Map Reduce MPI

Erik Saule
esaule@uncc.edu

Intro to Parallel Programming

Learning Outcomes

At the end of this lecture, you will be able to

- Explain how hash map relates to Map Reduce
- Give one application that expresses well in Map Reduce
- Express dependency structure of a Map Reduce application
- Write simple programs in MapReduce-MPI

Outline

Why?

- Programming MPI can tedious.
- All communications are explicitly made.
- Load balancing can be a pain for many applications.
- Many applications might not need that level of control.
- Can we get something simpler to program even if it does not do everything?

It is all about hashing!

Word Count

For each word in file:
count[word] ++;

Matrix Multiplication

For i,j, in A:
y[i] += A_{ij} x[j];

Only two important functions

Map

Make a list of (key,value) pairs (out of something)
(in word count):

- foo, 1
- bar, 1
- foo, 2

Reduce

Use all the value of the same key to make (key, value) pairs
(in word count):

- foo, 3
- bar, 1

Note that the key could be change. Or that one key can generate many (key,value) pairs.

It is all about hashing!

Word Count

For each word in file:
count[word] ++;

- Generate all pairs $(word, 1)$
- Reduce by summing gives $(word, wordcount)$

Matrix Multiplication

For i,j, in A:
y[i] += A_{ij} x[j];

- For each i, j, generate a pair $(i, A_{ij}x[j])$
- Reduce by summing gives $(i, y[i])$

