William Shin 884 819 645
SOFTENG 370 A01

Questions

1. (Linux machines on UG4 computers)

Output of ‘uname -a’:

Linux en-368398.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

Kernel name	Linux
Kernel release	4.15.0-54-generic
Kernel version	#58~16.04.1-Ubuntu
Machine hardware name	x86_64
Processor type	x86_64
Hardware platform	x86_64
OS	GNU/Linux

Output of ‘free’:

```
wshi593@en-368398:~/Documents/Github/370a01$ free
              total        used        free      shared  buff/cache   available
Mem:       16347732     1265320    11998628      105072     3083784    14604088
Swap:       2097148           0     2097148
```

The machine has a total of 16,347,732 kibibytes (16GB) installed memory.

Output of ‘lscpu’:
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s): 1
Model name: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz

2.

```
wshi593@en-368398:~/Documents/Github/370a01$ ./a.out 1100000
starting---
Segmentation fault (core dumped)
wshi593@en-368398:~/Documents/Github/370a01$ ./a.out 1050000
starting---
Segmentation fault (core dumped)
wshi593@en-368398:~/Documents/Github/370a01$ ./a.out 1000000
starting---
---ending
sorted
wshi593@en-368398:~/Documents/Github/370a01$
```

The original programme can sort an array of 1,000,000 elements and no more than 1,050,000 elements.

3. The limits ensure that programming errors such as infinite recursive calls do not use all available memory and cause the OS to crash. Instead, when the soft limit is exceeded (the programme attempts to access memory addresses out of its bounds), the programme crashes with a ‘Segmentation Fault’ error. The soft limit (as shown in step 1) can be modified by the programmer when needed whereas the hard limit is set by the superuser/root. The hard limit is the soft limit’s boundary, and by setting the soft limit to be lower, while still ensuring that the programme has enough memory to run, the programmer can detect segmentation faults earlier.

4. The programme will end up creating too many threads. The vast number of threads consume and deplete available system resources (memory) due to the fact that they each have a stack. Also, the threads need to be scheduled by the OS which decreases performance. Furthermore, the programme becomes more complex with more threads, making it more difficult to understand and debug.

5. With mutexes, threads go to sleep when it fails to access a locked resource and are woken by the thread holding the lock whereas with spinlocks, threads constantly retry to access the resource until it succeeds. So, mutexes are considered heavyweight as it is computationally expensive to put threads to sleep and wake them up again. If the resource is only locked for a short period of time, the overhead with mutexes (i.e. putting a thread to sleep and waking it up again) can exceed the time it would have taken for a thread to constantly poll a spinlock.

In our case, when running the programme on a multi-core system with locks that are held for only short periods of time, it is acceptable to use spinlocks as we can achieve greater processing throughput by using spinlocks instead of mutexes. This is because mutexes decrease performance by wasting a considerable amount of time from repeatedly putting threads to sleep and waking them up again.

Introduction

The SOFTENG 370's first assignment involved parallelizing the MergeSort algorithm in many different ways, and comparing the performance of each implementation.

The steps taken to parallelize the algorithm include:

1. Base case- running MergeSort on one thread
2. Using two threads
3. Using a new thread for every recursive MergeSort function call
4. Using as many threads as there are cores in the machine and using pthread_mutexes to keep track of a shared state, the number of active threads
5. Using as many threads as there are cores in the machine and using pthread_spinlocks to keep track of the number of active threads
6. Using two processes with fork() and using a pipe to communicate between the processes
7. Using as many processes as there are cores in the machine with fork(), using a pipe to communicate between the processes and using pthread_mutexes to keep track of the number of active threads
8. Using two processes with fork() and sharing the memory to be sorted by both processes with mmap
9. Using as many processes as there are cores in the machine with fork(), sharing the memory to be sorted by all processes with mmap and using pthread_mutexes to keep track of the number of active threads

This report will outline the performance testing methodology used, the results of each implementation, findings from each implementation and details of the bonus task.

