2^64 − 1 = largest 64−bit unsigned int
- pagesize = size (in bytes) of 1 page of memory

Internal storage limits within library:

KV = KeyValue, KMV = KeyMultiValue

- UINT64MAX = max byte count of KV or KMV data across all procs
- UINT64MAX = max # of KV or KMV pairs across all procs
- UINT64MAX = max # of values in a single KMV pair
- UINT64MAX = max pagesize
- min(pagesize,INTMAX) = max size of 1 KV pair
- INTMAX = max number of KV or KMV pairs in one page (on a processor)
- INTMAX = max # of values in single KMV pair, before split across pages
- INTMAX = max summed value size in single KMV pair, before split across pages

Additional notes:

The user sets the "pagesize" via the *memsize* setting, in Mbytes. The pagesize can exceed INTMAX, though it should not exceed the physical memory available. See the discussion above for more details.

Since the data set size is written to disk, when the library operates in out−of−core mode, the data size cannot exceed available disk space, either on a per−processor basis (if each processor is writing to its own local disk), or in aggregate (e.g. for a parallel file system). Some MR−MPI operations convert data from one form to another (e.g. KV to KMV) or make intermediate copies of data (e.g. for sorting). At a minimum this typically requires 2x the disk space of the data set itself.

As discussed here, a KeyValue pair requires 2 integers plus the key and value, plus alignment space. For a 1−byte key and a 0−byte value, this is a minimum of 12 bytes. By storing no more than INTMAX KeyValue pairs on a page, this still allows for pagesizes of nearly 24 Gb, more if KeyValue pair sizes are larger.

The various INTMAX limits mean that user calls to the library, and library callbacks to user functions can use int parameters rather than uint64 parameters. It also reduces storage requirements for individual KeyValue and KeyMultiValue pairs. One exception is that all the library methods return a uint64 for the final number of KeyValue or KeyMultiValue pairs stored by the library. Another exception is the uint64 "itask" variable passed back to one flavor of the user mymap() function via the map() method.

The INTMAX limits on the number of KeyMultiValue values stored in one page, mean that individual KeyMultiValue pairs that exceed this will be split across multiple pages. The user callback functions access these pages via the multivalue_blocks() and multivalue_block() methods, described witht the reduce() method.

Hash functions

The convert() and collate() methods use a hash function to organize keys and find duplicates. The MR−MPI library uses the hashlittle() function from lookup3.c, written by Bob Jenkins and available freely on the WWW. It operates on arbitrary−length byte strings (a key) and produces a 32−bit integer hash value, a portion of which is used as a bucket index into a hash table.

Callback functions

Several of the library methods take a callback function as an argument, meaning that function is called back to from the library when the method is invoked. These functions are part of your MapReduce program and can perform any operation you wish on your data (or on no data), so long as they produce the appropriate information. E.g. they generate key/value pairs in the case of map() or compress() or reduce(), or they hash a key to a processor in the case of aggregate() or collate(), or they compare two keys or values in the case of sort_keys() or sort_values().

The mymap() and myreduce() functions can perform simple operations or very complex, compute−intensive operations. For example, if your parallel machine supports it, they could invoke another program or script to read/parse an input file or calculate some result.

Note that in your program, a callback function CANNOT be a class method unless it is declared to be "static". It can also be a non−class method, i.e. just a stand−alone function. In either case, such a function cannot access class data.

One way to get around this restriction is to define global variables that allow your function to access information it needs.

Another way around this restriction is to use the feature provided by several of the library methods with callback function arguments which allow you to pass in a pointer to whatever data you wish. This pointer is returned as an argument when the callback is made. This pointer should be cast to (void *) when passed in, and your callback function can later cast it back to the appropriate data type. For example, a class could set the pointer to an array or an internal data structure or the class itself as "(void *) this". Specify a NULL if your function doesn't need the pointer.

Python overhead

Using the MR−MPI library from Python incurs two not−so−obvious overheads beyond the usual slowdown due to using an interpreted language. First, Python objects used as keys and values are "pickled" and "unpickled" using the cPickle Python library when passed into and out of the C++ library. This is because the library stores them as byte strings. The pickling process serializes a Python object (e.g. an integer, a string, a tuple, or a list) into a byte stream in a way that it can be unpickled into the same Python object.

The second overhead is due to the complexity of making a double callbacks between the library and your Python script. I.e. the library calls back once to the user program which then calls back into the library. Consider what happens during a map() operation when the library is called from a C++ program.

