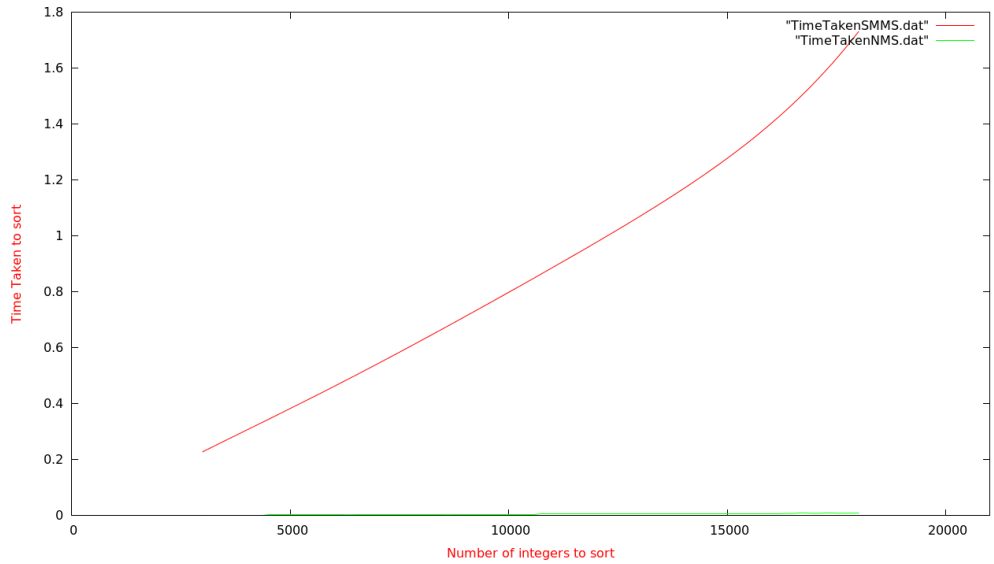
REPORT – QUESTION 3

This question asked us to compare the performance of normal merge sort (NMS) against another version of merge sort which creates a separate process for each partition and uses shared memory between the forked processes (SMMS – Shared Memory Merge Sort).

The input was a text file which contained randomly generated numbers.

The number of integers in the different input files were 3000,6000,9000,12000,15000 and 18000. The tests were stopped at 18000 integers as beyond this number, the SMMS was unable to fork new processes.

In each instance the NMS outperformed the SMMS by a huge margin as one can see from the charts.

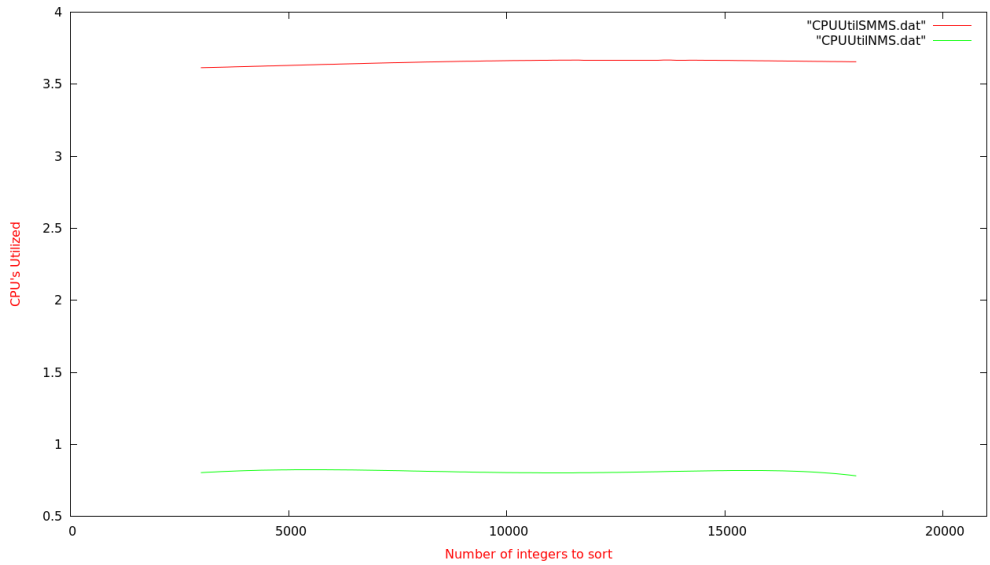


The Time Taken Chart (Notice that the green line corresponding to Normal Merge sort is not even visible – This happens in many other graphs too)

The main reason for this is the forking that we perform for each partition in SMMS.
This creates a lot of child processes for huge input which results in sluggish performance.

A large number of child processes, though they are using shared memory, implies
a) More Context Switches
b) More CPU Migrations
c) More Page Faults

