PERFORMANCE REPORT: MERGE SORT OF FILE (>1GB) USING BASH COMMAND SORT

Testing has been done using file size of 1.1 GB.
Test Machine – AMD A8 6410 (4 cores, 2 Ghz), 3.4 GB usable RAM. Running Ubuntu 15.04

I) Profiling Normal bash sort

Performance counter stats for 'sort -n -T /media/susobhan/EADA79CADA799413/Code/ random.txt':

```
      705835.908450      task-clock (msec)         #    1.455 CPUs utilized
          12,36,201      context-switches          #    0.002 M/sec
             24,215      cpu-migrations            #    0.034 K/sec
             36,397      page-faults               #    0.052 K/sec
13,73,18,11,43,584      cycles                    #    1.945 GHz
[49.95%]
                   0      stalled-cycles-frontend   #    0.00% frontend cycles idle    [49.93%]
                   0      stalled-cycles-backend    #    0.00% backend cycles idle     [49.97%]
 9,68,05,56,10,885      instructions              #    0.70  insns per cycle         [50.06%]
 2,05,90,16,88,296      branches                  #  291.713 M/sec
[50.08%]
    4,06,64,63,792      branch-misses             #    1.97% of all branches
[50.04%]

      485.257774217 seconds time elapsed
```

II) Profiling Custom bash script

Custom Bash Script is made to split the file until a certain threshold is reached, beyond which splitting is stopped and bash sort command is used.

In my tests, data has been collected for thresholds of 100 MB, 200 MB, 300 MB and 600 MB. The profiling results are as follows:

a) 100 MB threshold

Performance counter stats for './mergesortscript.sh random.txt sorted.txt':

```
      673754.944672      task-clock (msec)         #    1.037 CPUs utilized
          30,25,590      context-switches          #    0.004 M/sec
             31,464      cpu-migrations            #    0.047 K/sec
          16,16,105      page-faults               #    0.002 M/sec
13,04,84,12,00,832      cycles                    #    1.937 GHz
[50.03%]
                   0      stalled-cycles-frontend   #    0.00% frontend cycles idle    [50.17%]
                   0      stalled-cycles-backend    #    0.00% backend cycles idle     [50.20%]
11,11,33,46,97,832      instructions              #    0.85  insns per cycle         [50.06%]
 2,37,34,21,05,036      branches                  #  352.268 M/sec
[49.93%]
```

```
    4,63,66,53,788      branch-misses             #     1.95% of all branches
[49.91%]

     649.468859235 seconds time elapsed
```

b) 200 MB threshold

```
 Performance counter stats for './mergesortscript.sh random.txt sorted.txt':

     709126.317919      task-clock (msec)         #     1.107 CPUs utilized
        27,75,327       context-switches          #     0.004 M/sec
           39,178       cpu-migrations            #     0.055 K/sec
         9,03,429       page-faults               #     0.001 M/sec
13,66,77,16,79,144      cycles                    #     1.927 GHz
[50.01%]
                 0      stalled-cycles-frontend   #     0.00% frontend cycles idle    [50.19%]
                 0      stalled-cycles-backend    #     0.00% backend  cycles idle    [50.14%]
10,64,76,32,40,547      instructions              #     0.78  insns per cycle         [50.08%]
 2,27,45,85,82,788      branches                  #   320.759 M/sec
[49.88%]
    4,57,13,17,295      branch-misses             #     2.01% of all branches
[49.95%]

     640.607647192 seconds time elapsed
```

c) 300 MB threshold

```
 Performance counter stats for './mergesortscript.sh random.txt sorted.txt':

     693201.749195      task-clock (msec)         #     1.037 CPUs utilized
        26,37,512       context-switches          #     0.004 M/sec
           49,932       cpu-migrations            #     0.072 K/sec
         7,52,711       page-faults               #     0.001 M/sec
13,50,43,24,14,447      cycles                    #     1.948 GHz
[50.05%]
                 0      stalled-cycles-frontend   #     0.00% frontend cycles idle    [50.15%]
                 0      stalled-cycles-backend    #     0.00% backend  cycles idle    [50.05%]
10,38,41,52,05,272      instructions              #     0.77  insns per cycle         [50.02%]
 2,21,77,64,57,943      branches                  #   319.931 M/sec
[49.91%]
    4,54,17,71,703      branch-misses             #     2.05% of all branches
[50.02%]

     668.397767750 seconds time elapsed
```

