IMPLEMENTING THE MAP REDUCE ALGORITHM USING MPI AND OPENMP

Shramana Chakraborty, Raunaq NandyMajumdar

Introduction:

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a *map* procedure (or method), which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a *reduce* method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

To implement and test this map reduce function we were given 19 files from the Gutenberg project and our aim is to find the word count of the files using the map reduce algorithm and implement it using parallel programming concepts. Then we have to compare the speedup of the parallel implementation with respect to the serial implementation. This project also requires us to scale the number of processes and basically do a hard-scaling which means we need not change the problem size as we increase the number of processors and calculate the efficiency, Karp-Flatt metric for each case and plot it. The report is divided into the following parts. The first part briefly describes the map reduce algorithm in general. The section following it describes our pseudo algorithm followed by a section on **Boost.MPI** library. The section after that consists of the results, plots, analysis, compilation and run command and conclusion.

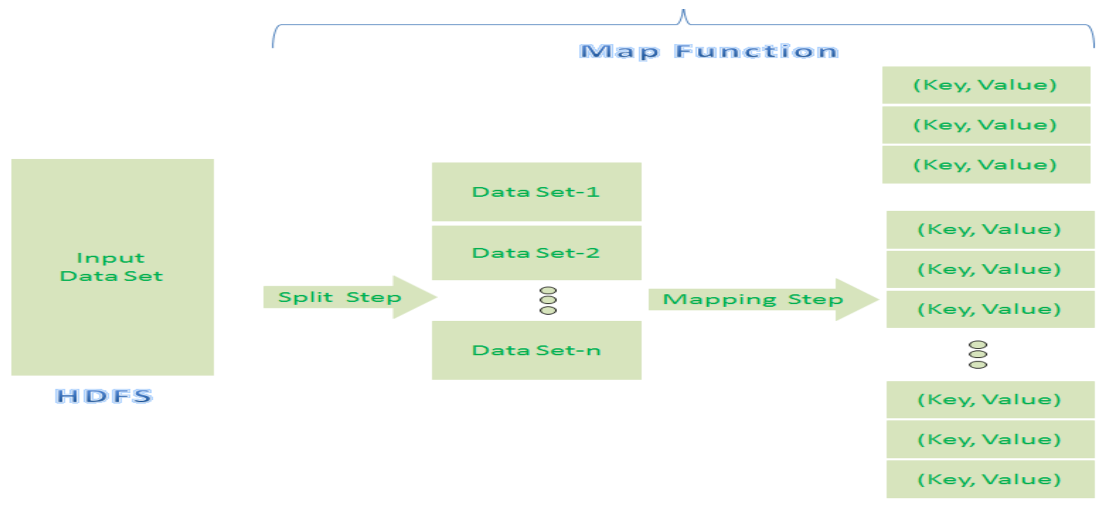
Map Reduce algorithm

MapReduce is a Distributed Data Processing Algorithm, introduced by Google in its MapReduce Tech Paper. MapReduce Algorithm is mainly inspired by Functional Programming model. MapReduce algorithm is mainly useful to process huge amount of data in parallel, reliable and efficient way in cluster environments. MapReduce Algorithm uses the following three main steps namely map function, shuffle function, reduce function.

Map Function is the first step in MapReduce Algorithm. It takes input tasks (say Datasets. I have given only one Dataset in below diagram.) and divides them into smaller sub-tasks. Then perform required computation on each sub-task in parallel. This step performs the following two sub-steps:

- Splitting step: It takes input DataSet from Source and divide into smaller Sub-DataSets.
- Mapping step: It takes those smaller Sub-DataSets and perform required action or computation on each Sub-DataSet.