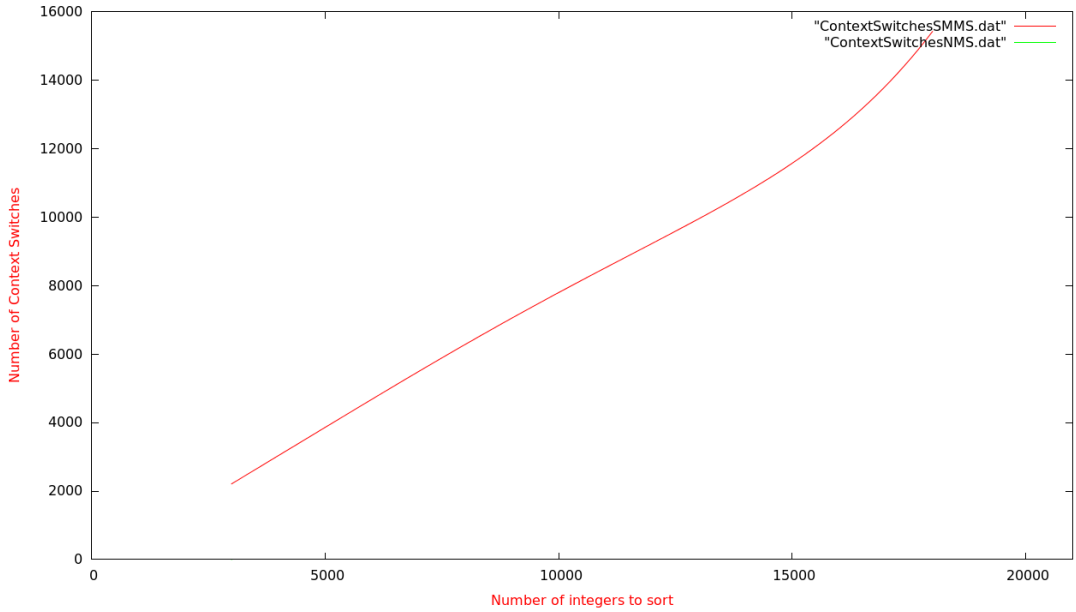
It can be seen from the respective charts that SMMS has a huge number of context-switches, cpu-migrations and page-faults as compared to NMS.



The CPU Utilization chart A lot of processes have been forked which leads to extensive CPU Utilization without yielding desired results. If we optimize SMMS, we will fully utilize the CPU and get better results than NMS which cannot utilize all the cores.

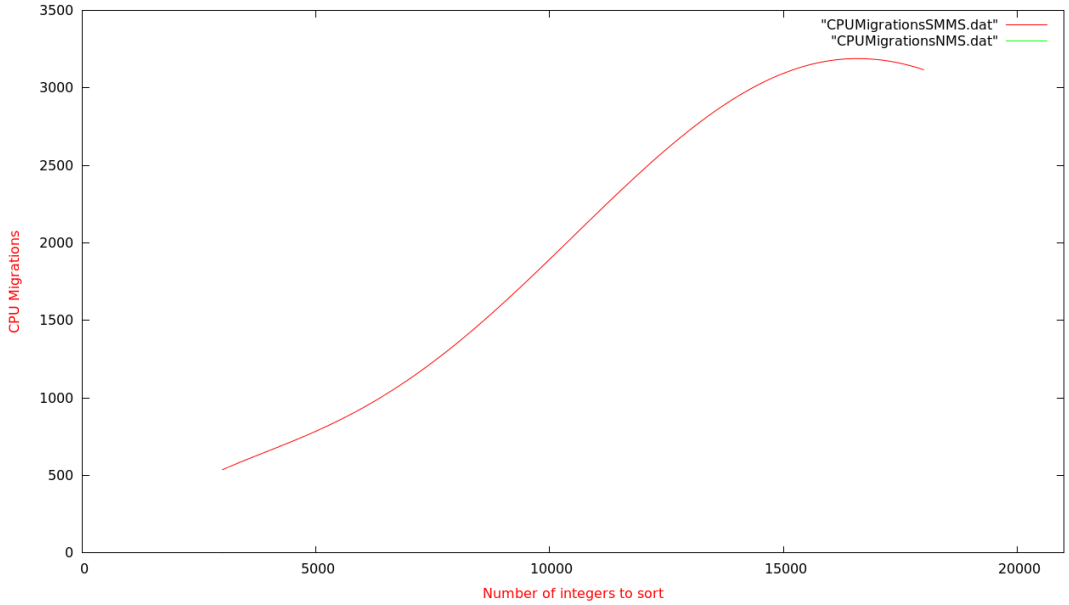
More number of processes leads to a lot of time consumed in switching between processes.

It is also observed that though SMMS utilizes more CPU than NMS, it still is very slow in comparison due to the time consumed for switching between processes.

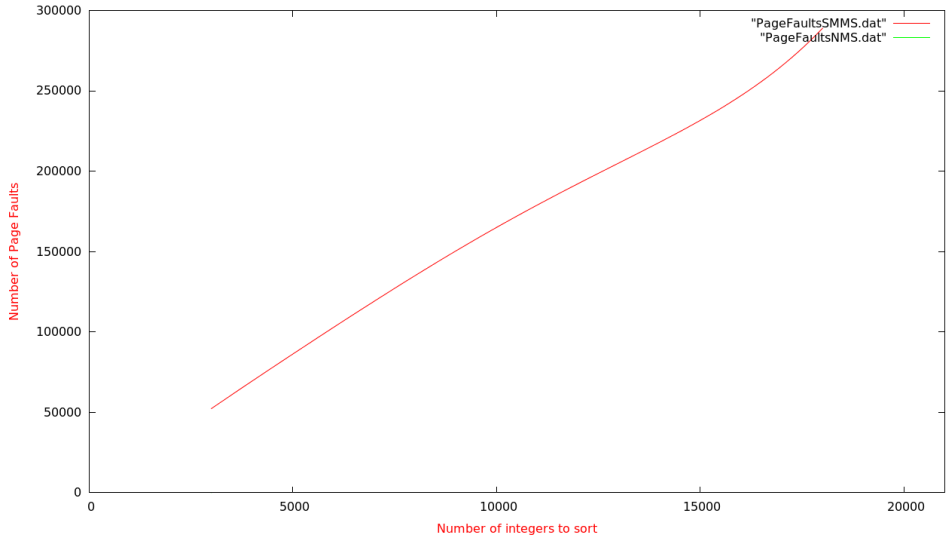


The context-switches chart If we optimize SMMS, we may still have a greater number of context-switch than NMS but it wouldn't be as large as above as there would be a cap on the number of processes.

