Report for Softeng 370 Assignment 1

1. Linux Ubuntu 18.04.2 LTS, 8199476 kilobytes of memory, 8 CPUs.

2. The array size is approximately 1047500 before a segmentation error occurs.

3. The limit is smaller than the amount of available memory out of safety. Running out of stack is better than running out of memory. The latter would affect the OS negatively and the former would only cause the process to crash.

4. When a new thread is started every time merge_sort is called, scheduling becomes difficult for the processor, which slows down the system. Also, only one thread can run on a core at a time, which would make too many threads inefficient. In the case of the merge_sort example, stack space runs out exponentially quickly when more and more threads are called to sort items, and the algorithm in step 3 which demonstrates this can only sort up to 10,000 integers before a segmentation fault occurs.

Step 1:

This step saw the use of mergesort using one thread.

The time it took to run the program to sort 10,000,000 numbers.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq1.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
---ending
sorted

real    0m26.852s
user    0m26.308s
sys     0m0.524s
```

Step 2:

This step saw the use of mergesort using two threads to complete the sort. One thread was in charge of merge sorting the left half, and the other was merge sorting the right half. These halves were then merged.

From running the program on two threads, I have found out that the stack size for each thread can be half of what it would be with one thread to be able to sort 10,000,000 numbers.

The time it took to run the program on two threads to sort 10,000,000 numbers.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq2.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
Starting first thread.
Starting second thread.
Finishing first thread.
Finishing second thread.
---ending
sorted

real    0m14.788s
user    0m26.173s
sys     0m0.608s
```

The “real” time it took to run this program is 14.788s, compared to the 26.852s of running the program on one thread.

Step 3:

This step involved the creation of an algorithm that creates a new thread for every merge_sort call.

I’ve found that with creating a new thread every time I call merge_sort, more and more stack space gets used up. This results in the algorithm only being able to sort 10,000 units before running out of stack space and a segmentation fault occurs.

Step 4:

This step involved the use of as many threads as cores to merge sort. Since a thread count must be kept to ensure this, mutexes were used to ensure only one thread at a time writes to the thread count variable.

Using 8 threads to run MergeSort is significantly faster than using two threads. This time, the algorithm took 9.834 seconds to run, compared to the 14.788 seconds it took to run on two threads.

Each core is being used with around 100% of it’s processing power being occupied.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq4.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 7200000000
starting---
System has 8 cores.
New thread added. This brings the total count to 1
New thread added. This brings the total count to 2
New thread added. This brings the total count to 3
New thread added. This brings the total count to 4
New thread added. This brings the total count to 5
New thread added. This brings the total count to 6
New thread added. This brings the total count to 7
New thread added. This brings the total count to 8
New thread added. This brings the total count to 9
Thread has joined. This brings the total count to 8
Thread has joined. This brings the total count to 7
Thread has joined. This brings the total count to 6
Thread has joined. This brings the total count to 5
Thread has joined. This brings the total count to 4
Thread has joined. This brings the total count to 3
Thread has joined. This brings the total count to 2
Thread has joined. This brings the total count to 1
Thread has joined. This brings the total count to 0
---ending
sorted

real    0m9.834s
user    0m30.738s
sys     0m0.894s
```

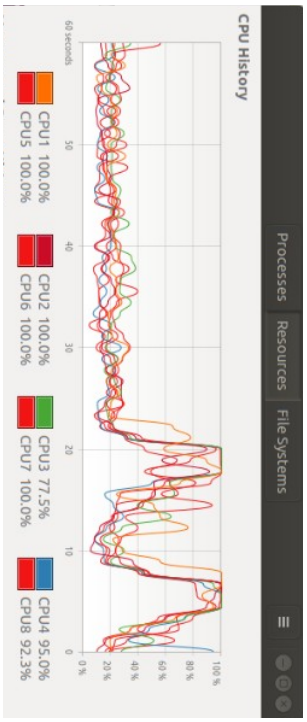
Step 5:

This step was similar to step 4 but using spinlocks instead of mutexes to ensure safe variable access.

Switching from mutexes to spinlocks has reduced the processing time by a very small amount.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq5.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 7200000000
starting---
System has 8 cores.
New thread added. This brings the total count to 1
New thread added. This brings the total count to 2
New thread added. This brings the total count to 3
New thread added. This brings the total count to 4
New thread added. This brings the total count to 5
New thread added. This brings the total count to 6
New thread added. This brings the total count to 7
New thread added. This brings the total count to 8
New thread added. This brings the total count to 9
Thread has joined. This brings the total count to 8
Thread has joined. This brings the total count to 7
New thread added. This brings the total count to 8
Thread has joined. This brings the total count to 7
Thread has joined. This brings the total count to 6
Thread has joined. This brings the total count to 5
Thread has joined. This brings the total count to 4
Thread has joined. This brings the total count to 3
New thread added. This brings the total count to 4
New thread added. This brings the total count to 5
New thread added. This brings the total count to 6
New thread added. This brings the total count to 7
New thread added. This brings the total count to 8
Thread has joined. This brings the total count to 7
Thread has joined. This brings the total count to 6
Thread has joined. This brings the total count to 5
Thread has joined. This brings the total count to 4
Thread has joined. This brings the total count to 3
Thread has joined. This brings the total count to 2
Thread has joined. This brings the total count to 1
Thread has joined. This brings the total count to 0
---ending
sorted

real    0m9.248s
user    0m30.955s
sys     0m1.055s
```



