MapReduce
ECE 563 Spring 2017 Default Large Course Project
version 1.0 (1/25/2021)

There is also a video that provides some complementary information for the project.

Projects may be performed by teams of two people. If you would like to do another project, get in touch with me about it.

MapReduce is a programming model that involves two steps. The first, the *map* step, takes an input set I and splits it into N different groups $I_0, I_1, I_2, ..., I_{N-1}$. Each I_p can be thought of as set of pairs, or tuples, *<key,data>*. We also have a function $H(key)$ that returns an int such that for all members $<key_{p,i},data_{p,i}>$, of some I_p, $H(key_{p,i})$ returns the same value. Moreover, for two sets I_p, I_q, $H(key_{p,i})$ will return a different value than $H(key_{q,j})$, for all p, q and i, j. Thus $I_0, I_1, I_2, ..., I_{N-1}$. I_j are equivalence classes. The value of $H(key_{p,i})$ gives the *mapper* that will process $<key_{p,i},data_{p,i}>$.

See the project overview video for a description of *H*.

Stated more simply, *mappers* take data to be reduced, and based on the key for the data, send it to the reducer responsible for those keys. The reducer takes all *<key, data>* pairs with the same key and reduces them to a single *<key, data'>* pair. In our map-reducer that performs a word count, the *key* is a word and the *data* is the count of the number of occurences of that word.

MapReduce has become very popular in part because of its use by Google, but is an old parallel programming model. It is surprisingly general.

To perform a parallel MapReduce, the input is spread across the available processors. Each processor runs one or more instances of *map*, followed by executing one or more instances of *reduce*. Each instance of *map* will form equivalence classes $I_0, I_1, I_2, ..., I_{N-1}$, where some might be empty and send the key/data pairs to the corresponding reducers.

Consider the word counting problem, which can be solved in parallel using MapReduce. Given a list of words, the output should consist of how many times each word appeared in the list (or text). Viewing the input as tuples, the word is the *key*, and the data is the constant 1. A naive *map* function would collect all instances of a word into an equivalence class. Each equivalence class would then be assigned to a reducer process p_r, and process p_r would determine the count of each word, i.e., the word count. Because the same reducer gets all instances of some word, say "cat", from all mappers, that reducer will, by counting all of the *<cat,1>* pairs sent by different mappers, get a total count of the word "cat".

A problem with this scheme is that if "cat" is seen 1000 times by a mapper, it will send a thousand *<cat,1>* pairs to the corresponding reducer. Because communication is expensive, the mapper can *combine <cat,1>* tuples, and only send a single tuple <cat,1000> to the reducer. The reducer is still responsible for combining all the *<cat,cnt>* tuples from different mappers to form a final count.

A more intelligent *map* function would form singleton equivalence classes I_{word}, where the only element is *<word, count>*. The process p_r that reduces I_{word} would receive the I_{word} equivalence classes from all of the *map* functions, and would perform a reduction on the class. In Google terminology, the function that performs this optimization is called a *combiner* and executes on the same process as the map. This is important since its function is to combine many members of an equivalence class into a single member so as to decrease the volume of communicated data sent form the needed between the *map* and *reduce* stages.

A second optimization is to start reducing before all of the mapping is done. Mappers can start sending data to a reducer before it has examined all of its input. This will allow more overlap between mapping and reducing, at the possible cost of sending more <*key,data*> pairs. I.e., <*cat,152*> might be sent, and then, as the mapper find 50 more cats, <*cat,50*> is sent.

What we will program

We will program a map reduce that executes on a distributed memory machine and uses OpenMP on each core to compute the map reduce. The project will be done in three steps:

The OpenMP version and a wordcount map reduce (20% of the project grade)
The MPI version that uses the OpenMP version to perform node-local computation with a wordcount map reduce (20% of the project grade)
Final turn-in. (60% of the project grade)

Details are given below. **Note that even though I use OpenMP you can use Pthreads, Java or other code that supports multithreading to write the shared memory version. Note that if you use Java you will need to use Java isolates to communicate between nodes/processes.**

General information:

Let N_{mpi} be the number of MPI processes and C the number of threads running in each MPI process. Let F_I be the number of data files containing words. Then, $F_I > N_{mpi}*C$,

OpenMP code (i.e. OpenMP code on a node) (you can modify this in your solution, if you want – it's your project!):

There will be several kinds of threads:

Reader threads, which read files and put the data read (or created by self-initialization) into a work queue. For *wordcount* each work item will be a word. For the numerical problem, each entry can be a section of the array that a thread should work on;