On creating a new process for each partition, we can sort that partition in parallel with other partitions. But this gain is overwhelmingly negated by the huge number of processes created which though executed in parallel, have to be context switched in and out between each other due to the limitation of the number of CPUs. There are also a large number of CPU migrations in each case of the input in SMMS as compared to NMS.



The CPU Migrations chart If we could somehow minimize this, the performance of SMMS would be a lot better. Also notice the slight flattening of the curve towards the higher ranges for SMMS. It would be tough to attribute a reason to this but maybe this is because the CPU is fully utilized at the higher range and the OS doesn't get any perceived benefits on migrating processes even more than what is currently being done, from one core to the other, due to the full utilization of all the cores already.



The Page Faults chart Though we use shared memory, SMMS has a lot of page-faults because Merge Sort is not an In Place algorithm. So we would be requiring extra space in the memory for sorting and this leads to more Page Faults as different processes sort different segments.

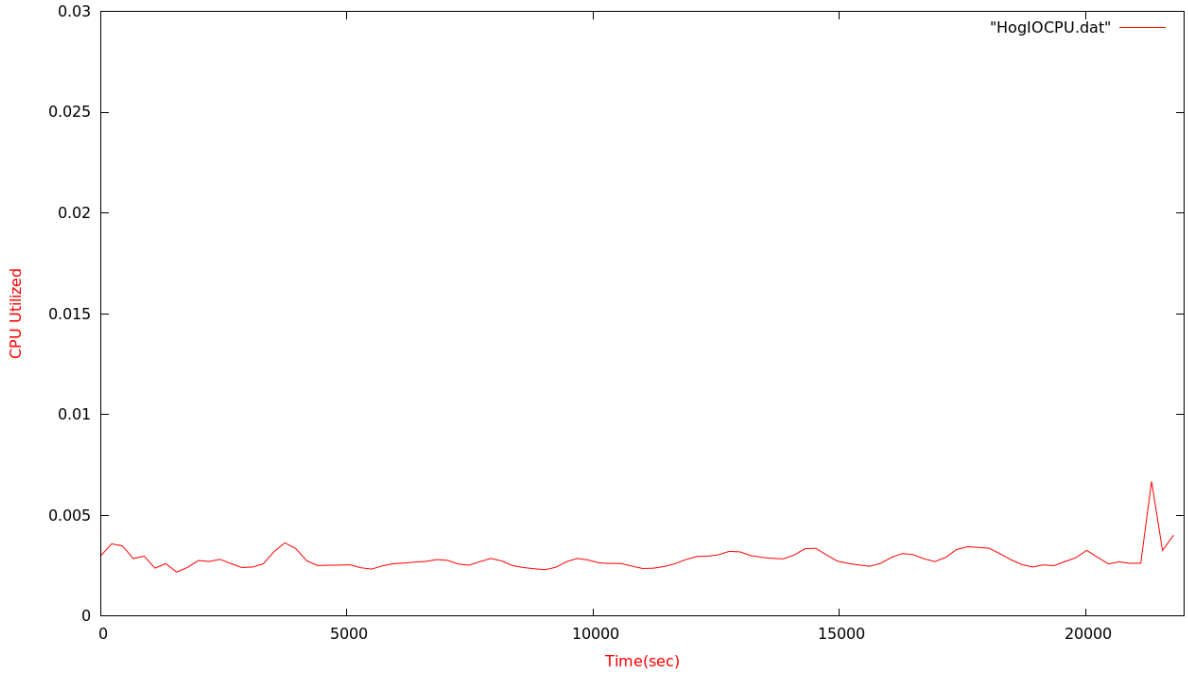
So to improve the performance and attain optimality one needs to ensure that the number of processes forked equals the number of CPUs. In this way each Partition which is a different process runs independently of other such forked processes on a separate CPU. (Note that context switches and CPU migrations will still occur as the OS has to schedule other processes on these CPUs as well but they wont be huge in number) In this implementation we will get the advantage of using shared memory as there will be optimal utilization of memory and the advantages of parallelism to go with it. There should be a count of the number of currently running processes being used for sorting which should be equal to the number of cores and a new process should be forked only when there is an empty slot.

REPORT – QUESTION 4a

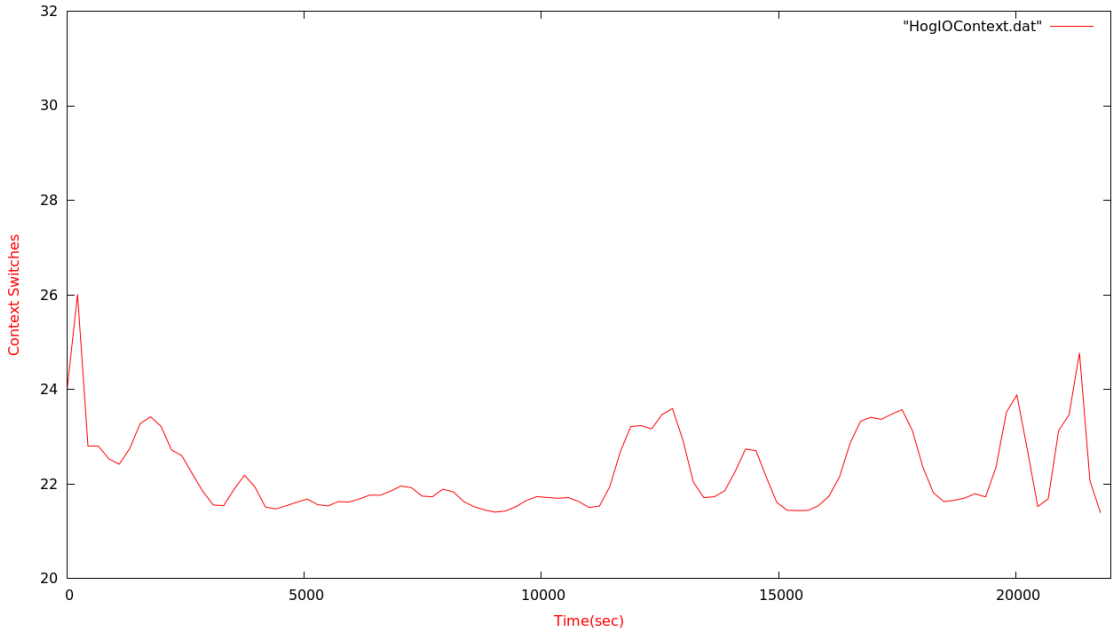
The objective of this question was to run dtella and linuxdcpp for around 3 hrs and to run perf on linuxdcpp every 10/20 seconds to collect data about its CPU usage and the number of context-switches.

