MapReduce−MPI Library Manual

MapReduce−MPI Library

http://www.cs.sandia.gov/~sjplimp/mapreduce.html − Sandia National Laboratories

Table of Contents

MapReduce−MPI Library

This document describes the 13 April 2009 version of the MapReduce−MPI (MR−MPI) library that implements the MapReduce operation popularized by Google. The library is designed for parallel execution on distributed−memory platforms, but will also operate on a single processor. The library is written in C++, is callable from hi−level langauges (C++, C, Fortran, Python, or other scripting languages), and requires no additional software except linking with MPI (a message passing 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 by 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 goal of this 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.

Source code for the library 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 it can be used by anyone for any purpose. See the LICENSE file provided with the distribution for more details.

The distrubution includes a few examples of simple programs that illustrate the use of MapReduce.

The authors of this library are Steve Plimpton at Sandia National Laboratories and Karen Devine who can be contacted via email: sjplimp at sandia.gov, kddevin at sandia.gov.

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, since their Linux cluster has one disk per processing node.

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.

This library also has the following limitations, which may be overcome in future releases:

- No fault toleranace. 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.
- No out−of−core operations. Most of the large parallel machines at Sandia do not have one disk per processor or node. Rather they have a few I/O nodes shared by 1000s of processors. This makes out−of−core processing via disk access by all processors less effective and less portable. While these machines do have huge aggregate memory, it does mean the library is limited to processing data sets that will fit in that memory. This can be a limitation, particularly when the intermediate data set (between the map and reduce operations) is large.

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.

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 in 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 libraries. 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.

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
- python
- src
- user

The doc directory contains this 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 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 −f Makefile.foo
```

where you should create and use a Makefile.foo appropriate for your machine, using one of the provided Makefiles as a template. Note that Makefile.serial builds the library for a serial machine, using the dummy MPI library in mpistubs. Other Makefiles build it for parallel execution using a MPI library installed on your machine. You may need to edit one of the Makefiles to be compatible with your compilers and MPI installation. If you use the dummy MPI library, you will need to build it first, by typing "make" from within mpistubs. Again, you may need to edit mpistubs/Makefile for your machine.

If you successfully build the MR−MPI library, you should produce the file "libmrmpi.a" which can be linked by other programs. As discussed below, both a C++ and C interface are part of the library, so the library should be usable from any hi−level language.

To use the MR−MPI library from Python, you don't need to build it from the src directory. Instead, you build it as a dynamic library from the python directory. Instructions are given below in the Python interface section.

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

```
make −f Makefile.foo
```

from within the examples directory. Again, one of the provided Makefiles may need to be modified for your platform. Some of the example programs are provided as a C++ program, a C program, and as a Python script.

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" from the library as it performs the parallel map and reduce operations.

A typical 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 Library

This 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 at the end of the section.

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 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:

aggregate()	KV −> KV	keys are aggregated onto procs	parallel
clone()	KV −> KMV	each KV key becomes a KMV key	serial
collapse()	KV −> KMV	all KV keys become one KMV key	serial
collate()	KV −> KMV	aggregate + convert	parallel
compress()	KV −> KV	calls back to user program to compress duplicate keys	serial
convert()	KV −> KMV	duplicate KV keys become one KMV key	serial
gather()	KV −> KV	collect keys on many procs to few procs	parallel
map()	create or add to a KV	calls back to user program to generate keys	serial
reduce()	KMV −> KV	calls back to user program to process multi−values	serial
scrunch()	KV −> KMV	gather + collapse	parallel
sort_keys()	KV −> KV	calls back to user program to sort keys	serial
sort_values()	KV −> KV	calls back to user program to sort values	serial
sort_multivalues()	KMV −> KMV	calls back to user program to sort multi−values	serial

If a method creates a new KV or KMV, the old one is deleted, if it existed. 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.

Instantiate 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 by copying from an existing MapReduce object (see below). 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:

```
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);
```

Copy a MapReduce object

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

Any of these segments of code will create a new MapReduce object mr2 with contents that are a deep copy from the existing MapReduce object mr1:

```
MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
MapReduce *mr2 = new MapReduce(*mr);

MapReduce mr(MPI_COMM_WORLD);
MapReduce mr2 = mr;

MapReduce mr(MPI_COMM_WORLD);
MapReduce mr2(mr);
```

A deep copy 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 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 KeyValue::add(KeyValue *kv) method for how to merge the key/value pairs from two MapReduce objects into one.

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
```

MapReduce aggregate() method

```
int 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.

MapReduce clone() method

```
int 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.