Biology example

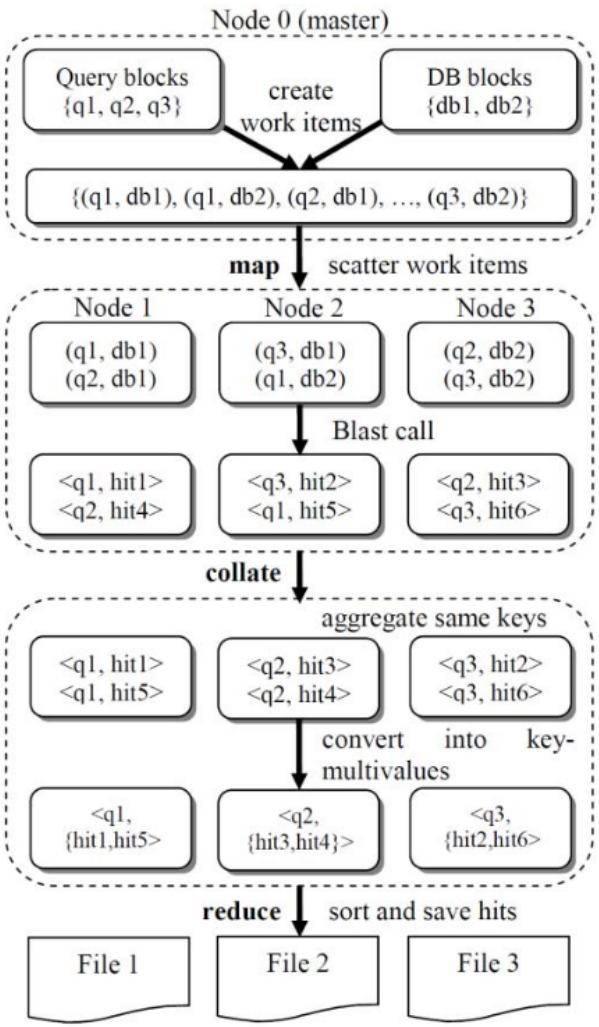

Figure 1. Control flow of the MR-MPI BLAST

Source: Sul, Tovchigrechko, HICOMB 2011

Biology example

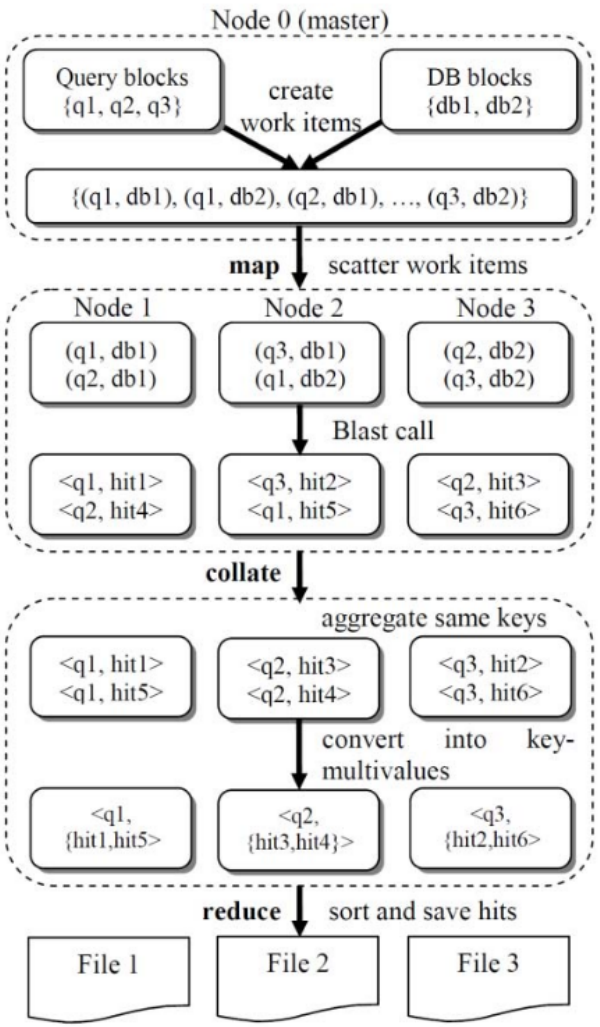

Figure 1. Control flow of the MR-MPI BLAST

Source: Sul, Tovchigrechko, HICOMB 2011

The good

- Seems reasonably straight forward to program
- Gets load balance because distribution is automatic

Biology example

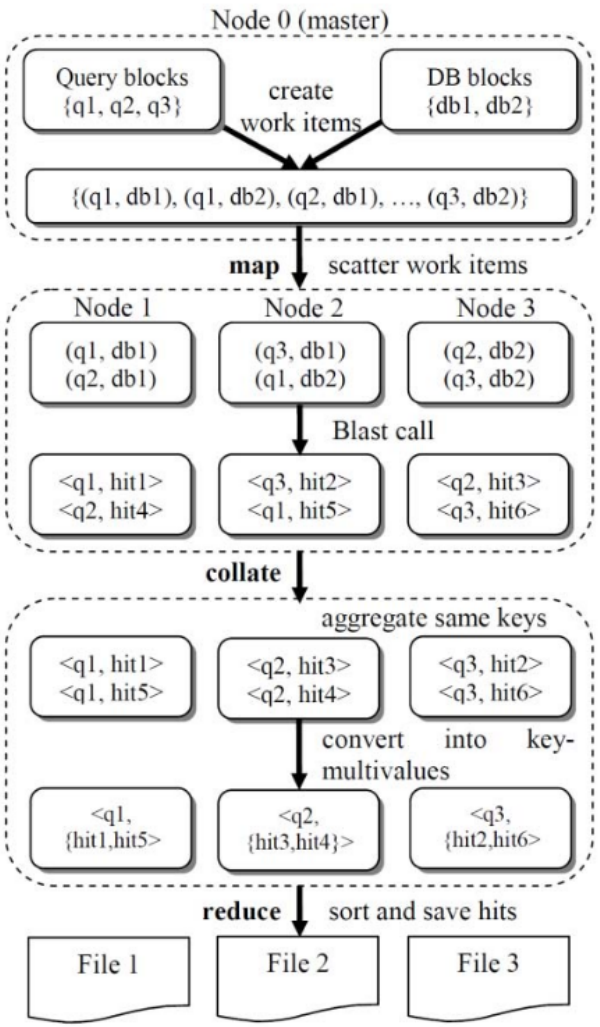

Figure 1. Control flow of the MR-MPI BLAST

Source: Sul, Tovchigrechko, HICOMB 2011

The good

- Seems reasonably straight forward to program
- Gets load balance because distribution is automatic

The bad

- Doesn't collate need an all to all communication?
- Node 2 ends up reading 2 databases
- Databases are kept as files to read to avoid crazy moving on the network
- How do I minimize IO (disk and network) ?