Three runs of the test were performed:

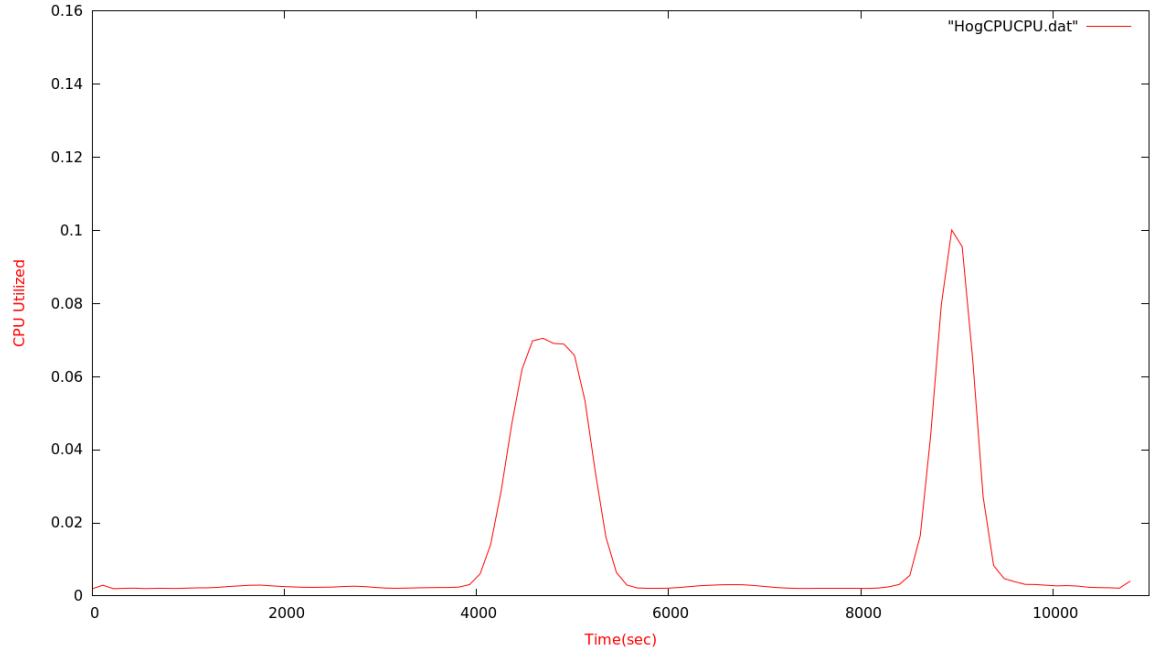
1) In this run, linuxdcpp was run along with a lot of IO intensive processes. To simulate a lot of IO intensive processes, the stress command was used with the -i option.



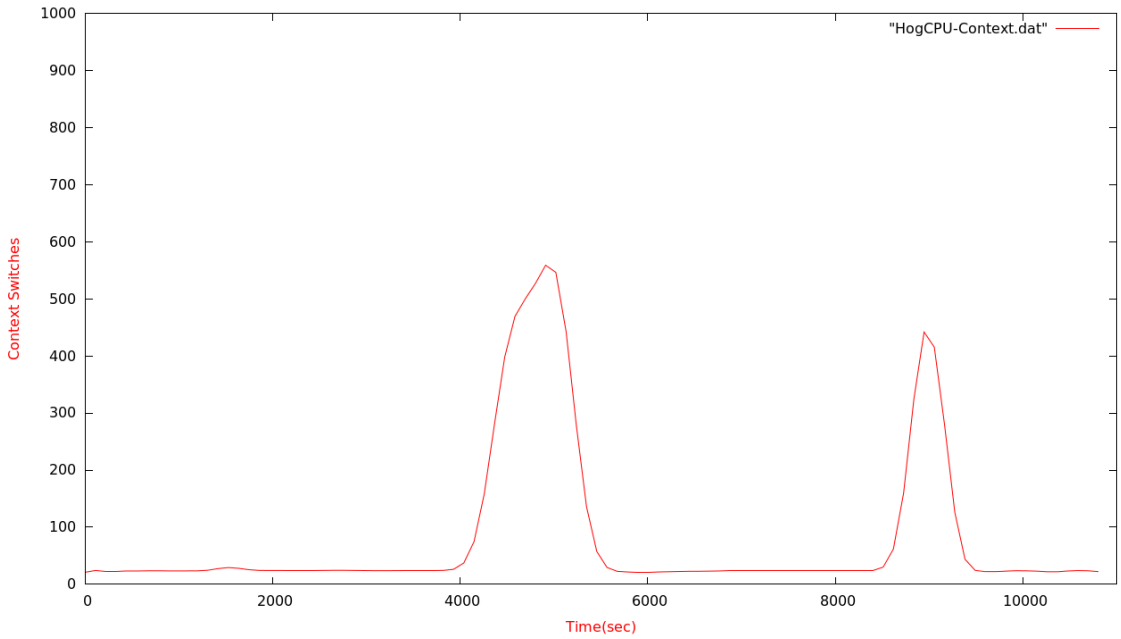
In this run, nothing was put to download on linuxdcpp nor did some other client download something.
As a result we get a consistent number of context-switches/sec and a consistent CPU utilization with no huge spikes or dips when we analyse linuxdcpp.