The output of this Map Function is a set of key and value pairs as <Key, Value>



MapReduce – Mapping Function

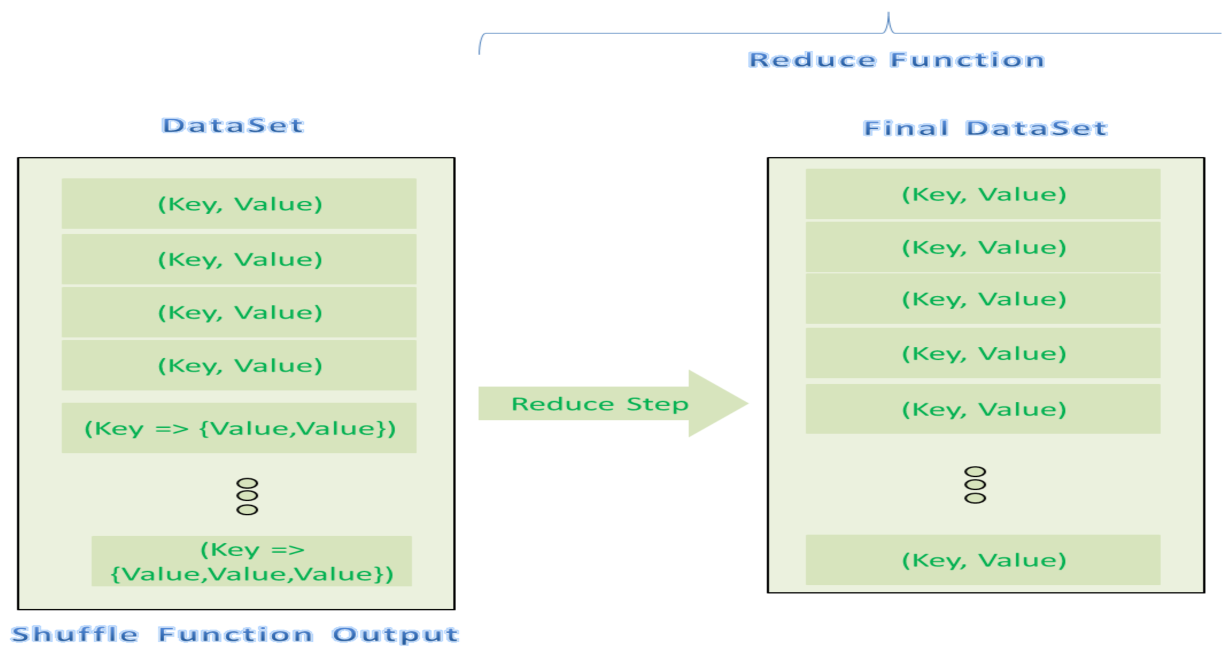
The shuffle part of the function performs 2 steps: merging and sorting. It takes a list of outputs coming from the map function and perform these 2 steps.

- Merging step combines all key-value pairs which have same keys (that is grouping key-value pairs by comparing "Key"). This step returns <Key, List<Value>>.
- Sorting step takes input from Merging step and sort all key-value pairs by using Keys. This step also returns <Key, List<Value>> output but with sorted key-value pairs.

Finally, Shuffle Function returns a list of <Key, List<Value>> sorted pairs to next step.
The next step is the reduce function and it just reduces the values passed onto it by the mapper function. It takes list of <Key, List<Value>> sorted pairs from Shuffle Function and perform reduce operation.
For our job and algorithm we don't need to sort the key value pairs. We just look up words from the read queue and put it into the mapper queue along with count of that word

Map Function Output = List of <Key, Value> Pairs

The reduce function again takes word and count out from the mapper queue and accumulates it into proper reducer queue which can be present on the same process or might be on different process which depends on the hashing function. More on the algorithm in the following discussion.