Testing Methodology

For each of the 9 steps, testing the performance of the programme involved measuring how long the programme took to sort an array of 100,000,000 elements. To achieve this, each programme was compiled and timed on the same machine for all 9 steps. The details of the machine's environment is outlined defined in the answer to Question 1 above the report. For an accurate measurement, './time ./a.out 100000000' was executed three times and results were defined by calculating an average of the three measurements. Furthermore, the './time' command produced three different measurements, "real", "user", and "sys". For consistency and accuracy, the "real" measurement will be used for analysis in this report.

Results

Data from the measurements can be interpreted to indicate which parallelism approach is best suitable for the MergeSort algorithm. The results written in a plaintext format can also be found in Appendix A.

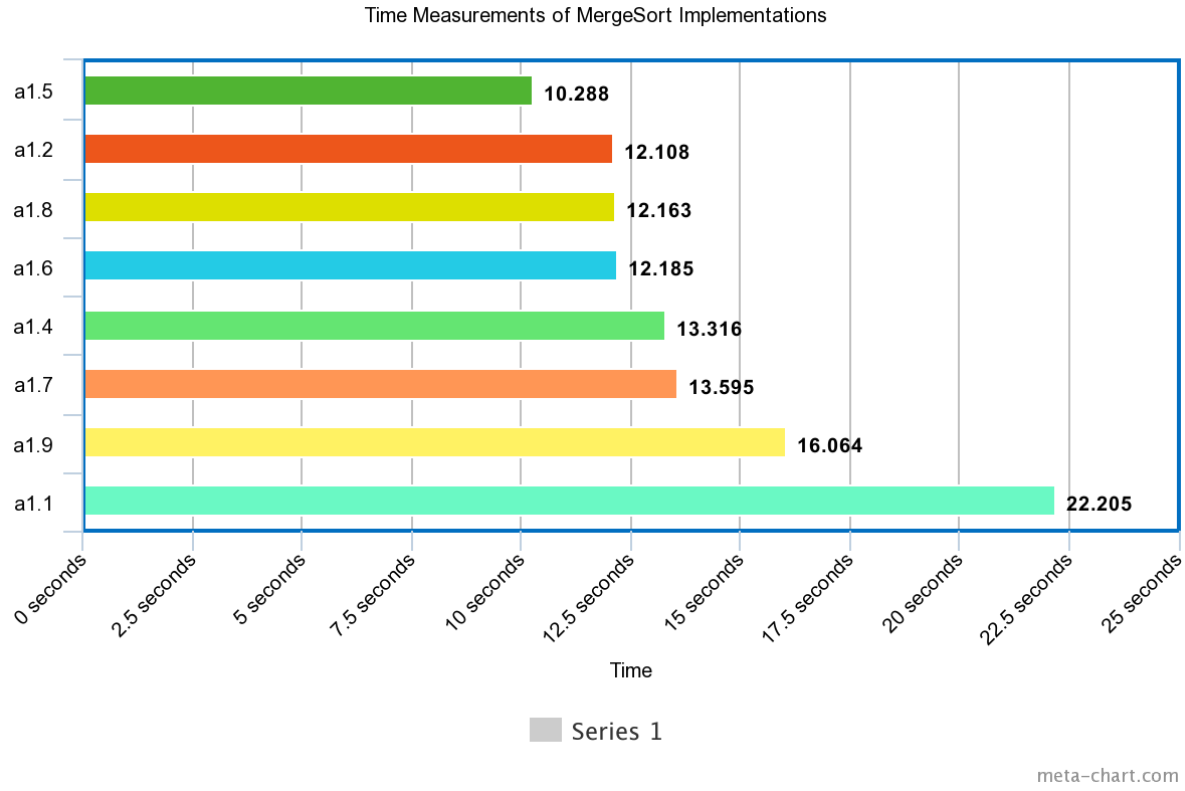


Figure 1: Bar graph of time taken to sort 100,000,000 elements

The result from the third implementation, creating a new thread for every MergeSort() call, was omitted as it lead to a Segmentation Fault.

In addition, as defined in the requirements of the assignment brief, screenshots of the System Monitor programme for steps 4, 5, 7, 8 and 9 have been included in the report and can be found in Appendix B.