Step 6:

This merge sort uses 2 processes to sort.

This took a similar amount of time as step 2.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq6.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
---ending
sorted

real    0m14.763s
user    0m2.304s
sys     0m0.654s
```

Step 7:

This step involves producing an algorithm that merge sorts using as many processes as cores.

This takes slightly more time than the multithreaded operation. However, not all of the cores were used to their maximum potential, whereas with the multithreaded program, all cores were being used at approximately 100% capacity.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq7.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
Forking left block, number of processes is 1.
Forking right block, number of processes is 2.
Forking left block, number of processes is 3.
Forking left block, number of processes is 4.
Forking right block, number of processes is 5.
Forking left block, number of processes is 6.
Forking right block, number of processes is 7.
Forking right block, number of processes is 8.
Left block fork finished.
Right block fork finished.
Left block fork finished.
Right block fork finished.
Left block fork finished.
Right block fork finished.
Left block fork finished.
Right block fork finished.
---ending
sorted

real    0m10.272s
user    0m2.419s
sys     0m0.725s
```

Step 8:

This step uses the same algorithm as step 6, but instead of using pipes to communicate between processes, shared memory locations are used.

The time it took to run step 8. This is a slightly better time than step 6.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq8.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
Starting right block process.
Starting left block process.
Finishing left block process.
Finishing right block process.
---ending
sorted

real    0m14.588s
user    0m25.838s
sys     0m0.577s
```

Step 9:

This step is the same as step 7, but instead of using pipes to communicate between processes, shared memory locations are used.

This is slightly faster than the step 7 algorithm. I assume this is because piping data to different processes takes more processing time than accessing a shared memory location normally. However not all cores were used, despite 8 processes running.

```
cheesegr8er@TQuPC:~/git/Softeng370Assignment1$ time ./aq9.out 100000000
The stack limit before was: 8388608
The stack limit after is now: 900000000
starting---
Forking left block, number of processes is 1.
Forking right block, number of processes is 2.
Forking left block, number of processes is 3.
Forking left block, number of processes is 4.
Forking right block, number of processes is 5.
Forking right block, number of processes is 7.
Forking left block, number of processes is 6.
Forking right block, number of processes is 8.
Right block fork finished.
Finishing left block process.
Finishing right block process.
Finishing left block process.
Right block fork finished.
Finishing right block process.
Finishing left block process.
Finishing left block process.
Right block fork finished.
Finishing right block process.
Right block fork finished.
Finishing right block process.
---ending
sorted

real    0m9.619s
user    0m27.281s
sys     0m0.812s
```

Conclusions:

The different steps each represent a method for the parallelization of merge sort. Ranked from fastest to slowest:

1. Step 5: 9.248s - Mergesort which uses as many threads as cores, and uses spinlocks
2. Step 9: 9.619s - Mergesort which uses as many processes as cores, and uses mmap.
3. Step 4: 9.834s - Mergesort which uses as many threads as cores, and uses mutexes.
4. Step 7: 10.272s - Mergesort which uses as many processes as cores, and uses pipes.
5. Step 8: 14.588s - Mergesort which uses as 2 processes, and uses mmap.
6. Step 6: 14.763s -Mergesort which uses 2 processes, and uses pipes.
7. Step 2: 14.788s - Mergesort which uses 2 threads.
8. Step 1: 26.852s - Mergesort which uses one thread.
9. Step 3: N/A (Cannot sort 100,000,000 elements) - Mergesort that constantly generates new threads.

From the different multithreaded mergesort versions, I have learned that more threads leads to less processing time up to the total number of cores. However, using too many threads can cause many problems like creating a segmentation error, and consuming large amounts of stack space. Spinlocks seem

provide a small performance boost over mutexes. I believe this is due to the cost of putting threads to sleep and waking them up again with mutexes being greater than the cost of polling with spinlocking.

From the different multiprocess mergesort versions, I have learned that, much like threads, more processes leads to less processing time up to the total number of cores. Writing and reading from pipes is more expensive than simply accessing a memory mapped location.