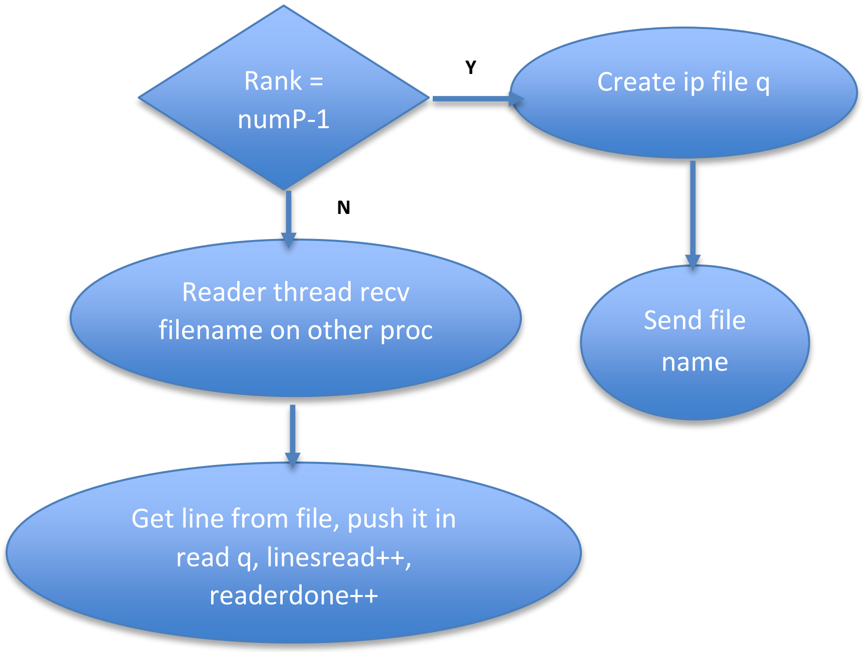
MapReduce - Reduce Function

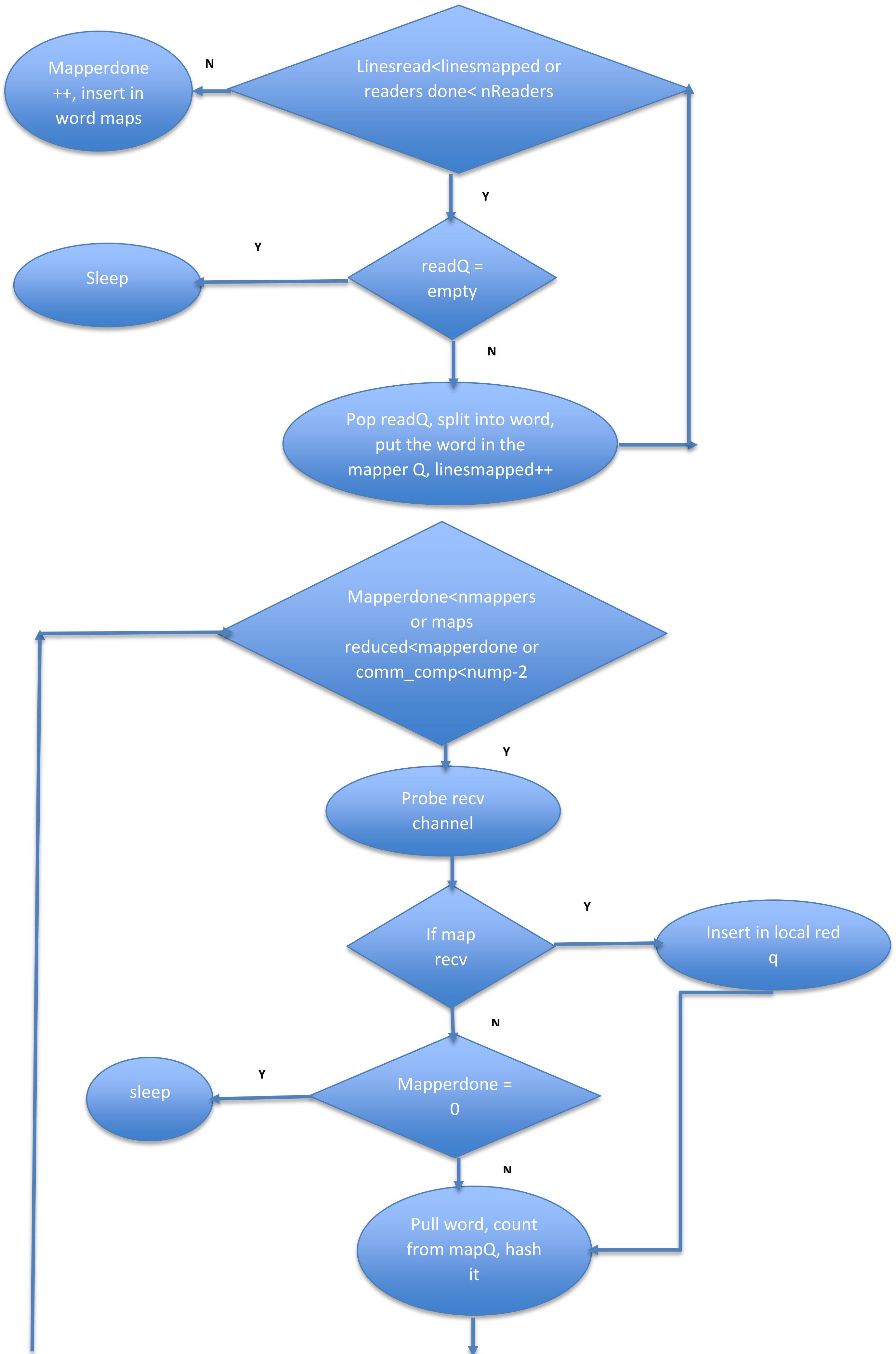
Our map reduce algorithm here adds 2 more layers, one in the front and the other in the back. The reader will pull out lines from the files and the writer will pull out word count pairs and write to a file.

