SANJAY SRIVASTAV 2015A7PS0102P
SAMIP JASANI 2015A7PS0127P

1. SIEVE OF ERATOSTHENES:

The logic used for this question was primarily based on the fact that only the primes less than square root of n are the possible candidates for prime factoring all the numbers less than n. Alternatively, all the numbers less than n that are not prime must be divisible by at least one prime number less than square root of n. So in order to parallelize the code when run on multiple nodes, if each node has the access to these primes less than square root of n, it will find out the other primes that are distributed to its range (assuming each node is distributed to work on some part of the entire range). In order to achieve this, a master-slave type of configuration was followed, where the master schedules and the slave works to remove multiples of number sent to it by master and sends back a part of the primes it was scheduled to work on to the master process. In order to improve the performance the master process also works on a part of the data instead of just delegating.

First step in order to do this is scheduling . Here we assumed that if the there are p processes, each process works on the range n/p while the last process works on the last partition (may be greater that n/p). To achieve this master process first broadcasts the value n and each process can itself calculate its range.

OPTIMIZATIONS DONE FOR SCALABILITY:

- Firstly all even numbers (except 2) were removed from the list to be processed. In fact even numbers are not even added as we know for sure that every alternate number is not prime so we are not using even space for it.

- As we know that the input value may be very large, instead of the master process sending all primes less than square root of n beforehand, what we tried to improve upon was, as soon as the master finds a prime number it sends it to all other processes so that those processes might not waste time till the master computes all primes less than square root of n. This might improve scalability later, as every process is busy all the time and each process might not calculate the primes less than root n by itself and then proceed. Also one other optimization done during this process was that instead of the master sending all the primes less than square root of n one by one, it sends only those primes less than square root of square root of n sequentially. At every step it keeps track of all the multiples of all the primes less than square root of n, that are multiples of this prime. As we know that all the composites less than square root of n will have multiples only between 2 and square root of square root of n, we are now ready with all the primes less than square root of n when the master finds out the last prime less than square root of square root of n and hence it broadcasts everything together from square root of square root of n to square root of n at a time to all other processes, improving scalability for large inputs.
- Character array was maintained for the presence-absence checking of the primes of all the numbers. This optimizes the space utilization. Also the character array was iterated pointer wise rather than array indexing which improves the performance and also the usage aspect (as array indices cannot be long ints).
- For checking of multiples of a particular prime, we considered crossing out the multiples starting from the square of that prime, all composites less than square of this prime must be multiples of some other prime less than this prime, which must have already crossed it out. Also the numbers weren’t iterated each time by one, rather they were iterated incrementally by an amount equal to that prime number starting from the first prime after square of that number.

PERFORMANCE MEASURES:

The final code was tested for multiple inputs ranging from 10^5 to 10^10 on multiple cluster sizes (1,2,4,8).
The following were its results when run on 8 nodes.

16 processes:
10^9: 4.14s
10^10: 38.60s

8 processes:
10^9: 4.35s
10^10: 41.57s

4 processes:
10^9: 4.14s
10^10: 38.60s

2 processes
10^9: 7.15s

1 process:
10^9: 10.81s
10^10:

2. <u>DOCUMENT FREQUENCY INDEX:</u>

For this question the basic data structure considered was a hash table. The types of hash tables were considered - one for indexing the words document wise, second for cumulating the words and the docs in which it appears node wise, and a final global structure which stores the merged structure.

A basic hash function which was giving decent results for some test cases was considered, and separate chaining hash table structure was used for both document wise hash table and the local node wise hash table. For the document wise hash table, a linked list of words was considered for those words that were indexing into same hash entry. Each node has the name of the word and the count of its occurrence. The local index considered has a similar structure except that it also has the number of documents it occurs in and a head pointer to the linked list of documents it appears in. Each document node further has the name of the document along with the count of that word in that document. Each document name has the node number (index number of the node it is working on) appended to it (Ex: if the document name is “test” , it will be saved as “test[i]” where i denoted the node number of the document) . The global structure was taken dynamically and written onto a file in a specific format (similar to that of a python dictionary having a map of word to its documents’ list in all the nodes it occurs).