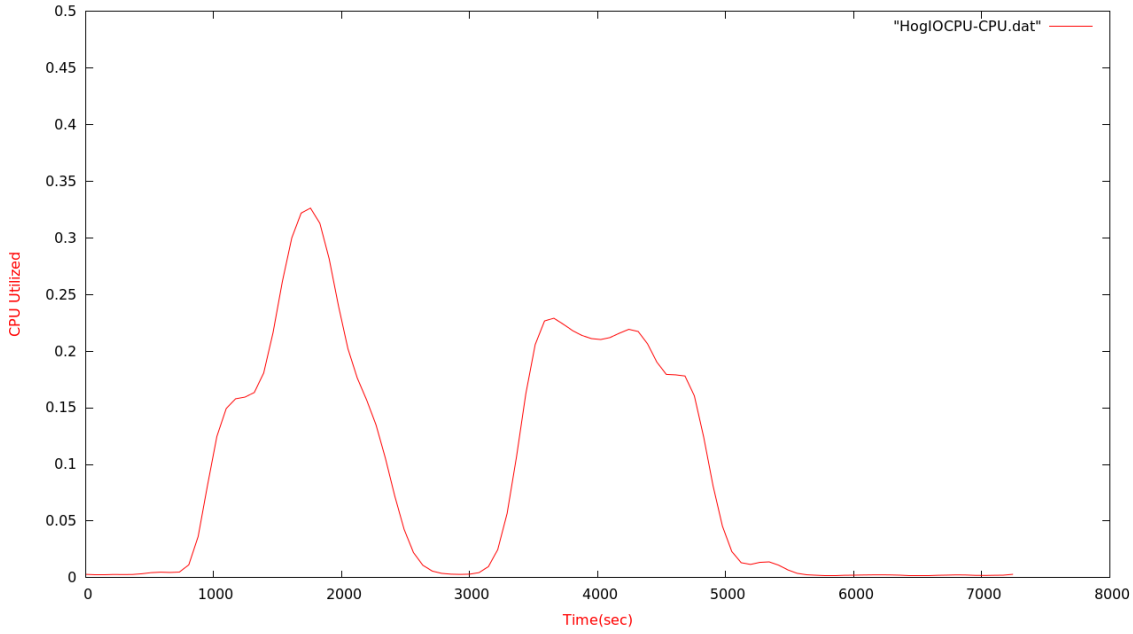
2) In this run, linuxdcpp was run along with a lot of CPU intensive processes. To simulate a lot of CPU intensive processes, the stress command was used with the -c option.



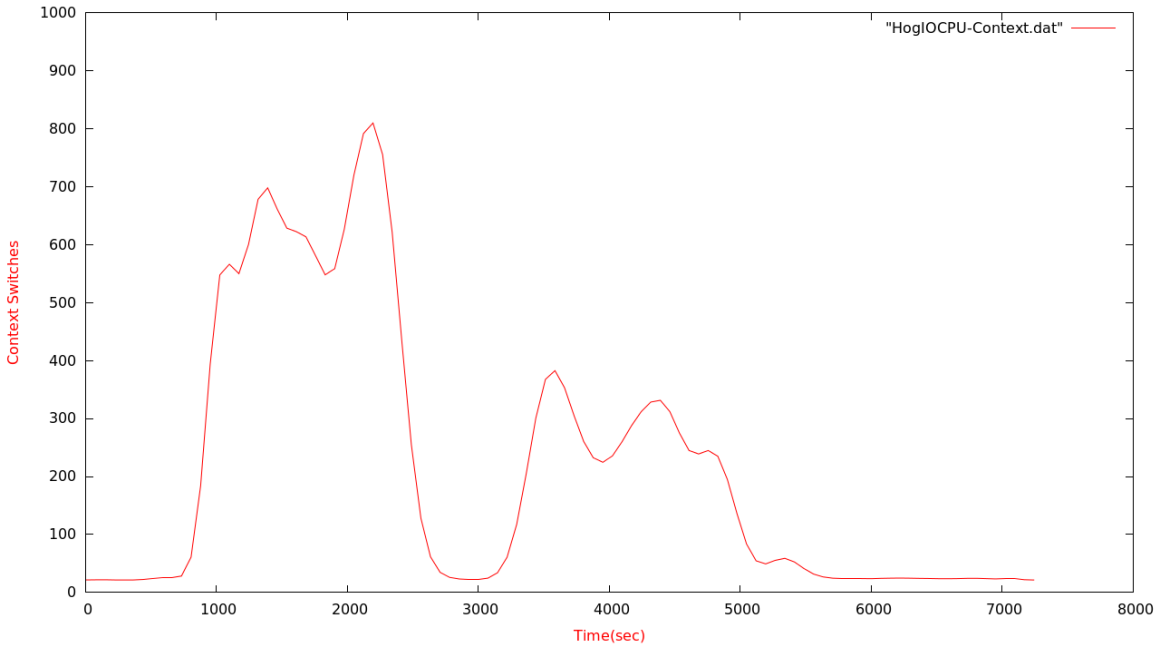
In this run, a couple of files were queued for download on linuxdcpp and after some time, some other client downloaded a couple of files from this system.
As a result there was a huge spike in the CPU utilization and the number of context-switches/sec in those time intervals.