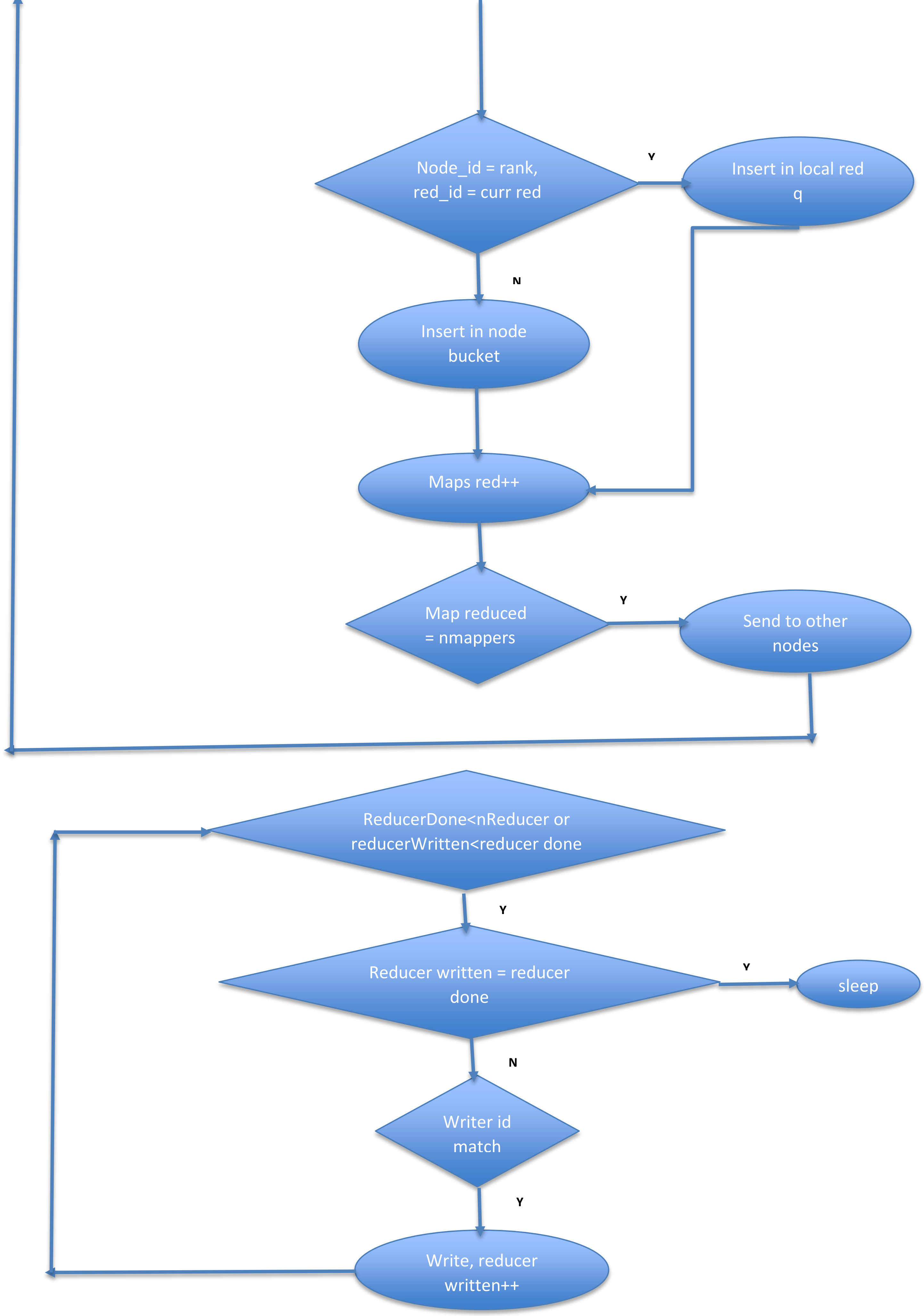
Our algorithm:

In our code everything runs in parallel. We have used C OpenMP and Boost.MPI to write this algorithm. Initially we create a file queue. We have kept one process with one thread (master process) only to distribute the files to reader threads of the different process using sends and receives of Boost.MPI. When a reader thread receives the file name from the master process then it opens the file and extracts out the lines of the file and pushes it into read queue. While testing and debugging we were extracting out word by word but we analyzed that for big files that can be a bottleneck, both in memory and time as the queue length would have been increased and mappers would have waited long time to start mapping the values.
As soon as a line has been mapped the mappers start acting on it. Each element of the read queue is a structure consists of a string which in this case is a line and the length of the string. The mapper thread pulls out the structure from the read queue, the line is split into words, a small punctuation check and processing is done and the word is put into the mapper queue. The mapper queue is implemented using a map where the key is the word and the value is the count. Each mapper has its own map. Whenever the word is passed on to the map it checks if there exists that word in the map, if yes then the count is incremented and if not then it is inserted in the map. The number of threads for mappers is equal to the number of times an omp task is spawned of as a function call to the Mapper funcion from the for loop in the main function. There is absolute possibility that the same word is in different mapper in same or different process. The gathering is to be done by the reducer function which is described below. Whenever a mapper function exits, a variable is incremented which tells us how many mappers are done and the mapper pushes its map into the

global mapper queue so that the reducers can start working on it. The mappers continue to be in function unless number of lines mapped is equal to the final lines read count of the Readers.
For the reducers, the data structure we are using is an array of maps. For the hybrid version our target is that each word will have a unique reducer across processes. When a mapper is done, all the reducers start to act on it and a word is pulled from its queue and passed through a hashing function. We have used Dan Bernstein hashing function as it is widely used and provides balanced hashed map. The hashing function returns us 2 ids, one is process id and other being the reducer id. If the process id matches with the rank of the process and the reducer id matches with the reducer index number then the word is passed to the reducer queue. The reducer queue has nReducers number of maps where nReducers is the number of reducers. The word is checked if it is present in the map or not, if it is then the count is appropriately incremented and if not then it is inserted. What happens if the word does not belong to the current node/process, then we implement another data structure where we create an array of map of maps where array index corresponds to the destination node/process number and each array has map with reducer id as the key and another map of word & its count as the value which needs to be sent across other processes. The word is thus directed to the correct node's correct reducer bucket if and only if, the returned reducer id from the hashing function matches the index of the reducer thread acting on it. This helps us achieves parallelism without the use of locks in the reducer function. After each reducer is done working on each element from the Mapper Queue, a count flag is updated which is checked against total number of mappers. When the mappers mapped by a reducer equals the total number of the mappers, then the map of maps (where the key is the process id and the value is another map of word and counts) is serialized using the boost serialization library and sent to the correct node/process. At the same time, continuous probing is done using iprobe function of Boost.MPI library to check incoming transactions and if some message is available to be received, a Boost.MPI recv is opened and then the incoming map is inserted in the local reducer map. The probing for incoming messages continues until it has received communication from all other processes except itself. Here, each reducer thread communicates with only its counterpart in the other processes. When a reducer is done then a small array is updated which informs which reducer id has been completed.
When the writer function is called it just traverses each local reducer data structure and pulls out the key value pair and it writes to an output file of its own. Once again, when a reducer finishes all the writers starts acting on it. The flow chart of the code for reader, writer, mapper, reducer is shown below.


