PA1 Report.pdf

Team Members: Raj Patel, Derek Sagun, Luiz Rivera

Our map and reduce framework works by initializing a large piece of shared memory and writing to, reading from, and reorganizing data in blocks. We avoided some synchronization by dividing shared memory into a number of blocks depending on the number of maps. This way, each thread or process writes into its own designated block assigned by ID, and no mutex needs to be used while the threads or processes are writing to shared memory. However, we still do use a barrier /wait to ensure that all of the map threads/processes finish before the reduce threads/processes start running. Our mapper function parses the input file and divides up the input buffer and sends a part to each worker node. Each worker node is also given the specific map function it needs, and it writes pairs in an array to shared memory; pairs consist of a char[] key and an int value. After all the pairs are written to shared memory, we call sort on each pairArray for each block, which sorts the pairs within each block using strcmp. Next, we use a writeBackOrganize function, which reorganizes shared memory based on the number of reduces, so that each reduce node can work on one block. Next, reducer is called, which serves a different function depending on wordcount or integer sort. For the integer sort, our reduce does the minimal task of setting every single key to “”. This way, when we call our method to write to the file, we can simply write both the key and the pair for both wordcount and integer sort. Our wordcount integer sort serves to combine words that are the same and increment the count value as necessary. After all the reduces run, the output can then be written to the file by iterating through every pair array and writing each key and value to the output file.

We evaluated performance between process and thread-based implementations of our framework with the use of a time function that outputs the time each map and reduce call took. We were surprised to learn that processes were actually around 2 to 3 times faster than threads. Since processes need more of an overhead to create, it was surprising to us that they ran more faster than threads. Sacrificing the overhead for processes is worth it if speed is more of a priority than space.

Some difficulties we encountered during the project was figuring out how incorporate the key value structure for the integer sort problem. For the wordcount it was simple since the key was the string itself, and the value was the number of times it appeared. For integer sort, we made the key the string version of the integer, and the value the integer. We then had to come up with a way to use strcmp to sort the integers, since we wanted our sort function to be generalized. We did this by rewriting the ints with a large number of bytes, such as 20. So, for instance, the number 100 would have the key “000000…100” and its value would be 100. Although this does create some space problems, it does make our sorting method more general so that we used the same sort for both wordcount and integer sort. Another issue we came across was segfaults when we tried running our working code for processes. The issue was that in our original pair struct, we had a char *, but this did not work because each individual process was malloc’ing on its own heap, and this was not persistent across all processes. So then, we had to go back and change our code to use a static char array[] and strcpy instead of malloc, and after doing this our code worked properly. Another problem we had was figuring out how to terminate our pairArrays, and we did this by adding a struct at the end of each block with a value of -1 and key “end”. This way, we could iterate through the pairArrays for each block and terminate as soon as we got to the pair that had a value of -1. This was particularly helpful because we could iterate over each pair array and print out the contents of shared memory in its entirety to debug.