d) 600 MB threshold

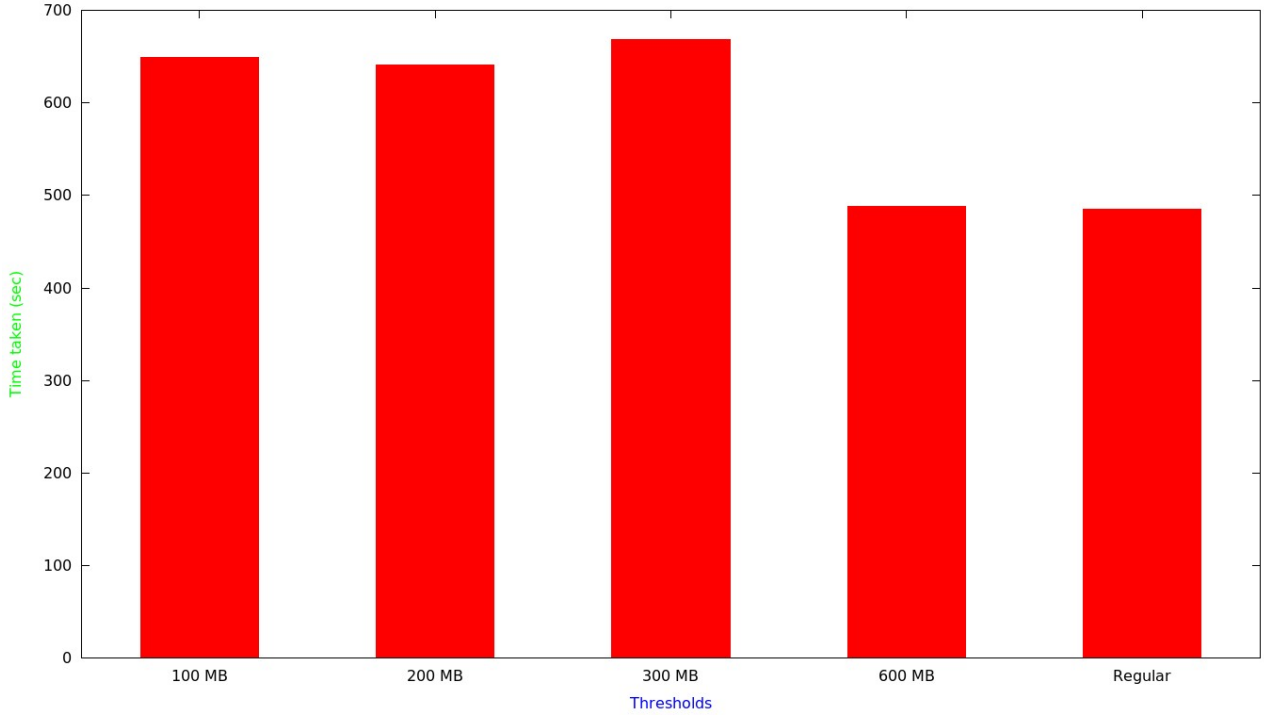
```
 Performance counter stats for './mergesortscript.sh random.txt sorted.txt':

      652837.001958      task-clock (msec)         #    1.336 CPUs utilized
         24,07,096       context-switches          #    0.004 M/sec
            26,853       cpu-migrations            #    0.041 K/sec
          1,30,940       page-faults               #    0.201 K/sec
12,82,14,41,29,204       cycles                    #    1.964 GHz
[49.87%]
                 0       stalled-cycles-frontend   #    0.00% frontend cycles idle     [50.02%]
                 0       stalled-cycles-backend    #    0.00% backend  cycles idle     [50.08%]
10,22,20,34,90,268       instructions              #    0.80  insns per cycle          [50.18%]
 2,17,98,43,96,916       branches                  #  333.903 M/sec
[50.04%]
     4,28,26,50,891      branch-misses             #    1.96% of all branches
[49.98%]

      488.476233113 seconds time elapsed
```