Rank = numP-1
Y
Create ip file q
N
Reader thread recv filename on other proc
Send file name
Get line from file, push it in read q, linesread++, readerdone++


Mapperdone ++, insert in word maps
N
Linesread<linesmapped or readers done< nReaders
Y
Sleep
Y
readQ = empty
N
Pop readQ, split into word, put the word in the mapper Q, linesmapped++
Mapperdone<nmappers or maps reduced<mapperdone or comm_comp<nump-2
Y
Probe recv channel
If map recv
Y
Insert in local red q
N
sleep
Y
Mapperdone = 0
N
Pull word, count from mapQ, hash it


Node_id = rank,
red_id = curr red
Y
Insert in local red q
N
Insert in node bucket
Maps red++
Map reduced = nmappers
Y
Send to other nodes
ReducerDone<nReducer or reducerWritten<reducer done
Y
Reducer written = reducer done
Y
sleep
N
Writer id match
Y
Write, reducer written++

Boost MPI:

Boost.MPI is a library for message passing in high-performance parallel applications. A Boost.MPI program is one or more processes that can communicate either via sending and receiving individual messages (point-to-point communication) or by coordinating as a group (collective communication). The **Boost.MPI** library provides an alternative C++ interface to MPI that better supports modern C++ development styles, including complete support for user-defined data types and C++ Standard Library types (STL), arbitrary function objects for collective algorithms, and the use of modern C++ library techniques to maintain maximal efficiency.
At present, Boost.MPI supports the majority of functionality in MPI 1.1. The thin abstractions in Boost.MPI allow one to easily combine it with calls to the underlying C MPI library. Boost.MPI currently supports:
Boost Implementation: A Boost.MPI program consists of many cooperating processes (possibly running on different computers) that communicate among themselves by passing messages similar to C MPI. Boost.MPI is a library (as is the lower-level MPI), not a language, so the first step in a Boost.MPI is to create an **mpi::environment** object that initializes the MPI environment and enables communication among the processes similar to MPI_Init(). Here, we have initialized the **mpi::environment** object to support MULTI_THREADED mode similar to MPI_Init_thread(). The available options are: Single (Default), Funneled, Serialized and Multiple (Used in this project).
The following table lists a mapping between C MPI and Boost MPI for point to point communication:

Table 1. For Point-to-point communication

C Function/Constant	Boost.MPI Equivalent
`MPI_ANY_SOURCE`	`any_source`
`MPI_ANY_TAG`	`any_tag`
`MPI_Cancel`	`request::cancel`
`MPI_Get_count`	`status::count`
`MPI_Iprobe`	`communicator::iprobe`
`MPI_Isend`	`communicator::isend`
`MPI_Irecv`	`communicator::irecv`
`MPI_Probe`	`communicator::probe`
`MPI_Recv`	`communicator::recv`
`MPI_Send`	`communicator::send`

Boost.MPI automatically maps C and C++ data types to their MPI equivalents. In general, built-in C++ types (ints, floats, characters, etc.) can be transmitted over MPI directly while for types defined by the standard library (such as std::string or std::vector or std::map etc.) and some types in Boost (such as boost::variant), the **Boost.Serialization** library already contains all of the required serialization code. In these cases, you need only include the appropriate header from the boost/serialization directory.
For types that do not already have a serialization header, you will first need to implement serialization code before the types can be transmitted using Boost.MPI. The class can be made

serializable by making it a friend of **boost::serialization::access** and introducing the templated **serialize()** function.
In addition to this, there are other libraries present in the **Boost.Serialization** library for serialization and deserialization. Two such examples for packing and unpacking binary data are as follows:
The **packed_iarchive** class is used for packing Serializable data types into a buffer using MPI_Pack. The buffers can then be transmitted via MPI. The **packed_oarchive** class is an Archiver (as in the Boost.Serialization library) that unpacks binary data from a buffer received via MPI. It can operate on any Serializable data type and will use the MPI_Unpack function of the underlying MPI implementation to perform deserialization.