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.

MapReduce collapse() method

```
int 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 clone() 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.

MapReduce collate() method

```
int 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. See the description of that method 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. One exception would be if your map operation produces a KeyValue object which is highly imbalanced across processors. The aggregate() method should redistribute the key/value pairs more evenly.

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

MapReduce compress() method

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

This calls the compress() method of a MapReduce object, which 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. 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 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. 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.

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 registration is described below with the KeyValue::add() method. For example, if the set of nvalues were integers, the compressed value might be the sum of those integers.

See the Technical Details section for details ont the byte−alignment of keys and values that are passed to your mycompress() function and on how to byte−align keys and values you register with the KeyValue::add() method.

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.

MapReduce convert() method

```
int 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().

MapReduce gather() method

```
int 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.

MapReduce map() method

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

int MapReduce::map(char *file, void (*mymap)(int, char *, KeyValue *, void *), void *ptr)
int MapReduce::map(char *file, void (*mymap)(int, char *, KeyValue *, void *), void *ptr, int addfla

int MapReduce::map(int nmap, int nfiles, char **files, char sepchar, int delta, void (*mymap)(int, c
int MapReduce::map(int nmap, int nfiles, char **files, char sepchar, int delta, void (*mymap)(int, c
```

```
int MapReduce::map(int nmap, int nfiles, char **files, char *sepstr, int delta, void (*mymap)(int, c
int MapReduce::map(int nmap, int nfiles, char **files, char *sepstr, int delta, void (*mymap)(int, c
```

This calls the map() method of a MapReduce object, passing it nmap which is the total number of map tasks to perform across all processors. A function pointer to a mapping function you write is also passed. This method either creates a new KeyValue object or uses an existing KeyValue object to store all the key/value pairs generated by your mymap function. The method returns the total number of key/value pairs in the KeyValue object.

The first set of variants (with and without addflag) simply specify a number of map tasks (nmap) to perform. The index of the map task (see below) is passed back to your mymap() function.

The second set of variants specifies a file that contains a list of files. Each filename and an index is passed back to your mymap() function. The specified file should list one file per line. Blank lines are not allowed. Leading and trailing whitespace around the filename is OK.

For the third set of variants you pass in a list of one or more files and a separation character (sepchar). For the fourth set of variants, you likewise pass in a list of one or more files and a separation string (sepstr). For these variants, the file(s) are split into nmap chunks with roughly equal numbers of bytes in each chunk. One chunk of one of the files is read and is passed back to your mymap() function, so your code does not have to read the file(s). See more details below about the splitting methodology and the delta input parameter.

You can give these 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.

If the last argument *addflag* is omitted or is specified as 0, then map() will create a new KeyValue object, after deleting an existing KeyValue object. If addflag is non−zero, then key/value pairs generated by your mymap() function are added to an existing KeyValue object (which is created if needed).

In this example the user function is called mymap() and it has one of three interfaces depending on which variant of the map() method is used:

```
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)
```

In the first case, a task ID (0 <= itask < nmap) is passed to your function. You can use itask to select a file for your mymap() function to open and read or perform some other operation.

In the second case, a filename and a task ID (0 <= itask < nfiles, where nfiles if the # of files listed in the file you specify) is passed to your function. You can open and read the file.

In the third case, a task ID (0 <= itask < nmap) and a chunk of bytes read from one of the input files is passed to your function. Its length in bytes, including a trailing '\0' that is appended, is given by size.

In all cases, your function is also passed a kv pointer to a new KeyValue object created and stored internally by the MapReduce object, and is also passed back the ptr you specified.

For map() methods that use files and a separation criterion, 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".

The MapReduce map() method assigns the nmap tasks to processors. Different options for how it does this can be controlled by MapReduce settings, described below. 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 below with the KeyValue::add() method.

See the Technical Details section for details on the length and byte−alignment of keys and values you register with the KeyValue::add() method.

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.

MapReduce reduce() method

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

This calls the reduce() method of a MapReduce object, passing it a function pointer to a reduce function you write. It operates on a KeyMultiValue object, calling your myreduce function once for each unique key/multi−value 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:

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

A single key/multi−value 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. 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.

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 below with the KeyValue::add() method. Alternatively, your myreduce() function can write information to an output file.

See the Technical Details section for details on the byte−alignment of keys and values that are passed to your myreduce() function and on how to byte−align keys and values you register with the KeyValue::add() method.

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.

MapReduce scrunch() method

```
int 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 description of that method for details. The key and keybytes arguments are used by the callapse() portion of the operation. See the description of that method 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).