Analysis:

Overall the normal sort performed better than the merge sort on my system with varying thresholds. Only when the number of splits was low, the performance tended to reach normal sort times. The reason being that with a lower threshold one has to keep splitting more and more files and do I/O on the hard disk. Most of the processing is done at first to split the files to the threshold limit. This is where time is lost and splitting overhead is accounted for, and this version of merge sort performs badly.

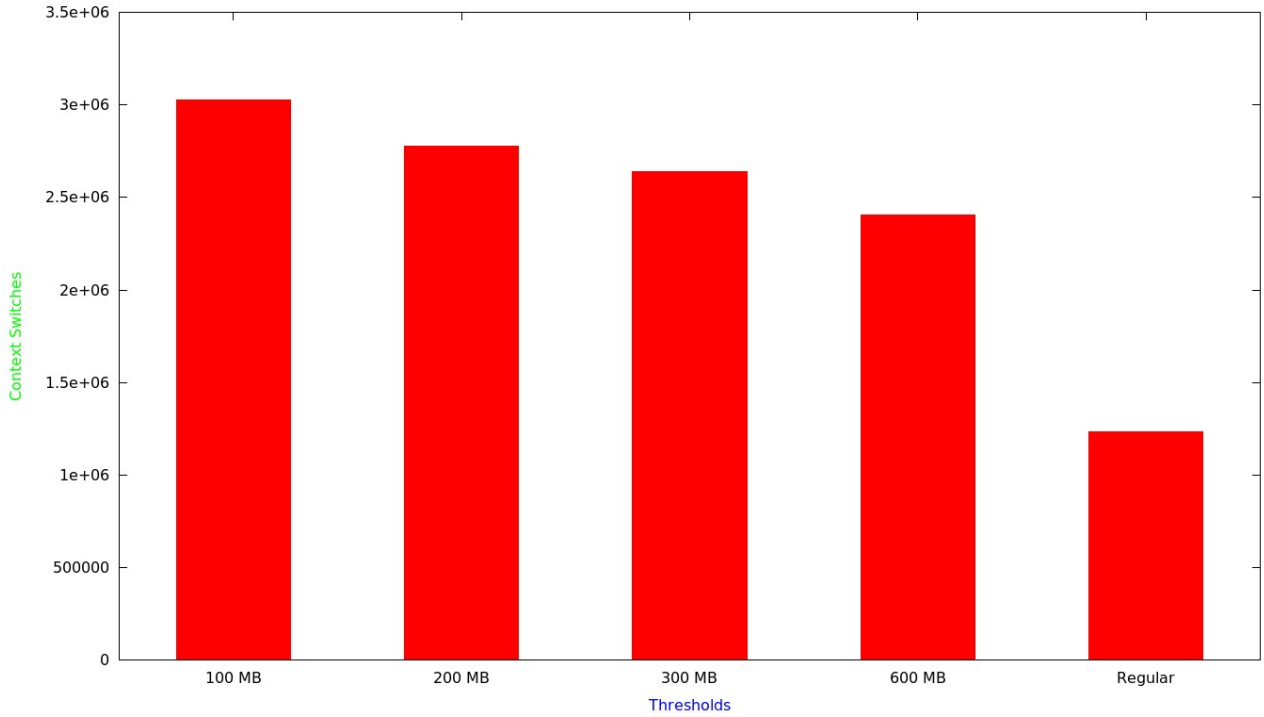
In depth analysis is as follows:

I) Time Taken:



The results show us that the normal sort provided the best result and in terms of splitting, the best was the threshold which split the file into 2 parts (600 MB). This was because of time overhead accounted for writing to the disk. The other split thresholds took almost the same amount of time to complete and were considerably more compared to regular sort (I/O and read from HDD overhead).