Analysis

Implementation 5

As shown in Figure 1 above, the fifth implementation of MergeSort has the greatest performance- where the programme uses as many threads as there are cores in the machine and pthread_spinlocks to keep track of the number of active threads. A comparison

between the fourth and fifth implementations can be used to deduce that spinlocks perform significantly better compared to mutexes (10.288 vs 13.595 seconds). This confirms my answer to the fifth question above the report- that on multi-core systems using spin locks achieves greater performance than using mutexes when the locks are held in place for only short periods of time.

Implementations 2, 6 & 8

It was interesting to discover that the next three implementations to perform the best all involved merely two processes or threads. However, the second, sixth and eighth implementations all lacked mutex locks which may have contributed to this outcome. So it is evident that mutex locks are severely decreasing the performance of the MergeSort algorithm.

In addition, together with the results from Implementation 5, it is evident that, in general, creating new threads is a better alternative performance-wise to creating new processes with fork(), as long as spinlocks are used to maintain the number of active threads. This is shown by the second and fifth implementations achieving lower execution times compared to the sixth, seventh, eighth and ninth implementations.

Furthermore, it can be seen that the execution time of the programme is inversely proportional to the number of threads between running the MergeSort algorithm on one thread and two processes/threads. This is demonstrated by the implementations involving two processes/threads (the second, sixth and eighth) performing almost twice as fast compared to the implementation involving merely one process (the first). However, this is not the case between two and four processes/threads. I believe that it is safe to assume that the overhead of using mutexes and spinlocks is responsible for this behaviour.

Implementations 4, 7 & 9

It was unusual to find that the implementations utilizing more threads/processes (the fourth, seventh and ninth) were performing slower than other optimization techniques that used only two threads/processes. It can be deduced that the mutexes are partially responsible for this, and that using spinlocks would result in better performance as explained above.

It was shown that using a pipe to communicate between processes was better performance-wise than using mmap by the seventh implementation completing the algorithm in a shorter period of time (13.595 seconds) compared to the ninth (16.064 seconds).

Again, it was demonstrated that using threads results in better performance than using processes for the MergeSort algorithm by the fourth implementation executing faster than both the seventh and ninth implementations.

Bonus Implementation

The bonus implementation is similar to the seventh implementation. The programme uses as many processes as there are cores available on the machine, with pipes to communicate between the processes and it uses mmap to share a struct containing a mutex lock and the number of processes.

The difference between the seventh and this bonus implementation is the position of the mutex unlock instruction. The bonus implementation takes advantage of the fact that reading the num_processes value is mostly safe and that the mutex lock can be released straight after it has been incremented by the merge_sort() function. This results in other processes being able to acquire the lock and continue execution.

This implementation of MergeSort() could complete sorting 100,000,000 elements in 7.609 seconds on average.

Appendices

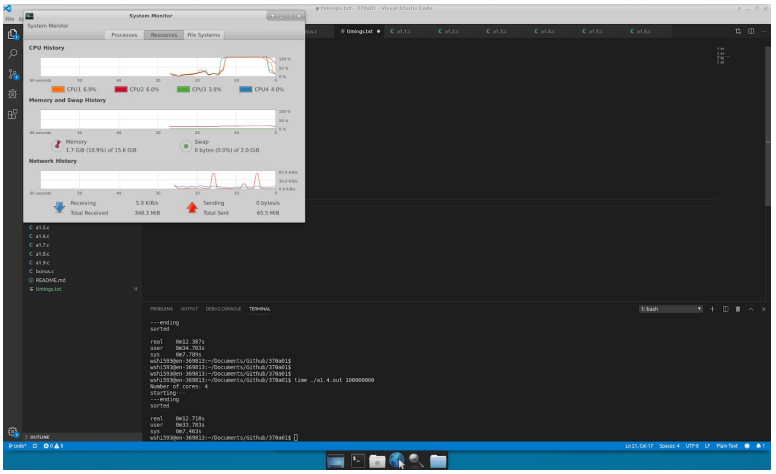
Appendix A - Measurement results in plaintext

Implementation Number	**Measurement Average (in seconds)**
1	Real: 22.205 User: 22.061 Sys: 0.144
2	Real: 12.108 User: 22.213 Sys: 0.236
3	Segmentation Fault Real: 1.831 User: 0.960 Sys: 1.929
4	Real: 13.316 User: 9.822 Sys: 3.21.
5	Real: 10.288 User: 28.629 Sys: 0.288