MapReduce sort_keys() method

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

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

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 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.

MapReduce sort_values() method

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

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

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 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.

MapReduce sort_multivalues() method

```
int MapReduce::sort_multivalues(int (*mycompare)(char *, int, char *, 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. The mycompare() function you provide 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 KeyMultiValue object which will be the same as in the original.

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 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.

MapReduce kv_stats() method

```
void MapReduce::kv_stats(int level)
```

Calling this method prints statistics about the KeyValue object stored within the MapReduce object. 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.

MapReduce kmv_stats() method

```
void MapReduce::kmv_stats(int level)
```

Calling this method prints statistics about the KeyMultiValue object stored within the MapReduce object. 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.

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)
void KeyValue::add(KeyValue *kv)
```

The first three of these add 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 above, 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 Technical Details section for details on the length and byte−alignment of keys and values you register with the add() method.

The fourth of these add methods can be called directly to add the key/value pairs from one MapReduce object to those of another. For example, these lines

```
MapReduce *mr1 = new MapReduce(MPI_COMM_WORLD);
mr1->map(ntasks,
MapReduce *mr2 = new MapReduce(*mr1);
mr2->collate();
```

```
mr2->reduce(
mr1->kv->add(mr2->kv);
delete mr2;
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.

Settings and defaults

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

```
mapstyle = 0 (chunk) or 1 (stride) or 2 (master/slave)
verbosity = 0 (none) or 1 (summary) or 2 (histogrammed)
```

Your program can set or change these values at any time, e.g.

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

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

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.

The default value for *mapstyle* is 0.

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 imbalance in memory usage.

The default value for *verbosity* is 0.

C interface to the 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);

int MR_aggregate(void *MRptr, int (*myhash)(char *, int));
int MR_clone(void *MRptr);
int MR_collapse(void *MRptr, char *key, int keybytes);
int MR_collate(void *MRptr, int (*myhash)(char *, int));
int MR_compress(void *MRptr,
                void (*mycompress)(char *, int, char *,
                                   int, int *, void *KVptr, void *APPptr),
                void *APPptr);
int MR_convert(void *MRptr);
int MR_gather(void *MRptr, int numprocs);
int MR_map(void *MRptr, int nmap,
           void (*mymap)(int, void *KVptr, void *APPptr),
           void *APPptr);
int MR_map_add(void *MRptr, int nmap,
               void (*mymap)(int, void *KVptr, void *APPptr),
               void *APPptr, int addflag);
int MR_map_file_list(void *MRptr, char *file,
                     void (*mymap)(int, char *, void *KVptr, void *APPptr),
                     void *APPptr);
int MR_map_file_list_add(void *MRptr, char *file,
                         void (*mymap)(int, char *, void *KVptr, void *APPptr),
                         void *APPptr, int addflag);
int MR_map_file_char(void *MRptr, int nmap, int files, char **files,
                     char sepchar, int delta,
                     void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                     void *APPptr);
int MR_map_file_char_add(void *MRptr, int nmap, int files, char **files,
                         char sepchar, int delta,
                         void (*mymap)(int, char *, int, void *KVptr, void *APPptr),
                         void *APPptr, int addflag);
int 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);
int 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);
int MR_reduce(void *MRptr,
              void (*myreduce)(char *, int, char *,
                               int, int *, void *KVptr, void *APPptr),
              void *APPptr);
int MR_scrunch(void *MRptr, int numprocs, char *key, int keybytes);

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

```
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);
void MR_kv_add_kv(void *KVptr, void *KVptr2);
void MR_mr_add_mr(void *MRptr, void *MRptr2);

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);
```

These functions correspond one−to−one with the C++ methods described above, except that for C++ methods with multiple interfaces (e.g. create() or 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 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 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 discussion:

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
```