3) In this run, linuxdcpp was run along with a lot of CPU and IO intensive processes. To simulate running a lot of IO intensive and CPU intensive processes simultaneously, the stress command with the -c and the -i option was used.



In this run, a batch of files was kept for download for an extended period of time and this was repeated again after an interval of time.
This resulted in high CPU utilization and a large number of context-switches/sec for a good amount of time.



It was also noticed that the mean CPU utilization of linuxdcpp was slightly higher when other IO intensive processes were running as opposed to when other CPU intensive processes were running. This is because when other CPU intensive processes run, then they hog the CPU and linuxdcpp gets less of the CPU.

Also the mean number of Context Switches of linuxdcpp was slightly higher when other IO intensive processes were running as against when other CPU intensive processes were running.

We can conclude that when a file is downloading or uploading on linuxdcpp, it utilizes the CPU and as a result is context-switched out from and in to very frequently as it utilizes system resources and is also scheduled more often.

REPORT – QUESTION 4b

This question entailed us to compare the performance of the command “sort” with the merge sort script for files that we create.
The merge sort script is written for Bash and splits files according to MAX_LINES_PER_CHUNK value which can be set so that the split file size is 100MB/200MB/300MB, etc as desired.
The script takes two arguments. The first argument is the large 1GB file to be sorted. The second argument is the file to which the output is to be stored.
The script then performs merge sort on the input file as required by the question.

The size of the input file used was 1.1 GB and it contained random 3 digit numbers.

Also please note that one cannot come to a conclusion on the results based on one run though appropriate care was taken to clean caches after each run. But certain inferences can still be drawn.

The “sort” command's performance and the merge sort script's performance for size of split file as specified is as follows:

Size of split files	**“sort” command**	**100 MB**	**200 MB**	**300 MB**	**400 MB**	**500 MB**	**600 MB**
Time taken (sec)	443.24	483.26	466.34	438.02	432.15	459.47	515.20
Task clock (msec)	1,145,542.09	1,035,820.30	995,855.53	993,081.86	1,027,099.29	1045134.27	1,097,596.35
CPU's utilized	2.584	2.143	2.135	2.267	2.377	2.275	2.130
CPU clock (msec)	1,145,580.89	1,035,845.39	995,878.43	993,106.56	1,027,129.95	1045176.77	1,097,625.69
Context switches	67,110	204,117	202,510	218,544	182,700	229,695	213,172
CPU migrations	801	7,405	5,536	5,731	5,621	6,501	8,793
Page faults	28,953	884,311	459,072	228,281	240,039	242,086	124,801

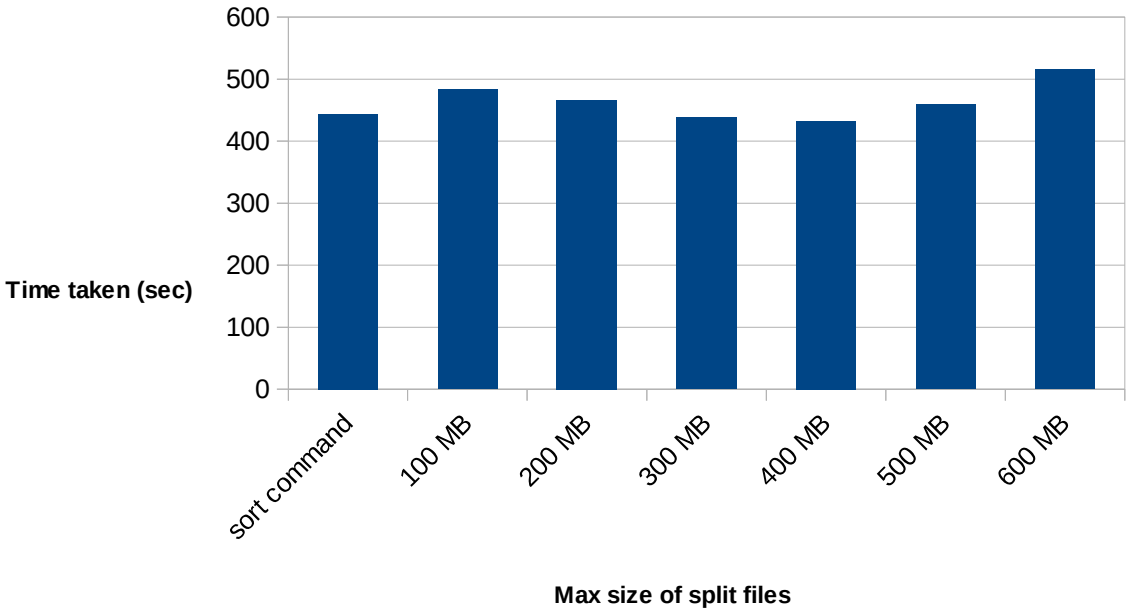
Note that 100 MB split size corresponds to approximately 11-12 files to be sorted individually, 200 MB split size corresponds to approximately 5-6 files, 300 MB split size corresponds to approximately 4-5 files, 400 MB split size corresponds to approximately 3-4 files, 500 MB split size corresponds to approximately 2-3 files and 600 MB split size corresponds to 2 files.

These split files are sorted individually and then merged.