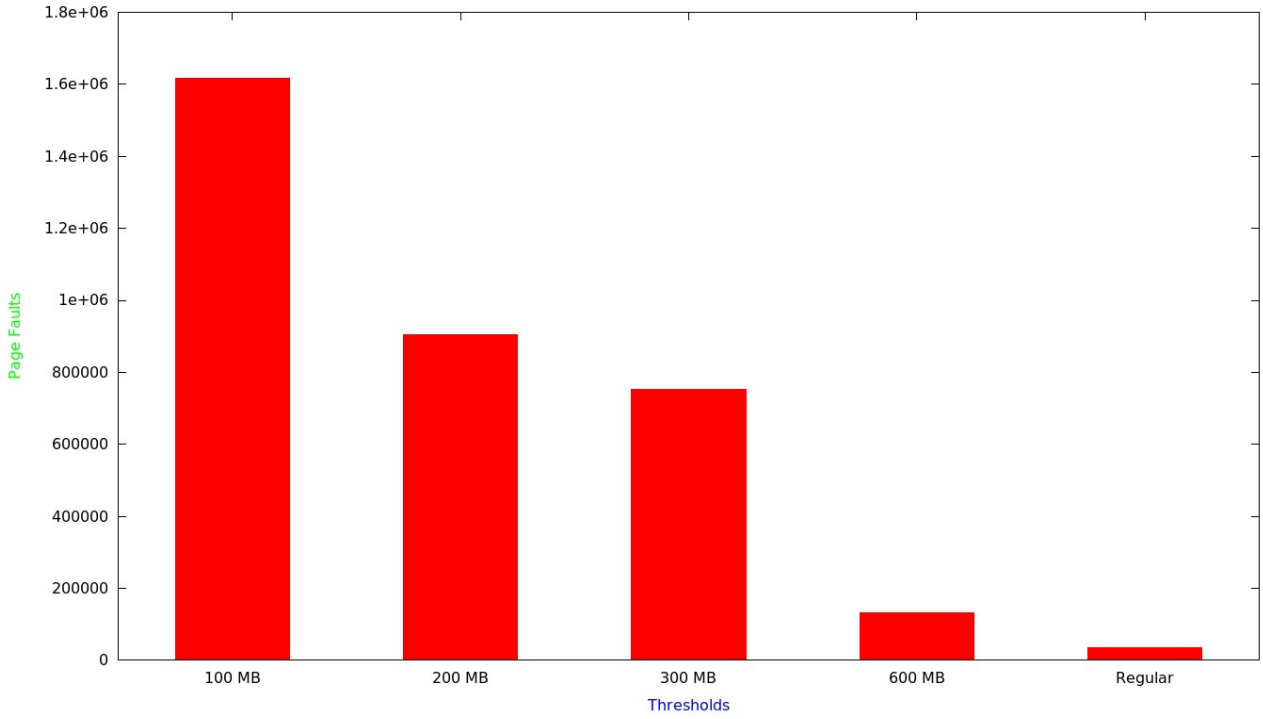
II) Context-Switches



We see that the number of context-switches is increasing in terms of lower thresholds. This is because more number of sorts are called on lower file sizes, thus more concurrent processes. And switching in between concurrent processes accounts for more number of context switches. The regular bash sort call has the least number of context switches as sort is invoked just once and thus the minimum. Others invoke sort more than once and switching between them accounts for more context switches.

III) Page Faults:

We find that more page-faults are accounted for lower thresholds. This is because of the splitting and individual sorts running on smaller number of inputs. As the data is splitted multiple times in case of lowering thresholds, the page fault increases as the numbers being worked on keep on changing as more splitting is done. So more memory accesses are required and more page faults are encountered.


1.8e+06
1.6e+06
1.4e+06
1.2e+06
1e+06
800000
600000
400000
200000
0
Page Faults
100 MB
200 MB
300 MB
600 MB
Regular
Thresholds