Mapper threads, which execute in parallel with Reader threads (at least until the Reader threads finish) and create *combined* records of words. I.e., if there are 2045 instances of "cat" in the files read by the program, the final output of the mapper threads will be a record that looks like <"cat",2045>;

Reducer threads that operate on Reducer work queue entries created by mapper threads and combine (reduce) them to a single record. Thus, for the word "cat", there is potentially a <"cat",$count_i$> record sent by every mapper thread t_i in the system and it will sum all of the *counts*. *For each word there is exactly one Reducer thread in the system that handles it.*

A work queue for each reducer thread. Mapper threads will put work items into this queue. For load balance purposes it is desirable that the range of function *H* that determines which reducer will get a work item be be greater than the number of reducers by some constant factor.

When Mappers have finished mapping they cause their output to be sent to reducer threads. The will do this by placing the work item on the appropriate *Reducer* queue.

You need to have mechanisms to ensure that Mapper threads wait until all Readers have finished before considering themselves complete, i.e. the work queue from which Mapper threads get their work may be empty at some point in time, but may have data at a later point in time because an unfinished Reader thread put data in it.

Mappers will need to put their data on a reducer's work queue based on the key (word) for that data:
As mentioned above, the reducer of a key should be determined by some sort of hash function $g = H(key)$.
All keys that map onto reducer *g* should be added to *g*'s work queue.
Each process can assume it will be receiving data from every other node. This will simplify the communication structure of your program when you go to the MPI version. A node that sends no data should send an "empty" record letting the other process no it will get no data from it.
As each process finishes its reduce work, it should write its results to an output file, close it, and notify the master thread that it is finished so that it can terminate the job, and then terminate itself.

MPI version (you can modify this solution if you want – its your project)**:**

The MPI version will use multiple nodes. Each node will run a copy of the OpenMP code above to perform local computations. A few changes need to be made to the OpenMP process on a node to communicate with the OpenMP processes running on other nodes.

Instead of mappers putting their results onto a reducers work queue, they should put them onto a list to be sent to other nodes. A *sender* thread should be used to send the results of reducers in these lists to the appropriate node.

Each node should have a *receiver* thread that obtains data sent to it by *sender* threads in other nodes
The *receiver* thread for a node will place its received data onto work queues in the node for each reducer.

MPI generally works best when there is a single thread in each process responsible for all communication. I suggest you do this.

Each node will read some portion of the $F_I > N_{mpi}*C$ input files. We could statically define the files each node will process, but this could lead to some nodes getting many big files and other nodes getting many small files. Instead, each node should request a file from a master node which will either send a filename back to the node or an "all done" that indicates that all files have been or are being processed.

Performance data and tuning:

You should collect performance data showing:

What the bottlenecks are in the code. This might involve time Mapper threads are waiting for work from Reader threads, how long I/O takes vs. Mapping (not counting waiting for I/O on mapping) and data to support this other numbers below.

How much load imbalance there is within a node.

How much load imbalance there is across nodes (i.e. the difference in time between the first *map* node is ready to send its data and the latest/last *map* node is ready to send its data to be reduced.

You should experiment with different numbers of Reader threads

Step deliverables:

For the OpenMP version: speedup numbers when using 1, 2, 4, . . . , #cores Mapper and Reader threads;

For the MPI version: speedup numbers when using 1, 2, 4, …, #nodes to run the program, with Mapper and Reader threads for each core on a node (i.e. you don't need to experiment with various numbers of nodes *and* cores

For the final turn-in version:

A paper not longer than ten pages that describes your overall strategy, performance bottlenecks, *Performance numbers* and *implementation positives and negatives* (what you are happy about, what you would like to change.)

A full set of performance numbers either the word-count problem, and scaling by number of nodes, and dataset size, for the matrix multiply problem.

Speedups and *efficiencies* for 2, 4, 8 and 16 processors. *Do the Karp-Flatt analysis* on 2, 4, 8 and 16 processors.

Curves showing the number of Reader threads and performance, and the number of map and reduce threads and performance.

Overall performance of the different parts of the map reduce, and the entire map reduce. For baseline "serial" numbers, use a system with one thread for each of the tasks above.

Performance numbers for different numbers of nodes along with the various speedup metrics (speedup, efficiency and Karp-Flatt).

An explanation of why you are getting the speedups you are getting.

The point distribution will be approximately 40% for a working parallel project with any speedup; 40% for the paper and presentation of your results and explanation of your results, 20% for acceptable speedups or non-trivial explanations of unacceptable speedups.