Each node builds a local index based on the document wise built hash table. Now every process stages its data for the next step that is to merge these local indices. The entire local index in each node is divided in such a way that every process tries to merge all those entries of all the local indices whose entry value is a multiple of the rank of the process.

A similar structure to that of document wise hash table was maintained for stop words.

The building of the linked list of words was in alphabetical order, as the comparisons of the words while adding to local index or global index was easier. Also the document nodes were inserted in order so that the merge process might be easier.

OPTIMIZATIONS FOR SCALABILITY:

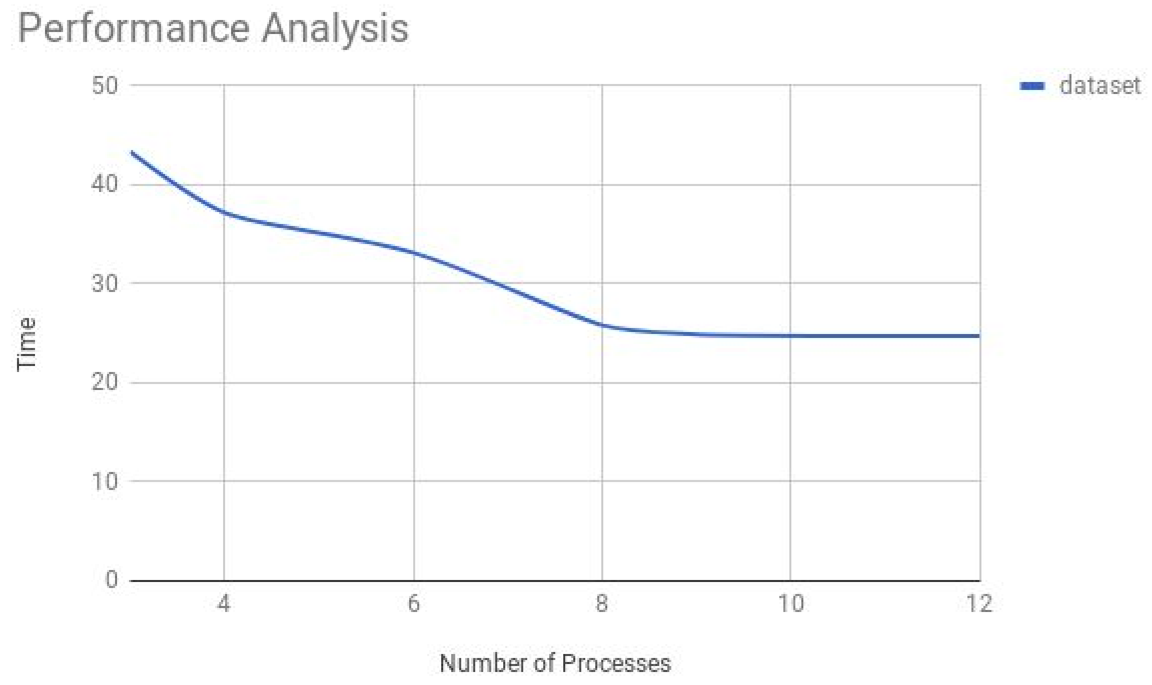
- The global data structure is nowhere maintained in RAM, as it immediately writes it on to a file. Since the final merged entries from the local hash tables are done serially, it saves memory utilization.
- A custom data structure for serializing the linked lists was defined in order to send and receive the data across the processes.
- As multiple processes perform the merge operations resources are utilized and hence parallelization is maintained. This makes the code scalable.
- Instead of using gather statements for every iteration (i.e. gathering data from all p processes at a time) send and receive was used so that the master which is writing, might receive and immediately write it out.

PERFORMANCE MEASURES:

The final code was run on 1.4 GB test data having multiple files. The following were its results when run on 3 nodes.

If number of processes =1 then time taken for calculating global index was 43.36s
If number of processes =3 then time taken for calculating global index was 37.18s
If number of processes =4 then time taken for calculating global index was 33.15s

If number of processes =6 then time taken for calculating global index was 25.82s
If number of processes =8 then time taken for calculating global index was 24.89s
If number of processes =9 then time taken for calculating global index was 24.75s
If number of processes =12 then time taken for calculating global index was 18.83s.


Performance Analysis
dataset
Time
Number of Processes