BSP structures only

Bulk Synchronous Parallel is the only form of dependency one can really have in a Map Reduce computation.

- all map tasks are inherently independent from one another.
- reduce can only happen once we have all the values for one key.

Outline

What is it ?

- It is an open source C++ library that provides Map Reduce functions.
- Serial or work with MPI.
- Can be mixed with regular MPI call.
- In-core if possible. Out-of-core otherwise.
- Python wrapper.
- No fault-tolerance (really an MPI issue).
- Written by Karen Devine and Steve Plimpton from Sandia National Labs.
- `http://mapreduce.sandia.gov/`

One-page API

Work on MapReduce objects.

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()	create or add to a KV	allows one MapReduce object to add KV pairs to another	serial	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

KV: Key Value

KMV: Key MultiValue

Typical workflow

- map: makes KV pairs out of *something*.
  - Tons of variant on what *something* can be.
- aggregate: KV pairs with the same keys are now on the same MPI rank.
  - So this communicates over the network.
- convert: KV pairs with the same key become one KMV pair.
  - This operation is local to the node.
- reduce: transform a KMV pair into other KV pairs.
  - There could be one generated KV pair. ("foo", 1, 2, 4) becomes ("foo", 7).
  - There could be no generated KV pair. (for instance to filter out words that appear less than 10 times).
  - There could be more that KV pair generated ("foo", 1, 2, 4) becomes, ("fo", 7) and ("oo", 7) to list all 2 character sub-words.

Map: variant 1: IDs

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

- creates map tasks 0, 1, 2, 3, ..., `nmap-1`
- `nmap` is the total number of map task.
- `mymap` is a user call back functions called for each task. taskID is the first parameter, KeyValue as an output, as a second parameter.
- `ptr` is given to `mymap` as last parameter.

Good if one knows what to do out of a single number.

Map : variant 2 : files

```
uint64_t MapReduce::map(int nstr, char **strings, int
self, int recurse, int readfile, void (*mymap)(int,
char *, KeyValue *, void *), void *ptr)
```

- makes one task per file in a list.
- `mymap` takes a filename (parameter 2) as an input. Other three parameters are the same.
- `strings` specifies the filename to process.
- `self` if 0, takes the filename from MPIrank 0. if 1, each MPIrank gives its own file.
- If a filename is a directory, take all the files in the directory. If `recurse` is 1, take all subdirectories as well.
- If `readfile` is 1. Open each file identified with `strings` and consider each line as a separate filename.

Map: variant 3 and 4 : parsing files

```
uint64_t MapReduce::map(int nmap, int nstr, char
**strings, int recurse, int readfile, char sepchar, int
delta, void (*mymap)(int, char *, int, KeyValue *, void
*), void *ptr)
```

- Use the content of files as tasks.
- The files indicated are opened and chunked in large blocks indicated by `sepchar`.
- `mymap` then takes a string read from the file (and not a filename). Note that the string may contain some `sepchar`.
- `delta` indicates the maximum length of a string that does not contain `sepchar`.

