MapReduce
ECE 563 Spring 2015 Default Large Course Project
version 1.0 (1/30/2015)

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 groups it into *N* equivalence classes $I_0, I_1, I_2, ..., I_{N-1}$. *I* can be thought of as a set of tuples <*key, data*>, and the function *map* maps *I* into the equivalence classes based on the value of *key*. In the second *reduce* step, the equivalence classes are processed, and the set of tuples in an equivalence class I_j are *reduced* into a single value.

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 potentially form equivalence classes $I_0, I_1, I_2, ..., I_{N-1}$.

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 processor *n*, and processor *n* would determine the cardinality of the equivalence class, which would be the word count. A more intelligent *map* function would form singleton equivalence classes I_{word}, where the only element is <*word, count*>. The processor assigned I_{word} to reduce 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 machine 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 that can be performed is to group multiple equivalence classes together to be sent together to the same reducer. Thus, the records for "cat", "dog", "test" and "homework" might be sent by different mappers to the same reducer. This enables all of the to be sent by a single communication operation, improving the efficiency of the communication. The question then becomes, how do we decide which equivalence classes to group together. This decision is done using a *hash function H*. Let's say we will have *R* reducers. Then having a function $0 \leq H(key) \leq R-1$ will group the equivalence classes into *R* groups to be sent to the *R* reducers.

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:

1. The OpenMP version and a wordcount map reduce (15% of the project grade)
2. The MPI version that uses the OpenMP version to perform node-local computation with a wordcount map reduce (15% of the project grade)
3. Final turn-in. (70% 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.**

General information:

1. The text for the map reduce will be distributed across F_I input text files, where $F_I > N_{mpi}*C$, where N_{mpi} is the number of nodes (machines and processes) used by MPI and C is the number of cores on each processor.

OpenMP code (i.e. OpenMP code on a node).

1. There will be four kinds of threads:
   i. 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;
   ii. 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>;
   iii. Reducer threads that operate on 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* and place it on a work queue. *For each word there is exactly one Reducer thread in the system that handles it.*
   iv. Writer threads that take a sum from the work queue and write it to a file.

   You may not need threads for for each of these but only different work queue entries. Thus, Reader and Writer threads run at different times. Mapper and Reducer threads, within a node, can be made to run at different times. These threads can be made to do different tasks by pulling different work out of work queues. This is not mandatory.

2. 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 from 0 to R where $R = k \cdot numMappers$, and k is some constant.
3. 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 Mapper threads get work from may be empty because they have gotten ahead of the Reader threads.
4. Mappers will need to put their data on a reducer's work queue based on the key (word) for that data:
   a. 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.
   b. 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.
5. 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:

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.

1. Instead of mappers putting their results onto a reducers work queue, they should put them onto a list to be sent to other nodes.

2. A *sender* thread should be used to send the results of reducers to an appropriate node.
3. Each node should have a *receiver* thread that obtains data sent to it by *sender* threads in other nodes
4. The *receiver* thread for a node will place its received data onto work queues in the node for each reducer.

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:

1. 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.
2. how much load imbalance there is within a node,
3. 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.
4. You should experiment with different numbers of Reader threads

Step deliverables:

1. **For the OpenMP version:** speedup numbers when using 1, 2, 4, . . . , #cores Mapper and Reader threads;
2. **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
3. **For the final turn-in version:**
   a. A paper not longer than ten pages that describes your overall strategy, performance bottlenecks, performance numbers and implementation positives and negative (what you are happy about, what you would like to change.) You should have
      i. 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. Compute speedups and efficiencies for 2, 4, 8 and 16 processors. Do the Karp-Flatt analysis on 2, 4, 8 and 16 processors. Describe the Iso-efficiency of your program
      ii. Curves showing the number of Reader threads and performance, and the number of map and reduce threads and performance.
      iii. Overall performance of the different parts of the map reduce, and the entire map reduce.
      iv. For baseline "serial" numbers, use a system with one thread for each of the tasks above.
      v. Performance numbers for different numbers of nodes along with various speedup metrics.
      vi. An explanation of why you are getting the speedups you are getting.
   b. I may have a meeting with each group to have you demonstrate your code. This would likely happen during dead week.
   c. The point distribution will be 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.