Assumptions:

1. We are passing 19 files as input. All the files are named as abc#.txt where # corresponds to 1 … 19
2. We are keeping one processor exclusively for file distribution, for our code it is the last processor. That processor only has one thread for the above mentioned function.
3. The pre-processing of data is not done. This means the words "Barry" and "Barry?" are two different words.
4. As mentioned in pt 2 above when running simulation for 4 processes the value of –n given in the sub file is 5 (1+4) and the same logic applies for 2 and 8 procs also. The extra process is for file distribution thread.
5. The plots for speed-up, efficiency, karp-flatt is done on sub-optimal number of threads due to limitation on the number of nodes. We ran all the configuration for 2 nodes and ppn=20, so we kept the sub-optimal number of threads to 5 and ran simulation till 8 processor as 8x5=40. For 16 processor it is 16x5=80, so we need to request for 4 nodes which forces the job to be queued up in the queue for large amount of time.
6. We are having multiple writer threads writing to different output files, so a single processor will have multiple output files with different word and count written in different files. The naming convention used for the output files is output#@.txt where # is the writer index and @ is the rank of the process its running on.
   For example: with 2 processors, each having 2 writer threads thee will be 4 output files.
7. For the suboptimal performance measurement, we have considered 2 Readers, 3 Mappers, 3 Reducers and 2 Writers.
8. The work size has been kept constant while increasing the number of processors.

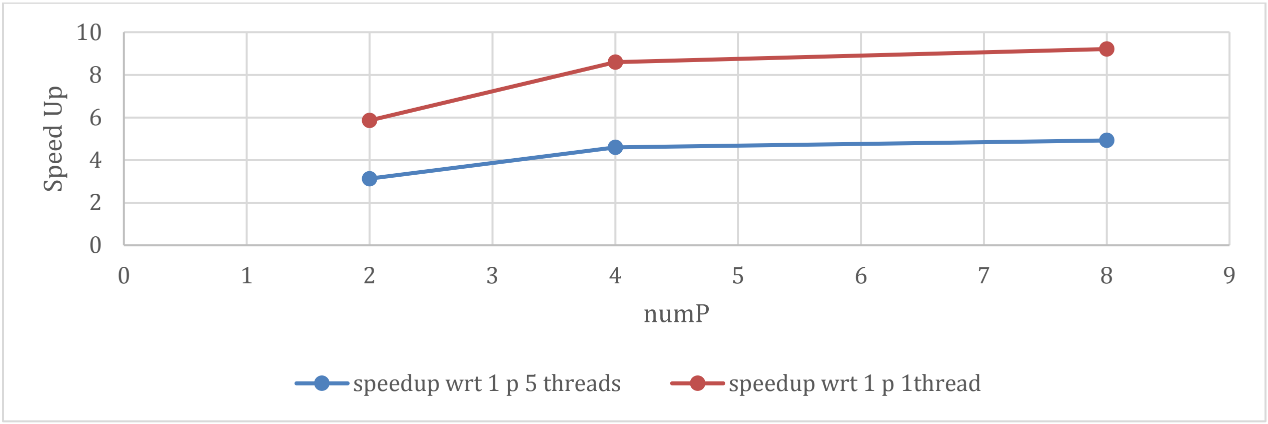
Results:

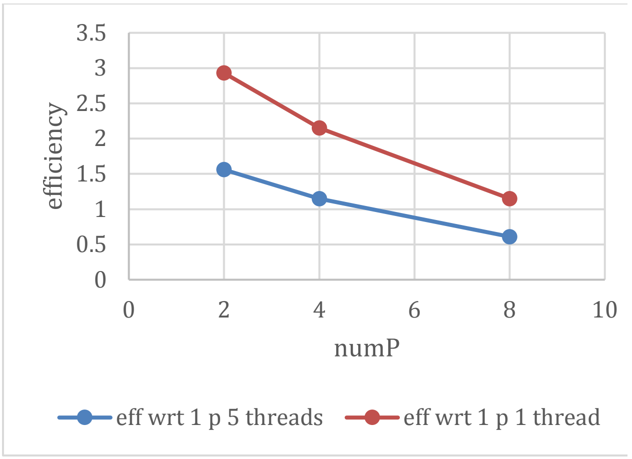
Nump	Threads/ NumP	Time(s)	Speedup	Eff	e
1	5	2.07	-	-	-
2	5	0.66	3.13	1.56	0.28
4	5	0.45	4.6	1.15	0.82
8	5	0.42	4.92	0.61	1.73