Map : variant 5 : An other Map Reduce object

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

- Use `mr2` as key values given to `mymap` as byte sequences given by the second and fourth parameter and of size given by the third and fifth.

aggregate

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

- Redistribute the KV pairs so that all the Key Value pairs with the same key are on the same MPI rank.
- Use `myhash=NULL` to let MapReduce MPI do the distribution.
- Or specify your own `myhash` to control the mapping.

convert

`uint64_t MapReduce::convert()`

- Goes from KV to KMV

reduce

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

- `myreduce` takes a Key and Multi value and produce KV pairs and is usually written as `void myreduce(char *key, int keybytes, char *multivalue, int nvalues, int *valuebytes, KeyValue *kv, void *ptr)`.
- `key` gives the key as a byte sequence of size `keybytes`.
- there are `nvalues` values that are packed at `multivalue` the one after the other and value `i` is of length `valuebytes[i]`.
- If the data does not fit in MR-MPI pages (default 64MB), then `multivalues` is `NULL` and check the manual for how to read that.

Notes

- One can pass additional information to `map` and `reduce` by using `ptr`.
- One can use local disk by using `self=1` in `map`.
- One can control distribution of tasks to ranks my passing a particular hash function to `aggregate`. (Could be useful for load balancing or if data stored on that node need to be used.)
- MR-MPI goes to disk when data is large.

Interaction with MPI

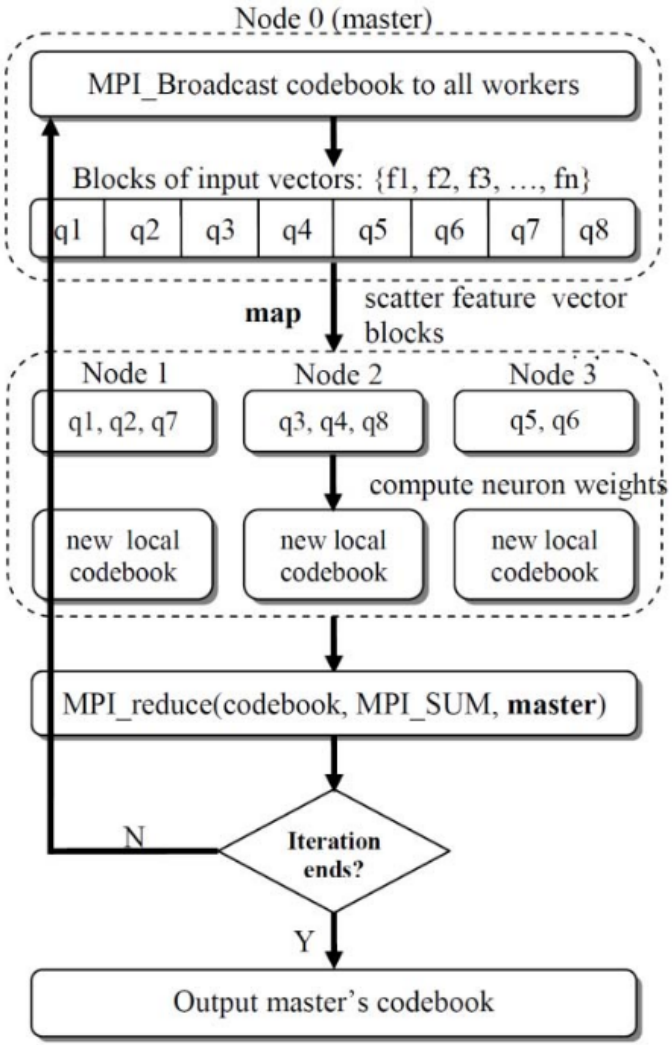

Figure 2. Control flow of MR-MPI Batch SOM

Outline

External

- MapReduce-MPI: `http://mapreduce.sandia.gov/`
- Full API: `http://mapreduce.sandia.gov/doc/Interface_c++.html`

- Seung-Jin Sul and Andrey Tovchigrechko, “Parallelizing BLAST and SOM algorithms with MapReduce-MPI library”, HICOMB 2011
- Plimpton and Devine, ”MapReduce in MPI for Large-Scale Graph Algorithms”, Parallel Computing, 2011