The “sort” command uses an External R-Way merge sorting algorithm. In essence it divides the input up into smaller portions (that fit into memory) and then merges each portion together at the end. It stores working data in temporary disk files (usually in /tmp).

Two things can greatly affect the performance of this command. They are
1) The location of the temporary files which can be specified with the -T option and should be stored on the fasted available device.
2) Amount of memory to use which can be specified through the option -S. A large value should be given but care should be taken that over-subscription doesn't occur which would result in swapping to disk.

The script that we have written splits the file into two parts recursively, using the split command, until the size of the partition is as desired. Then each of the smaller files is sorted using the sort command and finally the smaller sorted files are merged together.
The splitting is done based on maximum number of lines each split file can contain because splitting by size is dangerous as it may lead to integers being split in the middle.
The number of lines corresponding to the file size is calculated beforehand by trial and error.



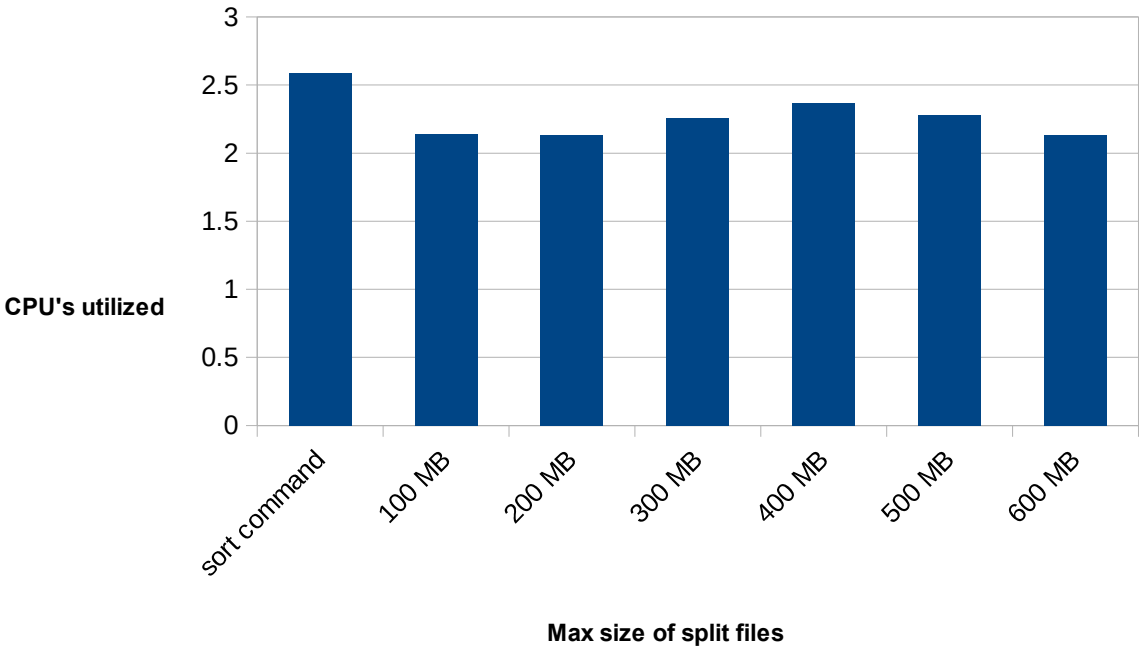
The best result for algorithm is achieved when the size of the split is around 400 MB which would result in approximately 4 evenly sized files to be sorted individually. Since we are doing the sorting of the individual files using sort command one after the other, the gain is not in doing the above in parallel. The gain may be at two places:
1) The sort command sorts the individual file chunks in parallel. So a size of 400 MB to be using the sort command on, might be optimal in terms of memory usage and size thus requiring less of the IO expensive disk storage.
2) The sort -m for merging the 4 400 MB files might be getting done in parallel resulting in a gain there too.

The most optimal result would be when we get the above gains of optimal memory usage and in parallel merge along with sorting the individual 400 MB chunks in parallel.
This means we are sorting the 400 MB files in parallel using the sort command for each of the files, which in turn sorts in parallel and utilizes CPU optimally.

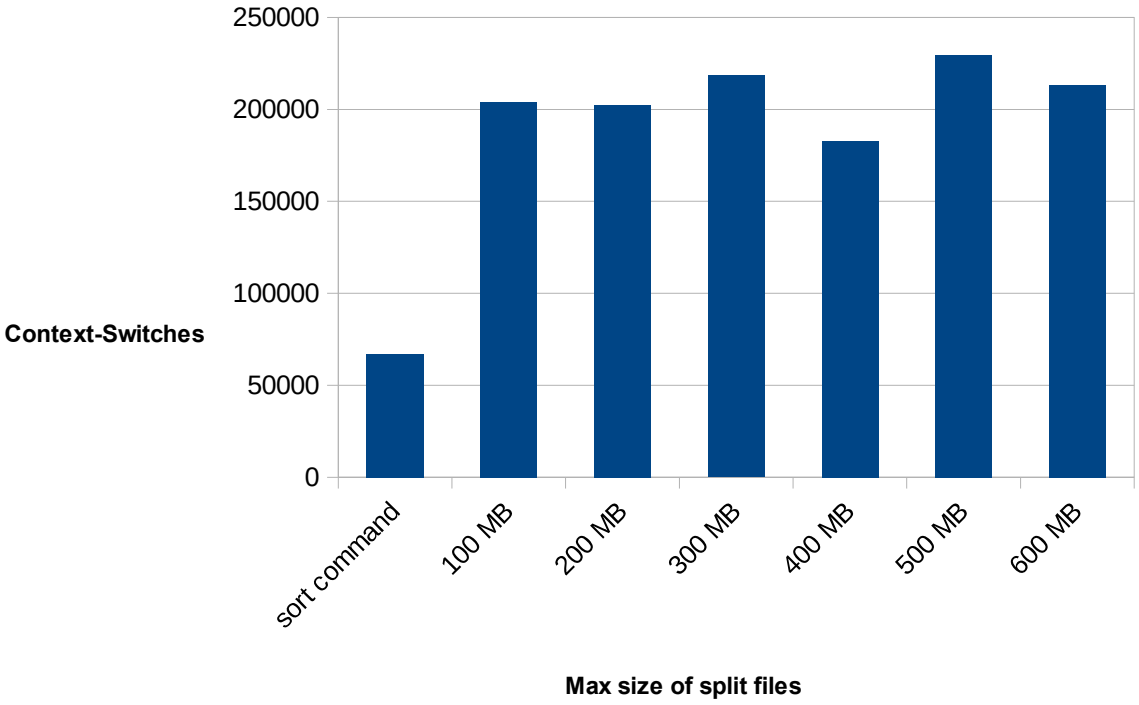
The rest of the split sizes correspond to either less than or more than 4 files to be sorted individually, which would mean that there are more number of smaller files to be sorted or there are less number of large files to be sorted than the number of CPUs and this leads to the overall sort taking more time due to less parallelism and more disk accesses.

