**Performance Report**

**External Memory Sort – 2PMMS**

**Timing disk\_sort:**

I ran the disk\_sort program on the original “Arizona State University Twitter Data Set”, and used the timing command to time evaluate the performance. I used the optimal block size of 4096 bytes. I ran this part of the experiment 5 times, and took the average of the results. Here is the performance of my disk\_sort program:

(Average) Total elapsed time: 1 min 27 sec.

(Average) Maximum resident set size: 207604 KB

Note that the maximum resident set size is fairly close to the allocated memory of 200MB (=200000KB).

**Performance and RAM (buffer) Size**

I performed this part of the experiment on the concatenated input file which is more than 1.5 times larger than the total main memory on the machine. I tried memory size of 1/2, 1/4, 1/8, 1/16, 1/32, and 1/64 of the original 200MB, and the program cannot perform the two-pass algorithm when it reaches 1/128 of the original 200MB.

However, since it is discovered that qsort actually uses more memory than the allocated memory, we are instructed to split the partition into two. For example, when we allocate 200MB to RAM, each sublist/run would actually have size of 100MB instead of 200MB. Thus the two-pass algorithm would not work when we get to 1/64 of the original 200MB.

Once again, I ran each memory size multiple times and took the average of the results, and here are the results:

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory size** | **Fraction of original 200MB:** | **Total elapsed time (mm:ss)** | **Max. resident set size (KB)** |
| 200MB | 1 | 3:14 | 207704 |
| 100MB | ½ | 3:18 | 105272 |
| 50MB | ¼ | 3:33 | 53844 |
| 25MB | 1/8 | 4:18 | 28372 |
| 12.5MB | 1/16 | 4:22 | 15496 |
| 6.25MB | 1/32 | 5:54 | 9084 |

Notice that for the max resident set size, all the results are fairly closed to the allocated RAM memory; the max resident set size are all around 2MB away from the allocated memory.

As for the total elapsed time, theoretically it should not depend on the number of runs K. This is seen in the first 3 cases of the experiment, the total elapsed time for 200MB, 100MB, and 50MB are all fairly close (~3:20ish). However, one can observe an increasing trend in the total elapsed time as memory size increases. I suspect that this is due to the structure of the program disk\_sort. In phase 1 of disk\_sort, my program will write the sorted sublists into different .dat files. Those sublists would be opened (and closed) every time the corresponding input buffer needs to be refilled. When the RAM memory size is small, there will be more sublist (smaller.dat files) and more run in phase 1. Thus in phase 2, the input buffer would be smaller as well. In other words, more fopen and fclose are called, thus decreasing the time efficiency of the program.

In the first 3 test cases, the total elapsed time is around the same because the number of sublists is still small. However when it does to memorize size of 25MB, there would be 160+ .dat files, and each time we have to fseek and read a relatively small number of elements into the input buffer. Thus this explains why there is a different in performance.

**Performance against Unix Sort**

This is the result when I tried to time the unix sort:

|  |  |  |
| --- | --- | --- |
| **Sort on original data** | **Total elapsed time** | **Max. resident set size (KB)** |
| My disk\_sort | 1:27 | 207640 |
| Unix sort | 2:08 | 345920 |

From the result, one can observe that the unix merge sort is slightly slower than the 2PMMS. After doing some research on the Unix sort command, I found that Unix uses “External R-Way merge” to sort through large data, which is similar to 2PMMS. This explains why the total elapsed time is still very close. The difference might come from instantaneous changes in CPU or because of the different in input file (disk\_sort takes in a ~600MB .dat file while unix sort takes in a 1GB+ .csv file). Unix sort might have to do more processing.

As for maximum resident set size, there is a difference of around 100MB in the amount of memory used. Resident set size is the amount of memory that belongs to a process that is held in the RAM. This can possible be due to the fact that we have full specific control over memory usage in our 2PMMS program, however when we run the unix sort, we have no control over how much RAM is used. Another reason is possibly due to the difference in input format (same reason as the time difference). The unix sort takes in anything in general while my disk\_sort program takes in something very specific.

**Summary**

Through this part of the project, I got more familiar with the 2PMMS algorithm. I think often times it’s easily to listen to information in lecture, but implementing them is a totally different story. The project definitely filled in the details of 2PMMS that we didn’t learn in class, such as using a heap for phase 2, or how the input buffers are refilled each time. It is also an interesting experiment to compare 2PMMS with the unix built-in sort. It’s amazing to see my implementation is actually faster than the built-in sort (for this specific file).