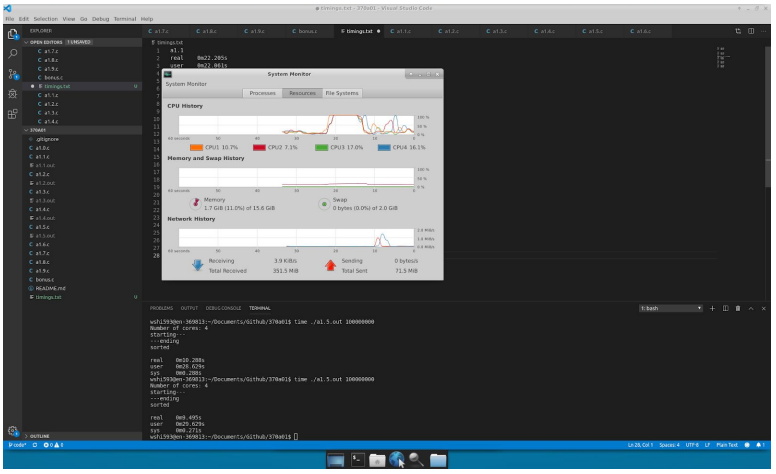
6	Real: 12.185 User: 11.868 Sys: 0.268
7	Real: 13.595 User: 32.627 Sys: 6.679
8	Real: 12.163 User: 22.223 Sys: 0.296
9	Real: 16.064 User: 28.583 Sys: 7.224
Bonus	Real: 7.609 User: 7.123 Sys: 0.307

Appendix B - Screenshots from the System Monitor

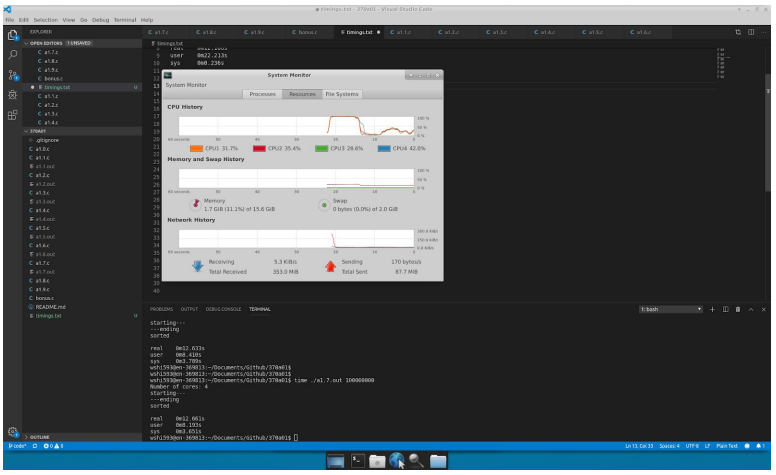
Implementation 4:



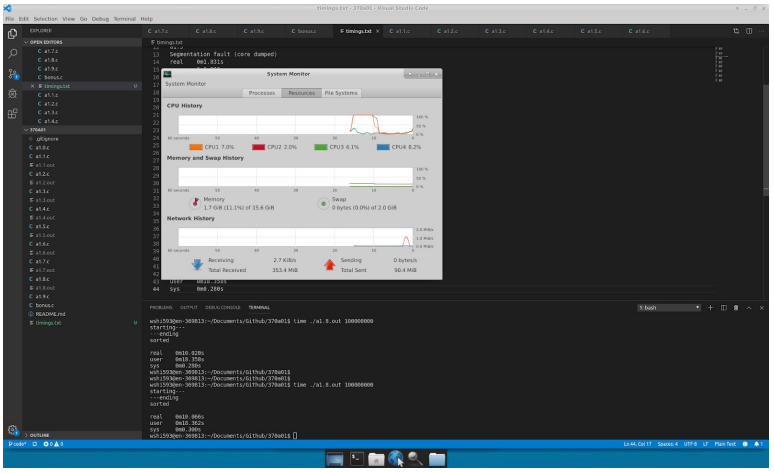
Implementation 5:



Implementation 7:



Implementation 8:



Implementation 9: