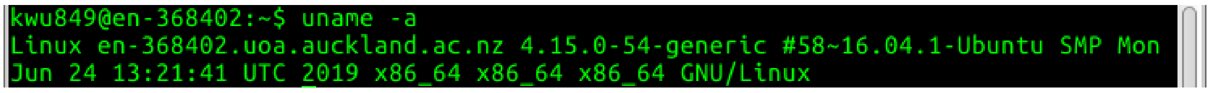
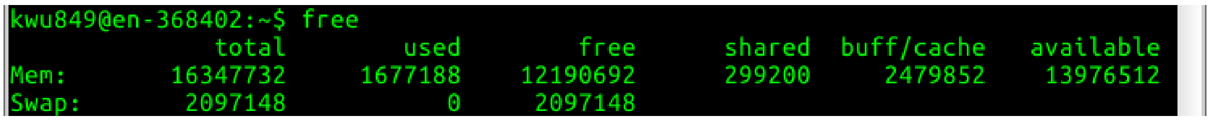
SOFTENG 370 Assignment 1 Report

Questions

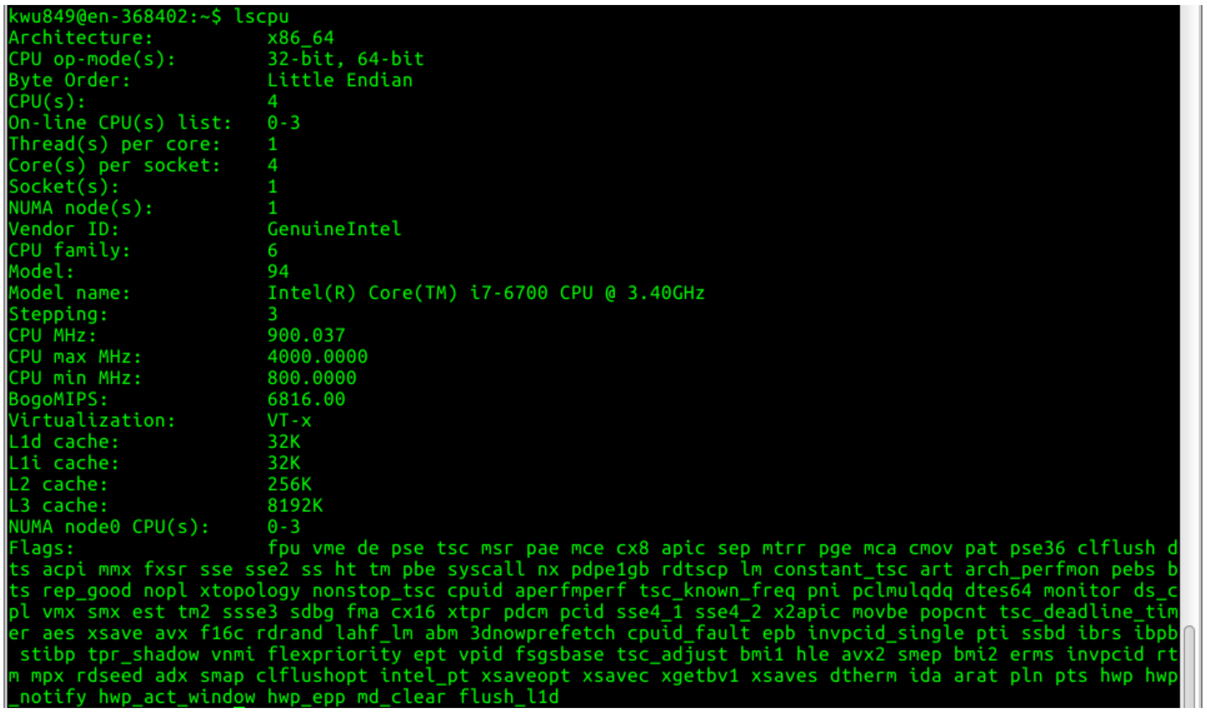
1. I ran the assignment on Linux on the lab computers in UG4. I did not use a virtual machine.



```
kwu849@en-368402:~$ uname -a
Linux en-368402.uoa.auckland.ac.nz 4.15.0-54-generic #58~16.04.1-Ubuntu SMP Mon
Jun 24 13:21:41 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```



```
kwu849@en-368402:~$ free
              total        used        free      shared  buff/cache   available
Mem:       16347732     1677188    12190692      299200     2479852    13976512
Swap:       2097148           0     2097148
```



```
kwu849@en-368402:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               900.037
CPU max MHz:           4000.0000
CPU min MHz:           800.0000
BogoMIPS:              6816.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush d
ts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs b
ts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_c
pl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_tim
er aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb
 stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rt
m mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp
_notify hwp_act_window hwp_epp md_clear flush_l1d
```

2. The approximate maximum size before segmentation error is 10000000.

3. The limit on the stack size smaller than the amount of memory actually available to the stack because the stack shares the same memory space with the heap. The stack increases downwards, and the heap increases upwards, therefore the stack cannot use the memory allocated to the heap.

4. It not a good idea to start a new thread every time merge_sort is called because a large amount of threads will be created as the size of the integer array increases. Creating more threads than the number of cores or virtual cores on a CPU will require scheduling to allow each thread to run for a share of the time. If there are too many threads, it puts a large burden on the operating system to schedule them fairly. Since each thread is allocated some memory, creating too many threads can also use up a lot of memory.

5. Spin locks constantly checks to see if a lock is active and will be less efficient the longer it has to check, however if the locks are active for a very short amount of time then it can be faster than mutex locks which uses expensive operations to sleep and wake up threads. The spin lock in step 5 is only used to check and increment the core count, and therefore the spin lock is faster than a mutex lock here.

This assignment taught me a lot about the different ways of parallelising algorithms in C. I learnt something new from doing each step of the assignment and trying to understand why I was getting the results. Step 3 did not manage to finish executing in a reasonable amount of time. Out of the other programs, step 1 was my slowest, followed by step 4, and 5, then step 7 and 2, then 9 and 6 and finally step 8 was my fastest.

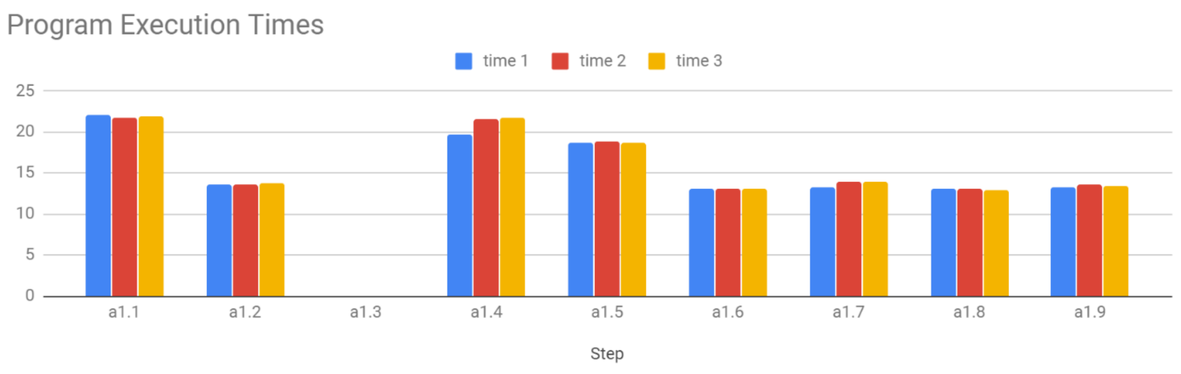
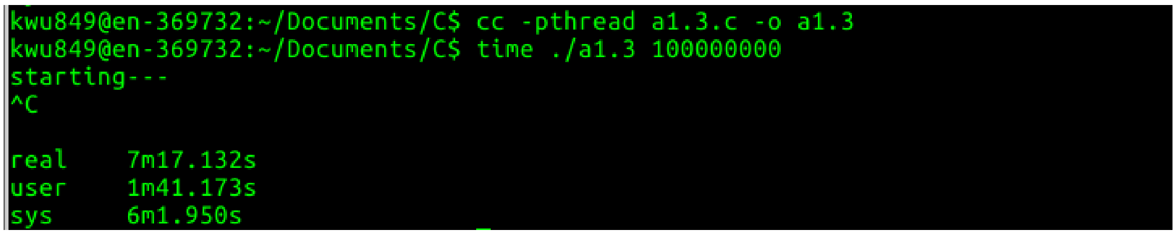


Figure 1: Program Execution Time Chart

It was not a surprise that step 3 did not manage to handle a 100,000,000-integer array since it was starting a new thread every time merge_sort was called. Even for smaller arrays, where it did finish sorting, it was significantly slower than the other programs. Since merge_sort is called recursively, a large array will need to call merge_sort a lot and every time a new thread will be created. This will very quickly overwhelm the scheduling capabilities of the CPU and result in problems that prevent the program from finishing.

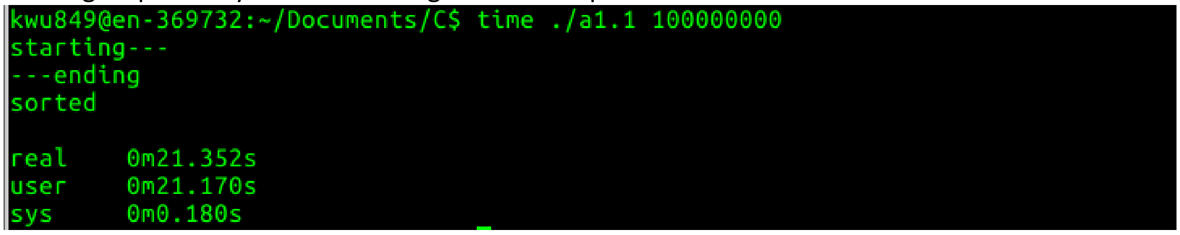


```
kwu849@en-369732:~/Documents/C$ cc -pthread a1.3.c -o a1.3
kwu849@en-369732:~/Documents/C$ time ./a1.3 100000000
starting---
^C

real	7m17.132s
user	1m41.173s
sys	6m1.950s
```

Figure 2: Step 3 Time

Out of the programs that did finish executing, as expected step 1 was the slowest at around 21.89 seconds. Step 1 was a simple modification on the original code to allow it to run for larger arrays. I used setrlimit to allocate a higher resource limit so that the program can store the 100,000,000-integer array and the two halves of the array that needed to be combined. The entire program is running sequentially without making use of multiple cores and therefore should be the slowest.

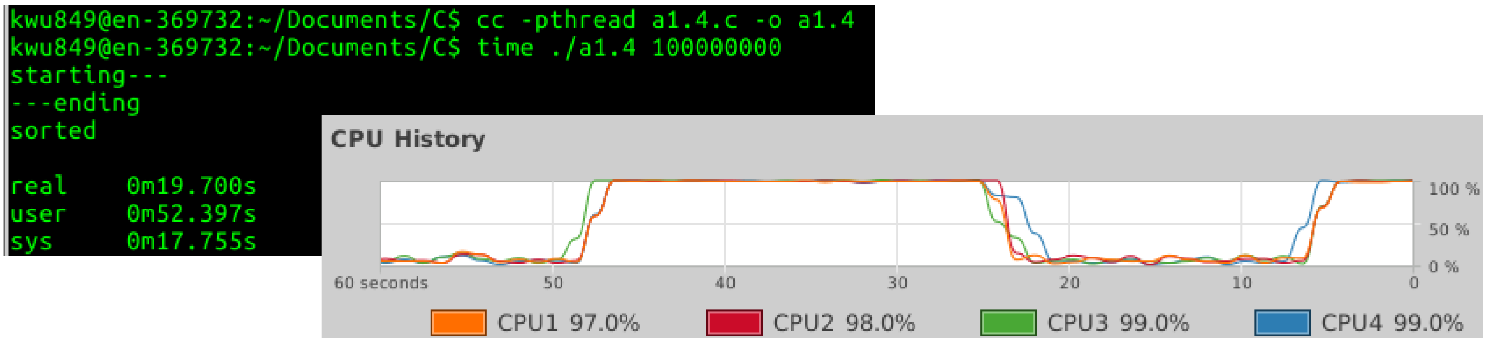


```
kwu849@en-369732:~/Documents/C$ time ./a1.1 100000000
starting---
---ending
sorted

real	0m21.352s
user	0m21.170s
sys	0m0.180s
```

Figure 3: Step 1 Time

I was very surprised when I saw that step 4 was not only slower than step 2 but also almost as slow as step 1 at an average of 20.98 seconds. I ran it multiple times and checked system monitor to make sure that all 4 cores were indeed being used. In theory splitting the work across 4 threads should make the program run almost 4 times as fast as step 1, and twice as fast as using 2 threads in step 2, however the time I recorded was marginally better than step 1 and significantly slower than step 2. I think one of the reasons for this is because of the mutex locks on the core count, every time merge_sort is called, it needs to check the number of threads already created to ensure it does not create more threads than cores. Since this variable may be being modified by another thread, there needs to be a lock every time it is checked and incremented. I tried removing all the locks and instead use Atomic variable for the count which resulted in a significant improvement to around 15 seconds. Although this was a huge improvement, it was still slower than step 2, which may be caused by uneven distribution of work among the four cores since my program creates new threads of the left block first before moving on to the right block.

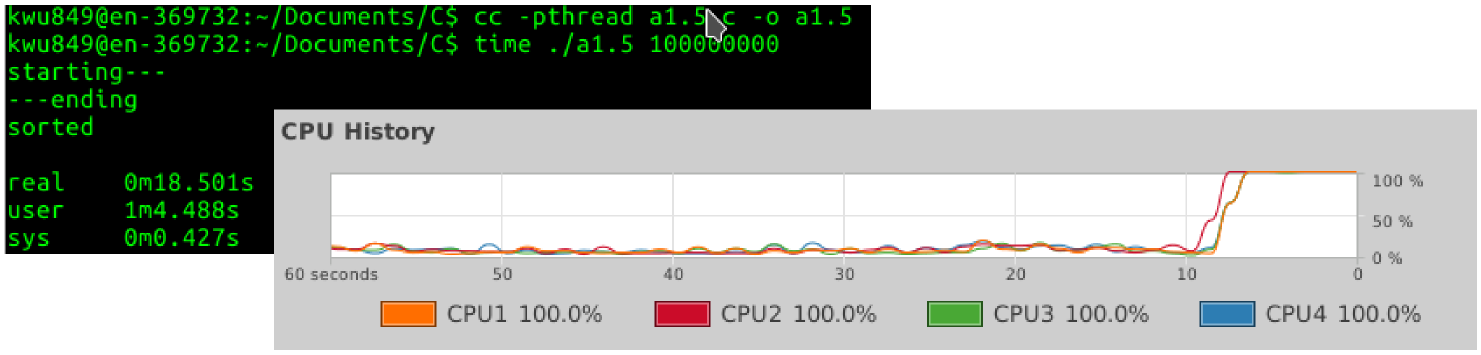


```
kwu849@en-369732:~/Documents/C$ cc -pthread a1.4.c -o a1.4
kwu849@en-369732:~/Documents/C$ time ./a1.4 100000000
starting---
---ending
sorted

real	0m19.700s
user	0m52.397s
sys	0m17.755s
```

Figure 4: Step 4 System Monitor and time

My next slowest average time was Step 5 at 18.74 seconds. I again checked the system monitor to make sure all four cores were being used. After trying to understand what was happening with Step 4, I expected Step 5 to produce similar results as it mostly shared the same code with the only difference being the spin locks instead of mutex locks on the counter. Since spin locks are generally regarded as less efficient than mutex locks, I did not expect it to outperform step 4, and was pleasantly surprised when it produced modestly better times. Then I realised that this was one of the few instances where spin locks are more efficient, when the locks are only held for very short amounts of time.

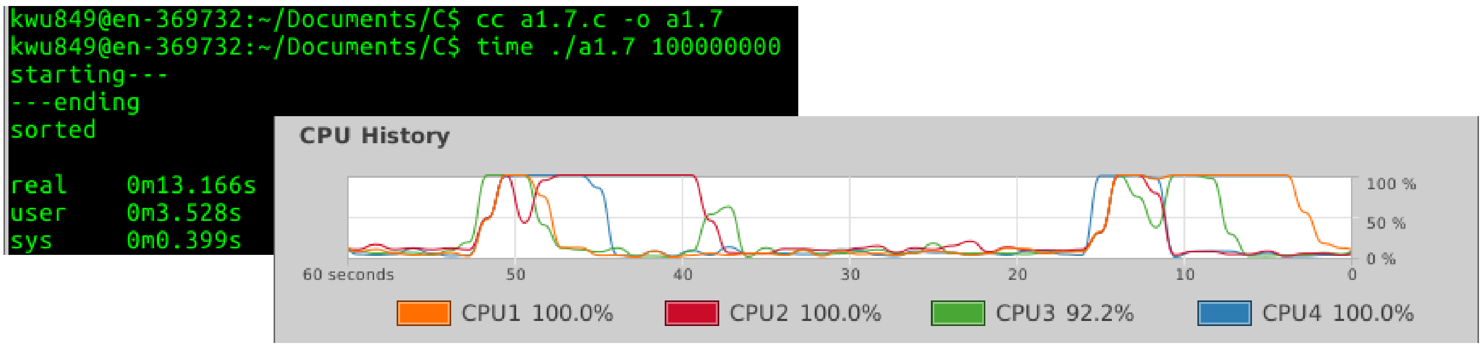


```
kwu849@en-369732:~/Documents/C$ cc -pthread a1.5.c -o a1.5
kwu849@en-369732:~/Documents/C$ time ./a1.5 100000000
starting---
---ending
sorted

real	0m18.501s
user	1m4.488s
sys	0m0.427s
```

Figure 5: Step 5 Time and System Monitor

My implementation of step 7 took an average of 13.69 seconds to sort the integer array. Step 7 uses as many processes as the number of cores as there are on the system and pipes to communicate with each other. Using processes instead of threads had a definite improvement on the execution time, which was expected since the processes did not have to wait for locks. Although this is an improvement over using 4 threads, it is still slower than using 2 processes and using 2 threads. This is again likely due to uneven distribution of work across the 4 CPU, as shown in figure 3, all four CPUs are being used initially, however some processes finish earlier than others resulting in that CPU no longer being used and waiting until the other processes have finished executing. Optimising CPU usage by creating new processes after one has finished should lead to even greater improvements in execution time.

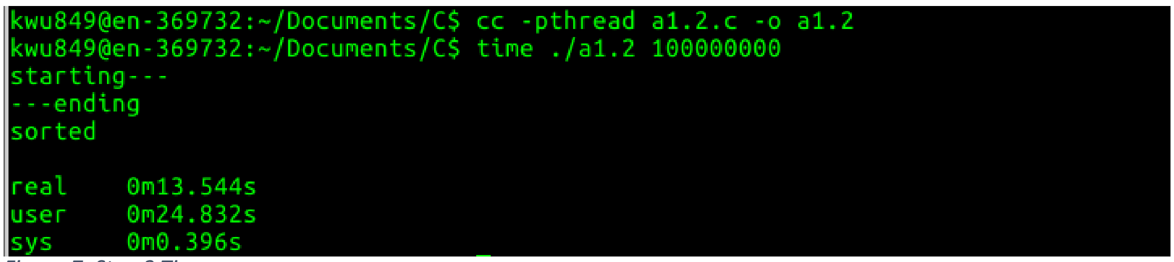


```
kwu849@en-369732:~/Documents/C$ cc a1.7.c -o a1.7
kwu849@en-369732:~/Documents/C$ time ./a1.7 100000000
starting---
---ending
sorted

real	0m13.166s
user	0m3.528s
sys	0m0.399s
```

Figure 6: Step7 System Monitor and Time

Step 2 simply runs the algorithm on 2 threads instead of 1. I was very surprised that step 2 turned out to be one of my faster versions of the algorithm, taking an average of 13.61 seconds. From step 2 onwards, I decided to use malloc instead to setrlimit to let the program allocate the necessary memory for the size of the array instead of doing it manually. My implementation involved manually splitting the integer array in half and then running merge_sort on each half in an individual thread. Since it doesn't dynamically generate threads, there is no need for locking and since the array is split manually, we can be sure the workload is distributed evenly across both threads. This means that my implementation of the dual threaded program has fewer areas where the execution is suboptimal compared to my multithreaded program.

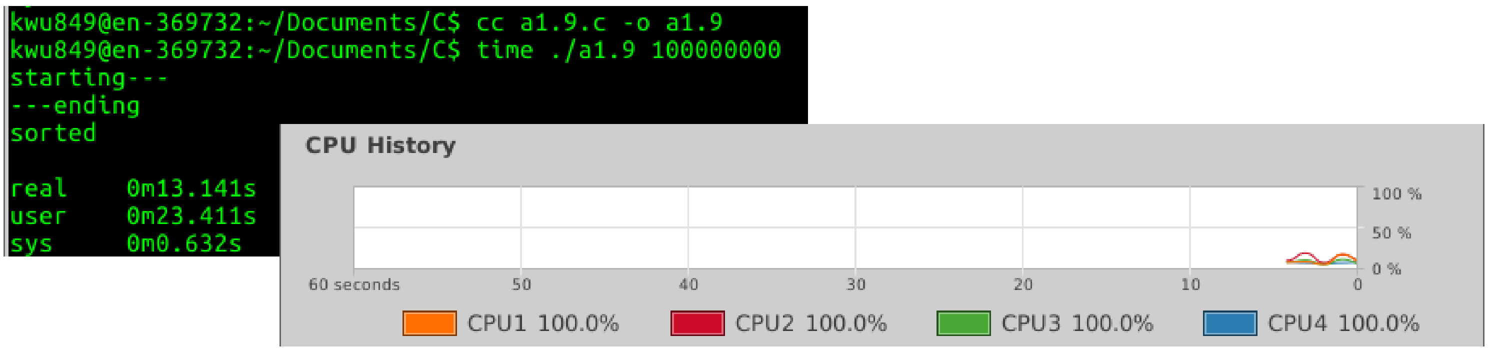


```
kwu849@en-369732:~/Documents/C$ cc -pthread a1.2.c -o a1.2
kwu849@en-369732:~/Documents/C$ time ./a1.2 100000000
starting---
---ending
sorted

real	0m13.544s
user	0m24.832s
sys	0m0.396s
```

Figure 7: Step 2 Time

Step 9 make use of multiple processes and shared memory instead of pipes in step 7 which resulted in an average time of 13.38 seconds. Using shared memory instead of pipes to transfer data provided a slight increase in performance since there is slightly less overhead in reading and writing from pipes and instead directly access the data as soon as the child process has finished. Step 9 initially uses all 4 CPUs but also suffers from the same problem as step 7 with some processes finishing before others resulting in suboptimal times.

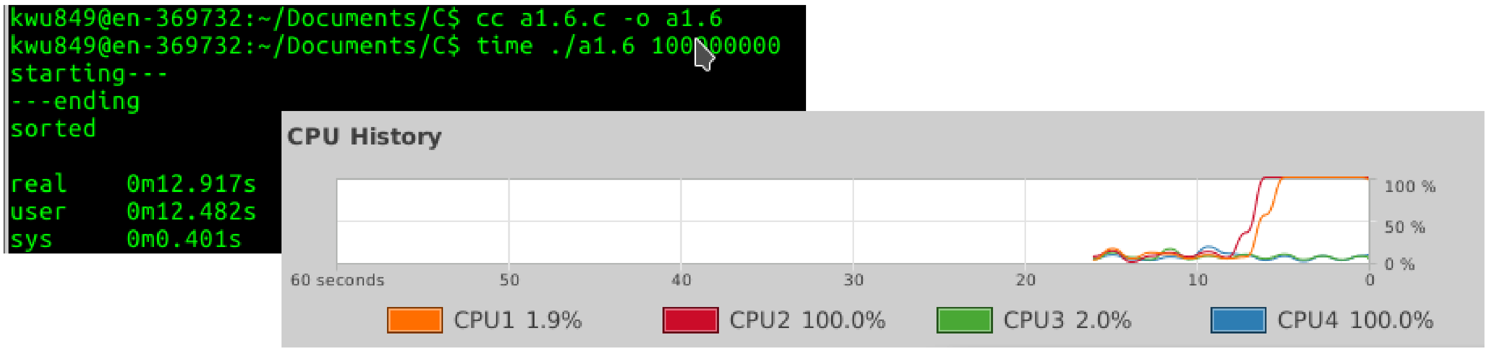


```
kwu849@en-369732:~/Documents/C$ cc a1.9.c -o a1.9
kwu849@en-369732:~/Documents/C$ time ./a1.9 100000000
starting---
---ending
sorted

real	0m13.141s
user	0m23.411s
sys	0m0.632s
```

Figure 8: Step 9 System Monitor and Time

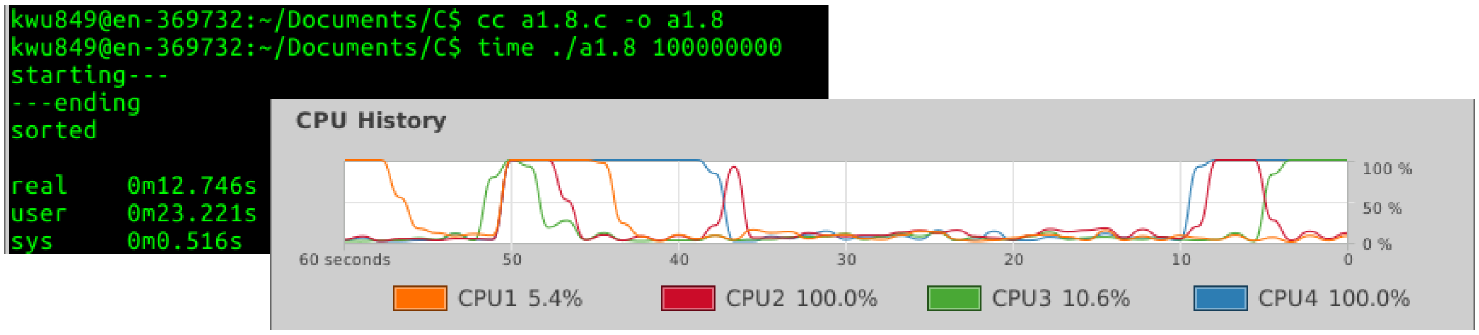
Finally, steps 6 and 8 were my fastest programs with average times of 13.12 and 13.00 seconds respectively. Both these programs ran using two processes with step 6 using pipes and step 8 using shared memory to communicate between them. It was expected that using shared memory would result in slightly faster times. The reason these programs were faster than their multithreaded counterparts is likely because they made use of both CPUs for the entire duration instead of some process finishing earlier.



```
kwu849@en-369732:~/Documents/C$ cc a1.6.c -o a1.6
kwu849@en-369732:~/Documents/C$ time ./a1.6 100000000
starting---
---ending
sorted

real	0m12.917s
user	0m12.482s
sys	0m0.401s
```

Figure 10: Step 6 System Monitor and Time



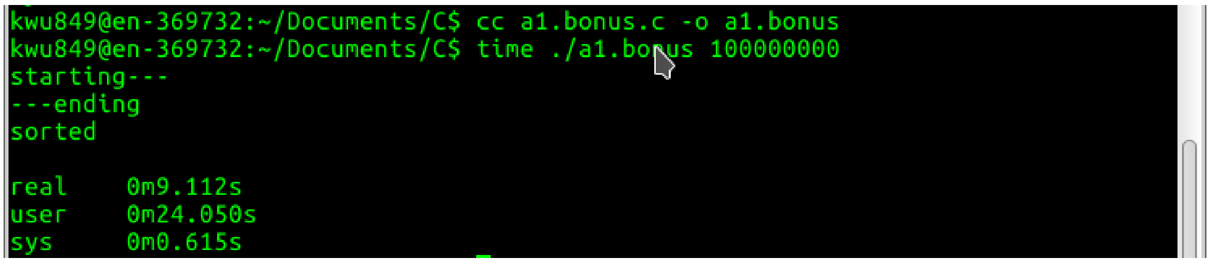
```
kwu849@en-369732:~/Documents/C$ cc a1.8.c -o a1.8
kwu849@en-369732:~/Documents/C$ time ./a1.8 100000000
starting---
---ending
sorted

real	0m12.746s
user	0m23.221s
sys	0m0.516s
```

Figure 9: Step 8 System Monitor and Time

In conclusion, I think the most important thing I learnt is that simply using more threads or processes does not automatically make a program run faster and in order to effectively parallelise a program it is just as important to optimise a variety of factors including task allocation, data transfer and concurrency.

My bonus step builds upon step 8 and takes advantage of the CPU scheduling to create a few more processes than cores to maximise CPU usage and was able to achieve an average of 9.23 seconds.



```
kwu849@en-369732:~/Documents/C$ cc a1.bonus.c -o a1.bonus
kwu849@en-369732:~/Documents/C$ time ./a1.bonus 100000000
starting---
---ending
sorted

real	0m9.112s
user	0m24.050s
sys	0m0.615s
```