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.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.clone()
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_list(file,mymap) # file is a file containing a list of filenames
                             # 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,sepchar,delta,mymap)
                             # files is a list of filenames
                             # 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,sepstr,delta,mymap)
                             # files is a list of filenames
                             # mymap is a function called back from the
                             #   library as mymap(itask,str,mr,ptr)
                             # as above, ptr and addflag are optional args

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.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 a if a

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

mapstyle(self,value)       # set the mapstyle to value
verbosity(self,value)      # set the verbosity to value

add(self,key,value)        # single key and value
add_multi_static(self,keys,values) # list of keys and values
                                   # all keys are assumed to be same length
                                   # all values are assumed to be same length
add_multi_dynamic(self,keys,values) # list of keys and values
                                    # each key may be different length
                                    # each value may be different length
mr.add_kv(mr2)             # add the key/values in mr2 to those in mr
```

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

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.

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
- Memory requirements for KeyValue and KeyMultiValue objects
- Hash functions
- Callback functions
- Python overhead
- Error messages

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 MapReduce reduce() operation, and their lengths are also returned to your function.

Keys and values are passed as character pointers to your functions which may need to convert the pointer to an appropriate data type and then correctly interpret the byte string. If the key or value is a variable−length text string, it should typically be terminated by a "0", so that C−library style string functions can 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 in memory such that it is mis−aligned with respect to 8−byte boundaries. The reason is that accessing a mis−aligned double for computaion may be slower.

As an example, 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 int to a double before storing both quantities in a 16−byte value string. Another solution is to create a struct to store the int 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.

Special care should also be taken if your keys and values are heterogeneous. This is because the MR−MPI library packs keys one after the other into one long byte string, and similarly for values; see below for a discussion of memory usage. When values from a KeyValue object are orgnized into a multi−value in a KeyMultiValue object, the values are likewise packed one after the other. Thus even if the components of your keys and values were aligned when you registered them with the KeyValue::add() method, they may not be if they later get mixed in with other keys and values of different lengths. For example, the collapse operation 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 ints, then the values will not be aligned on 4−byte boundaries.

The solution in this case is for your callback function to copy the bytes of a key or value into a local data structure (e.g. using the C memcpy() function) that has the proper alignment. E.g. in the collapse example above, these lines of code:

```
int myvalue;
memcpy(B>offsetsi,sizeof(int));
```

would load the 4 bytes of the Ith value in the multi−value into the local integer "myvalue", where it can safely be used for computation.

Memory requirements for KeyValue and KeyMultiValue objects

KeyValue and KeyMultiValue objects were described in this section. An instance of a MapReduce object contains a single KeyValue object (KV) and a single KeyMultiValue object (KMV), depending on which methods you have invoked.

The memory cost for storing a KV is as follows. The key and value strings are packed into contiguous arrays of bytes. For each key and for each value an integer is also stored, which is the offset into the byte arrays of where that key or value starts. Thus the total memory of a KV is the memory for the key/value data itself plus 2 ints per pair.

Recall that 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 a KMV is as follows. The key and multi−value strings are packed into contiguous arrays of bytes. For each key and for each multi−value an integer is also stored, which is the offset into the byte arrays of where that key or multi−value starts. For each key an additional integer stores the number of values in the associated multi−value. An array of offsets for individual values in each multi−value is also stored.

Thus the total memory of a KMV is the memory for the key/multi−value data itself plus 3 ints per pair plus 1 int per value in the original KV. Note that memory for key data is typically less than for the original KV, since the KMV only stores unique keys. The memory for multi−value data is exactly the same as the value data in the original KV, since all the KV values are in the multi−values.

When the KMV is formed from a KV via a convert operation, additional memory is temporarily required for a hash table used to find duplicate keys. This requires approximately 5 ints per unique key in the KMV.

Note that in parallel, for a KV or KMV, each processor stores the above data for only the fraction of key/value pairs it generated during a map operation or acquired during other operations. If this is imbalanced, one processor may run out of memory before others do.

If you run out of memory when performing a MapReduce operation, the library should print an error message and die. One solution is to run on more processors. Another way to save memory, if possible, is to write your mymap() function so that it combines multiple key/value pairs into fewer or smaller key/value pairs before registering them with the MapReduce object. Calling the compress() method before performing an aggregate() or collate() may also save memory.

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 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 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.

Citations

(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

(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.

Alexandaer Gray, Georgia Tech, http://www.cc.gatech.edu/~agray