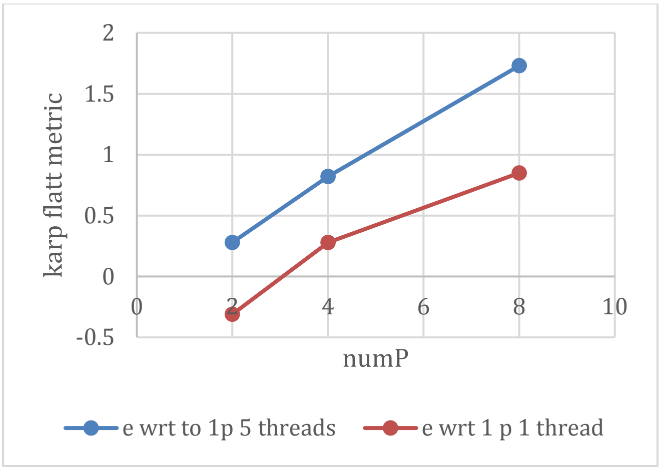
Table2: Comparison with the 1-process-5-thread

Nump	Threads/ NumP	Time(s)	Speedup	Eff	e
1	**1**	**3.87**	**-**	**-**	**-**
2	**5**	**0.66**	**5.86**	**2.93**	**-0.31**
4	**5**	**0.45**	**8.6**	**2.15**	**0.28**
8	**5**	**0.42**	**9.21**	**1.15**	**0.85**

Table3: Comparison with 1-process-1-thread







Analysis:

<u>Suboptimal Performance Measurement:</u>

For both these graphs we measured the performance sub-optimally with only 5 threads per processes. It can be observed from Fig. that the speedup increases with increase in the number of processes however the efficiency decreases and the Karp-Flatt metric (e) increases steadily with the increase in the number of processes. This is because we are keeping the problem size as constant and we are not scaling the work as we are increasing the number of processes to maintain the iso-efficiency. The increase in the Karp-Flatt metric (e) gives us an idea of the increase in the serial overheads and the communication overheads associated with the algorithm with the given problem size. The problem size is a limitation in this case. Also, we are only using 5 threads per processes which is itself suboptimal. So, these graphs only give us an indication.

Optimal Performance Measurement: Due to limited resources and the possibility of jobs getting queued up in the Scholar Machine Queue, we could only test a few cases considering only 2 nodes or 2 processes on scholar machine. Amongst the cases we considered, we found that for 2 nodes or 2 processes, where readers : mappers : reducers : writers is 3:4:8:3 is giving time as 0.38 secs when the number of threads are set to 20, which is approximately 10x speedup with respect to the completely serial code.

Challenges Faced: There was a learning curve involved in learning how to use Boost.MPI library, compile and execute our program. We also faced challenges to figure out how to initialize the MPI hybrid program in multi-threaded mode since multiple threads in each process is communicating to multiple threads across the other processes.

Compiling and running instruction:

Compiling command:

```
mpiicpc -std=c++11 filename.cpp -I /apps/cent7/boost/1.60.0_impi-2017.1.132_intel-17.0.1.132/include -L/apps/cent7/boost/1.60.0_impi-2017.1.132_intel-17.0.1.132/lib -lboost_serialization -lboost_mpi -fopenmp -o object_file -mt_mpi
```

Command to export library path in .sub file

```
export LD_LIBRARY_PATH=/apps/cent7/boost/1.60.0_impi-2017.1.132_intel-17.0.1.132/lib:/apps/cent7/intel/compilers_and_libraries_2017.1.132/linux/compiler/lib/intel64
```

Conclusion: This project gave us a wonderful opportunity to explore the implementation of Map Reduce Algorithm through OpenMP and MPI in distributed systems. We explored a new library called Boost.MPI which facilitates the use of C++ STL objects and helps in Serialization of Complex and User Defined Datatypes for MPI communication. Through the analysis of the results, we observed that the serial and communication overheads increase with the increase in number of processes to maintain iso efficiency. So in order to achieve optimal performance, we need to scale the problem size according to the iso-efficiency relation.