- the program calls the library map() method
- the library map() calls back to the user map() callback function
- the user map() calls the library add() method to register a key/value pair

When doing this from Python there are 3 additional layers between the Python program and the library, the Python mrmpi class, an invisible C layer (created by ctypes), and the C interface on the C++ library itself. Thus the callback operation proceeds as follows:

- the program calls the mrmpi class map() method
- the mrmpi class map() calls the invisible C map() function
- the invisible map() calls the C interface map() function
- the C interface map() calls the library map() method
- the library map() calls back to the invisible C callback function
- the invisible callback calls the mrmpi class callback method
- the mrmpi callback calls the user map() callback function
- the user map() calls the mrmpi class add() method to register a key/value pair
- the mrmpi class add() calls the invisible C add() function
- the invisible add() calls the C interface add() function
- the C interface add() calls the library add() method

Thus 3 calls have become 11 due to the 3 additional layers data must pass through. Some of these pass throughs are very simple, but others require massaging and copying of data, like the pickling/unpickling described above, which occurs in the mrmpi class methods. I was somewhat surprised this double−callback sequence works as well and as transparently as it does − Python ctypes is amazing!

Error messages

The error messages printed out by the MR−MPI library are hopefully self−explanatory. At some point they will be listed in these doc pages.

Examples

This section describes the MapReduce programs provided in the examples directory of the distribution:

- wordfreq
- rmat

Each are provided in 3 formats: as a C++ program, C program, and Python script. Note that the Python scripts use the PyPar package which provides a Python/MPI interface, as discussed above in the Python Interface section, so you must have PyPar installed in your Python to run them.

The C++ and C programs can be built (assuming you have already built the MR−MPI library) by typing

```
make −f Makefile.foo
```

from within the examples directory, using one of the provided Makefiles. As with the library itself, you may need to edit one of the Makefiles to create a new version appropriate to your machine.

Word frequency example

The wordfreq programs implement the word frequency counting algorithm described above in this section. The wordfreq programs are run by giving a list of text files as arguments, e.g.

```
wordfreq ~/mydir/*.cpp
mpirun −np 8 wordfreq ~/mydir/*.cpp
cwordfreq ~/mydir/*.cpp
mpirun −np 8 cwordfreq ~/mydir/*.cpp
python wordfreq.py ~/mydir/*.cpp
mpirun −np 8 python wordfreq.py ~/mydir/*.cpp
```

Total word counts and a list of the top 10 words should be printed to the screen, along with the time to perform the operation.

The 3 different versions of the wordfreq program should give the same answers, although if non−text files are used, the parsing of the contents into words can be done differently by the C library strtok() function and the Python string "split" method.

R−MAT matrices example

The rmat programs generate a particular form of randomized sparse matrix known as an R−MAT matrix. Depending on the parameters chosen, the sparsity pattern in the resulting matrix can be highly non−uniform, and a good model for irregular graphs, such as ones representing a network of computers or WWW page links.

The rmat programs are run by specifying a few parameters, e.g.

```
rmat N Nz a b c d frac outfile
mpirun −np 8 rmat N Nz a b c d frac outfile
crmat N Nz a b c d frac outfile
mpirun −np 8 crmat N Nz a b c d frac outfile
python rmat.py N Nz a b c d frac outfile
```

```
mpirun −np 8 python rmat.py N Nz a b c d frac outfile
```

The meaning of the parameters is as follows. Note that only matrices with a power−of−2 number of rows can be generated, so specifying N=20 creates a matrix with over a million rows.

- 2^N = # of rows in matrix
- Nz = average # of non−zeroes per row
- a,b,c,d = generation params for matrix entries, must sum to 1
- frac = randomization parameter between 0 and 1
- seed = random # seed, positive integer
- outfile = optional output file

A full description of the R−MAT generation algorithm is beyond the scope of this doc page, but here's the brief version. The *a,b,c,d* parameters are effectively weights on the 4 quadrants of the matrix. To generate a single new matrix element, one quadrant is chosen, with a probability proportional to its weight. This operation is repeated recursively within the chosen quadrant, applying the *frac* parameter to randomize the weights a bit. After N iterations, a single I,J matrix location has been identified and its value is set (to 1 in this case).

The total number of matrix entries generated is Nx * 2^N. This procedure can generate duplicates, so those are removed, and new elements generated until the desired number is reached.

When completed, the matrix statistics are printed to the screen, along with the time to generate the matrix. If the optional *outfile* parameter is specified, then the matrix entries are written to files (one per processor). Each line of any file has the form

```
I J value
```

where I,J are the matrix row,column and value is the matrix entry (all are 1 in this case). If the files are concatenated together, the full set of matrix entries should result.

The 3 different versions of the rmat programs should give the same answers in a statistical sense. The answers will not be identical because the same random number generation scheme is not used in all 3 programs.

(RMAT) D. Chakrabarti, Y. Zhan, C. Faloutsos, R−MAT: A Recursive Model for Graph Mining", if Proceedings of the SIAM Conference on Data Mining (2004), available at http://www.cs.cmu.edu/~deepay/mywww/papers/siam04.pdf.