An important factor which comes into play is the number of disk accesses that take place.
Since the temporary split files are stored on the disk, we should strive to minimize the number of accesses to those files. This also implies that the files which we sort individually should be able to fit into the memory as a whole, otherwise the sort command would split that file and store it on the disk.

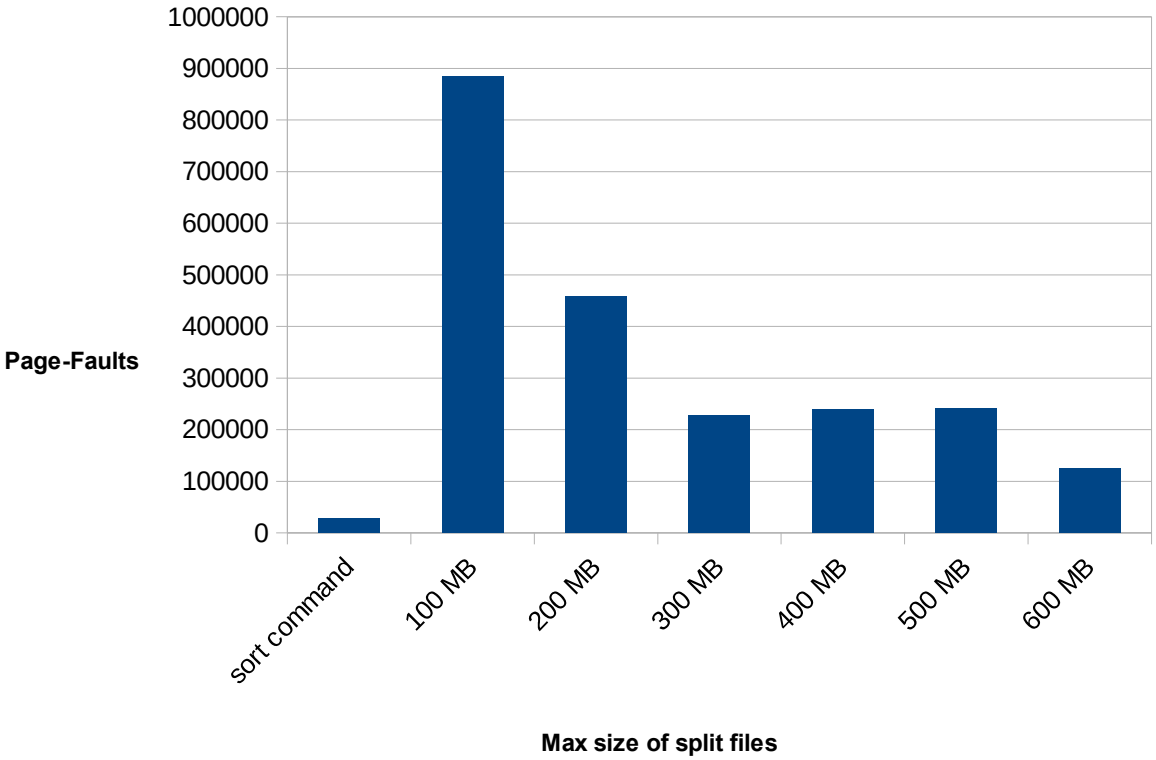
There should be optimal utilization of the CPU and the memory while minimizing the disk accesses.



The CPU Utilization of the merge sort version and the sort command is more or less the same. But the sort command utilizes the CPU more effectively. The 400 MB version utilizes a little less CPU but still does a bit faster.



The number of context switches is as shown in the graph.
We will have more context switches on both sides of the optimal 400 MB split because:
1) If the split size is less than 400 MB, then we are splitting into smaller files which will be stored on the disk. To sort, we need to read from the disk, which is an IO operation and as a result the process doing such IO operations gets context switched out so that the CPU can be utilized in the meantime.
2) If the split size is greater than 400 MB, then though larger split files are being stored on the disk, we will have the sort command splitting the files while it is individually sorting the original split files. Also the we need to read from the disk for the merge which is again an IO operation.



The number of page faults is maximum when the size of the split is minimum as the small files have to be repeatedly fetched from the disk. Only the file which is currently being sorted individually is stored in the memory and then another has to be fetched from the disk on a context switch.
The number of page faults is minimum when the size of the split is maximum as the whole of the large split file resides on the memory and disk accesses are therefore minimized.