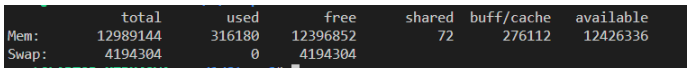
Questions to Answer

1.

I ran the assignment on my Windows Laptop using WSL2 - Ubuntu 16.0.4 LTS 64-bit. My computer has a i7-9750H processor, which has 12 cores @ 2.6 GHz, with virtualization turned on. It also has 16GB ram.

Output from "uname -a": Linux LAPTOP-M7BM40VA 4.19.104-microsoft-standard #1 SMP 2020 x86_64 x86_64 x86_64 GNU/Linux

Output from "free":



	total	used	free	shared	buff/cache	available
Mem:	12989144	316180	12396852	72	276112	12426336
Swap:	4194304	0	4194304			

2.

The number of clock ticks is dependent on the shortest time period recognised. The real time can vary greatly for the same number of clock ticks because of CPU utilisation.

3.

The first two debugging lines are expected, as the main process goes through main().

In the quick sort function, a fork() call is made.

The faster process sorts the right side of the input, and returns from the quick sort function first and prints the clock ticks. The array is not sorted because the other thread has not finished sorting yet. This is why "not sorted" is displayed.

The second process sorts the left side of the input, and returns from the quick sort function and prints the clock ticks again. The array is not sorted because there has been no sharing of information between the two processes. Therefore "not sorted" is displayed.

So there are two problems: The processes are not waiting for each other to finish, and there is no information passing between them.

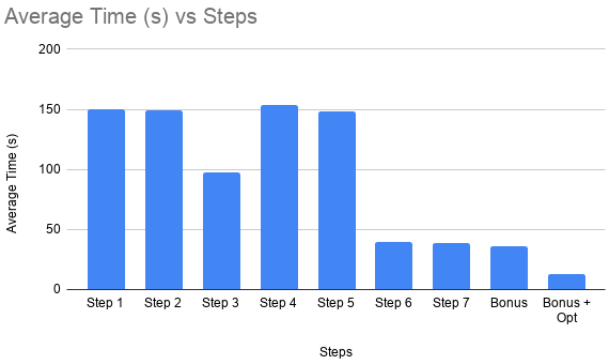
Assignment One Report

Findings

Each step of the assignment was completed successfully. To achieve an accurate measurement of the performance of each step, an average of five measurements was taken. Each measurement was performed on an input 10,000,000 to ensure consistency. The 'time' function was used to measure the time taken, and htop was used to analyse the thread/process behaviour. Fig 1a. shows the raw data and Fig 1b. shows a graph showing the relative speeds of the implementations. Already a clear pattern emerges. Steps 1, 2, 4, and 5 are around the same speed at 150 seconds. Steps 6, 7, and bonus are also similar in speed at around 40 seconds. The fastest implementation was the bonus + optimisation. All of these observations will be explored in the corresponding paragraphs, and an explanation as to why these implementations behave in relation to the others is given.

# Step	Measurements (s)					Avg Time (s)
Step 1	148	157	152	147	146	150
Step 2	147	153	146	151	149	149.2
Step 3	84	94	101	112	96	97.4
Step 4	145	150	165	158	151	153.8
Step 5	150	146	146	147	153	148.4
Step 6	39	39	39	40	39	39.2
Step 7	38	40	38	38	38	38.4
Bonus	36	38	38	35	34	36.2
Bonus + Opt	12	12	12	12	12	12

(a) Raw measurements



(b) Graph of average times

Figure 1: Findings

Step One

The basic implementation given runs on a single process and has an average time of 150 seconds. This will be used as a baseline to compare other versions.

Step Two

Building on top of the given program, step two introduces an extra thread to sort half of the array. As seen in Fig 2., each thread utilises the same process and core, and only one is running at a time, as the process switches between the two threads. Therefore, the performance is almost identical to step one. I expected that the performance would be slightly lower due to thread overhead. After some research I have found that thread creation and overhead is relatively small in C.

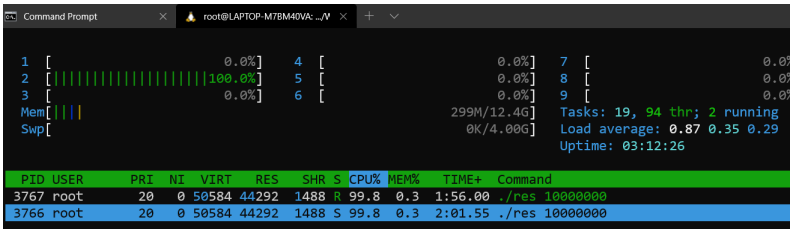


Figure 2: Step two htop

Step Three

This step is the same as the previous, except threads are created recursively until. After measuring three times an average of 129,000 threads were made for each execution of the program. The code to record these statistics was thread-safe through the use of a mutex lock, however, as the incrementation of a counter is a relatively small operation relative to the rest of the code (mostly split_on_pivot), I found that there was almost no difference after introducing this measure. Fig 3. shows that all cores are being fully utilised, which explains the speed differential compared to step two.

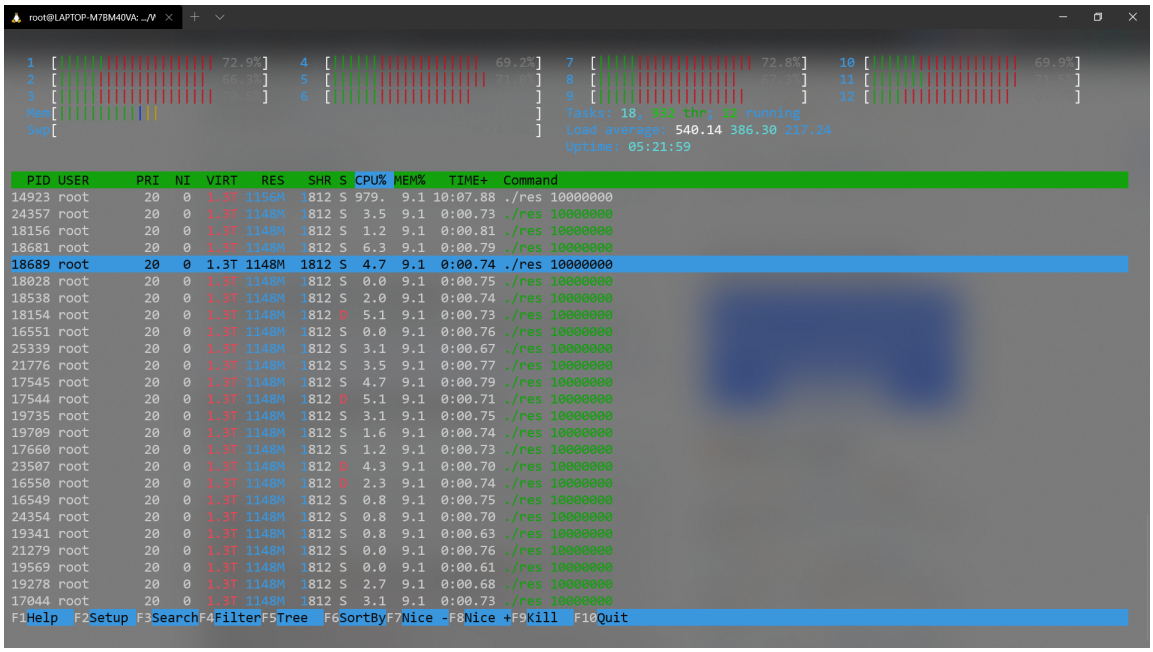


Figure 3: Step three htop

Step Four

I found this implementation to be the hardest to program, and the speeds do not justify the effort. This step has the same runtime as step two, for the same reasons. Observing htop, I found that the threads only utilised one core, so the speed (154 seconds) is essentially the same as step one, but with overhead for the threads (very little).

Step Five

This implementation is the first which utilises an additional process, although the performance is not very promising. From htop (Fig 4.), I observed that the second process is only utilised for five seconds before running out of work to do. This is also seen by the initial utilisation of two cores, before dropping down to one for most of the execution time. This explains why the runtime of 148 seconds is so similar to step one. Based on this analysis, we can see that there is potential for greater performance if processes are utilised to a greater extent.

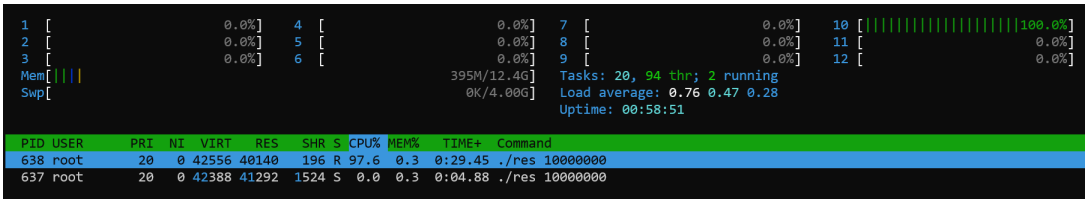


Figure 4: Step three htop

Step Six

Initially, I tested minimum size values of 5 and 10. After finding that there was negligible difference between these values, I increased my step size to a factor of ten. Three measurements were taken for the following sizes and then averaged (Fig 5): 5, 10, 100, 1000, 10,000, 100,000, 1,000,000, 5,000,000. The graph is flat from 5-100,000, and then there is a steep increase until 5,000,000. The speed of 5,000,000 is expected as this is essentially the same situation as in step five. Therefore, I chose my optimal size as 100,000. With this size value, I observed 95 processes created for the program on sytem monitor.

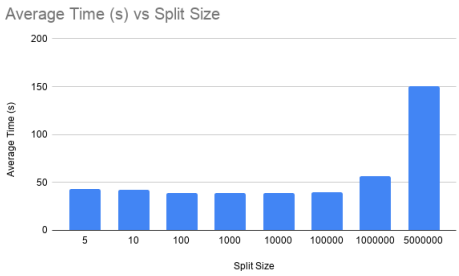


Figure 5: Step six split size testing

The speed when optimised is significantly faster than all previous steps at 39.2 seconds. The reason why increasing the amount of processes does not result in a greater amount of performance is the limit of CPU cores on my machine. Only twelve processes can be running at any given time, so decreasing the split size should not have a great impact on speed. In comparison to step five, all cores were being utilised for the majority of the runtime (Fig 6.), which explains the faster runtime.

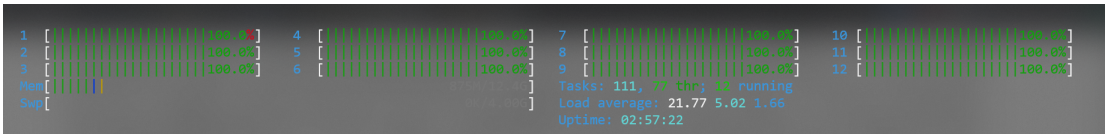


Figure 6: Step six split size testing

Step Seven

This implementation was the easiest to programme, and required the least amount of lines of modification. As shown in Fig 7. the split sizes chosen were the same as in step six, for consistency. The behaviour of split sizes is also similar, as the only difference between step six and this step is the memory passing. The same optimal size of 100,000 was chosen. The performance of this implementation is 38.4 seconds. This difference in comparison to step six could be due to the lower overhead of mmap vs pipes, although this could also be due to variance.

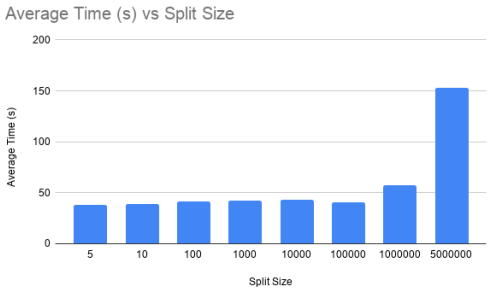


Figure 7: Step seven split size testing

Bonus Step

From my previous knowledge around TimSort - Python's preferred algorithm for sorting, I knew that using insertion sort below 16 values should be faster than quicksort. With some further research and testing, I decided to add this optimisation to the fastest previous step - step seven. The average time of this implementation is 36.2 seconds. All measurements of this step were the same or faster than all measurements for step seven. Therefore, I can be quite certain that this implementation is faster than all others.

A further optimisation that can be used to speed up the implementation is the compilation flags "-O3" and "-Ofast". These tell the compiler to optimise the code further. In my compilation, I used "-Ofast" as this provided the greatest performance gain. The performance from this step alone is extremely significant, with a consistent average speed of 12 seconds.

Ordering Techniques from Slowest to Fastest

Step number	Technique	Average time (s)
Four	Two threads - reuse extra thread	153.8
One	No parallelisation	150.0
Two	Two threads - no reuse	149.2
Five	Two processes	148.4
Three	Maximum threads	97.4
Six	Multiple processes - piping	39.2
Seven	Multiple processes - mmap	38.4
Bonus	Multiple processes (mmap) + insertion sort	36.2
Bonus + Opt	Multiple processes (mmap) + insertion sort + compilation optimisation	12.0

Explanations are in their corresponding step paragraphs.