Zac Brown, Pintu Patel, Priya Mundra
Operating Systems Design
10/6/2013

# Assignment 1: Map Reduce

We were able to implement wordcount and intsort for both threads and processes successfully.

We decided to take a static approach to the parallel programming. Instead of sending each thread one line at a time to compute, and having these threads send the mapped line back to the master thread, our algorithm instead sends entire portions of the file to each thread to compute all at once. We took advantage of the linux commands wc and sed. Our program begins by finding the number of lines in the whole file by calling linecount = wc -l. Each thread receives a chunk of lines equal to the linecount divided by the number of threads. Our algorithm uses the sed 'start_line_of_chunk,end_line_of_chunk' so each thread receives the appropriate chunk of the file to analyze.

Once each thread has its chunk, it creates a map and key value for each word it reads. Because we do not know the number of words in the chunk, we created a LinkedList of these map and keys. Each thread has its own linked lists of map key values, that upon completion of its chunk, merges with the global Linked List.

Once all threads have completed merging with the global, our algorithm implements a quick sort on all map key values to organize into its proper sequence. We then repeat a similar process for the number of reduces by giving each thread a portion of the global list equal to Size_Of_List/Number_Of_Reduces.

Our algorithm reduces each chunk of the global list by comparing the head of the list to the head's next node pointer. If the key's are identical it merges them and therefore increments the value associated with that key. head's next is then deleted from the list and the process continues.

Because we split the List of key values by the number of reduces, there is a chance that the head of one reduced list and the tail of the next reduced list have identical keys. Before merging all reduced lists back into the global list, our algorithm checks these two keys and merges them appropriately.

This algorithm has an overall run time of O(nlogn). It cannot run any faster than the sorting of the global linked list of key values. It would have been smarter to use a HashTable and sort the values in each column of the table.

Our Multiple Process Implementation also takes a static approach. We use the sed command and wc command to get the line count and appropriate chunks for each process. Our shared memory takes a hierarchial approach.

If there are N subprocesses (A,B,C,..,N) each will create 3 shared_memory locations:
/memahA, /memipA, /memqzA,    /memahB, /memipB, /memqzB, ....
Subprocess N will write any word read from its chunk as follows:
words whose first letter is from A-H or a-h go in /memahN
words whose first letter is from I-P or i-p go in /memipN
words whose first letter is from Q-Z or q-z go in /memqzN

This method allows for each sub process to have its own space and eliminates the complications of locks and semaphores. Once each process finishes its computations, the main process reads from memory every shared memory location created by its subprocesses.

The differnence using multiple processes and threads was not to a great extent. The threads were able to complete the mapreduce faster than the processes by 3-4 minutes. The performance in pthread was better because pthreads use the same memory space, for the code and globals, and heap, but it has its own stack. The threads could share global variables and write and read from the same global variables so the overhead is less than the processes. When a new process is fork(), the new process makes a copy of the parent process and attaches itself to a separate memory. The memory between the processes is not shared so each process would write to its memory, then the parent process would read from the child's memory creating an overhead, while in pthreads a  thread could just write into the parent threads memory space. The overhead created in the creation of processes and synchronization of processes, had a impact on the performance. In addition, the threads were easy to implement, then the processes.

The problems that were encounters during the implementation of the mapreduce are how to sort the key,values after using the map function. Our original implementation was to have each thread add words read from its chunk to the linked list in order, but for very large files this is not a smart implementation. We decided to instead use a quick sort to sort the global list when all the threads had completed writing to the list. We also encountered problems sharing memory with the process, we tried using the shm_open and mmap, but each process would write to the mmap char*ptr, but would delete it when it exited so we the parent would get an empty memory space when it tried to read from the mmap char*ptr. We also tried to use shm_get to get a shared memory segment with a unique key, and pass the key. So when the children open the same memory segment with the global unique key, and attach them self somewhere in the memory. Then after attaching they could write to the memory, but only one process could write to the segment at once. This was also unsuccessful because we were not able to retrieve the written data from the shared memory segment after the child had written to it.