1.

Environment (Mac Mini M1 2020):

- CPU: 8-core (4 performance and 4 efficiency) CPU
- RAM: 16GB
- Darwin Kernel
- OS Version: macOS Big Sur (11.2.3)
- Virtual Machine?: No

2.

I chose 26 as parameter. This parameter choice was due to the time pressure. In particular, at parameter of 26, step 2 took 10 minutes (and waiting longer than that seemed unreasonable), while consuming around 400 MB of memory on average (so the script wasn't taxing the machine excessively, but it's still quite a bit). So I decided to go with 26.

Other results, while being mostly under 10 seconds, are different enough to be used for making meaningful comparisons.

3.

times() reports CPU time (measured in clocks) used to execute instructions (the program currently reports user time on the console which is CPU time used to execute user instructions, but you can also take system time which is CPU time used to execute system instructions). Gettimeofday() reports the wall/real time used to execute the program. Of course, for those times to correspond to the time used to run the program, you should take starting time and end time, then subtract the result.

4.

The user time reported (clock ticks) indicate that the CPU time spent in executing user instructions (the a1.0 program) increased by a factor of ~2 (the first result ran in 187-20 = 167 clock ticks, while the second one ran in 396-36 = 360 clock ticks), which makes sense because as the input increased by a factor of 2, there are more instructions to execute (more recursion, more insertion sort).

Wall (real) time is much higher in second program, which implies that CPU spent much longer time in kernel in second program than in the first program. This is likely because of the memory pressure, given that the environment was virtual machine with 3 GB memory. Significant portion of the input array may be residing in the disk (virtual memory) rather than RAM, hence taking lots of times when the script tries to access/modify the memory (because hard drive is much slower than primary storage device like RAM).

SEE APPENDIX 0 for results & table (All tasks except 4 completed)

Step 0:
This is the baseline script that we will compare our further optimisations against. It currently just recursively mergesorts in a **single thread**, as noted here:

Process Name	Memory	Threads	Ports
a1.0	273.9 MB	1	10

We also note that **memory usage is around 274 MB**.
However, it should be noted that mac itself is optimizing the program run by allocating some works to different cores as shown in the CPU history below (Appendix 1 – step 0). Core 5 seems to be the primary core that does most of the work though.

The running of this program gave a baseline reference of **wall time (9 secs and 44074 microseconds) and clock time (887 ticks).**

Step 1:
Uses two threads. Each thread works on one half of the input. In the end, one thread merges the result. The expectation is that it works around twice as fast. This is because theoretically, the time taken would be as if we were running script on half the input (not entirely true as we merge at the end in one thread).

This expectation is more or less fulfilled in the running of this program, as the running time is **4 secs and 834071 microseconds** (**almost twice as fast**), and **clock time is 890** (which indicates more or less the same number of instructions).
We note that memory usage is similar (**275 MB**), while number of threads running is **2**.

Process Name	Memory	Threads	Ports
a1.1	275.0 MB	2	11

Again, mac seems to do its own thread scheduling optimization, as 4 cores peaked when running the program (refer to Appendix 1 – step 1).

Step 2
The first thread calls merge sort. The thread processes half the input, and if a thread can be created, we delegate the other half to the created thread – else, we simply make the thread work on that other half too (and so on, recursively). When both halves are sorted (if another thread works on it, we use join to wait for that thread to finish its work), we merge them.

The result is that we have **2596588** threads created in total, and a runtime of **653 secs and 957782 microseconds** with a clock time of **60696**. High runtime is expected as we attempt to create a thread at every level of recursion, which means high number of threads created; creation of thread is expensive (**time**/**memory**/**instructions** wise), and this will far outweigh the potential benefit of having lots of threads (in fact, there are only 8 cores anyways, and since each core runs one thread at a time, having any more threads than 8 threads aren't likely to bring any significant benefits to runtime). We can see that cores 5,6,7,8 are occupied all the time for the duration of the running program (Appendix 1- step 2).

Memory usage (380.7MB) is quite a bit more than previous ones (~270 MB), confirming that thread creation is definitely time consuming. I also observed that the number of concurrently running threads maxes out at 4096 threads.

Process Name	Memory	Threads	Ports
a1.2	380.7 MB	4,096	4,102

Step 3

We limit number of created threads to 8. The current implementation is such that we keep a (mutex protected) counter that ensures threads creations occur until thread count hits 8.

```
hajinkim@Hajins-Mac-mini 370 assignment % ./a1.3 26
start time in clock ticks: 42
finish time in clock ticks: 945
wall time 4 secs and 538147 microseconds
sorted
created this many threads: 8
```

Referring to Appendix 1-step 3, we can see that each of 8 cores becomes occupied when the program runs. The fact that instructions run concurrently on maximum number of cores makes us expect that time taken will be faster. We see that the wall time taken is **4 seconds 538147 microseconds**, which is less than step 1 (2 threads, 4 secs 834071 microseconds). This is only slightly less, but the results are consistently faster than step 1, so it's a performance gain. The reason for small increment in time performance could be due to apple's own optimisations (we observe that 2 threads implementation from step 1 actually runs on 4 cores). Furthermore, the child threads are created only on half of the each recursion level, which means that the right subtree of recursion is still a bottleneck.

Clock times is **903**, slightly higher than task 0 (887) and task 1 (890), but way less compared to task 2 (60696), probably because thread creation increases number of instructions to be run. Memory performance is 274 MB, which is effectively similar to previous ones. This is because the primary cost for memory is the input array.

Process Name	Memory
a1.3	274.5 MB

Task 4

Not completed

Task 5

Step 5 uses two processes to mergesort the input array. Each process sorts each half in a singlethreaded manner, and the parent process merges the result.

We observe that runtime of this step 5 (**18 secs and 570362 microseconds**) is actually significantly higher than runtimes of other steps (<10 seconds) apart from task 2. Furthermore, looking at the **CPU history** (Appendix 1-Step 5) shows that two cores are heavily utilized. **Core 5 and 6 are used at almost the maximum capacity,** which was surprising. As indicated by the red portion of the CPU history, it seems that a lot of the overhead in CPU usage is from system calls to OS kernel when creating custom processes and running things on them (note: red portion indicates amount of CPU time given to Mac

OS X operating system). Also, since I'm using pipe to send little bits of data rather than the whole chunk, it will also add to the kernel level overhead (as pipe related calls are system calls) as well as time required to block for reading/writing.

The number of instructions to be carried out is very similar to previous tasks (**902 cycles**). I was initially confused as to why, as each process gets half the input size (hence cycle counts are expected to be quite less), but I discovered that the significant number of cycles were just from read() and write() operations. Also, interestingly – but as expected – we see two processes in the system monitor. This is because we have **2 processes** running.

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	Architecture	% GPU	GPU Time
a1.5	92.2	9.97	1	0	Apple	0.0	0.00
a1.5	92.2	9.53	1	0	Apple	0.0	0.00

Under memory section, three processes show. This is unexpected because under CPU tab, there are only two processes. We observe that the top two processes' memories add up to ~290, so I think it's safe to assume that it is due to mac's internal memory optimizations.

Process Name	Memory	Threads	Ports
a1.5	145.7 MB	1	8
a1.5	146.4 MB	1	8
a1.5	274.2 MB	1	10

Step 6

This method uses 8 processors to merge sort each of 8 chunks, and uses pipes to pass around result from a processor to other processors. The runtime (**8 secs 473823 microseconds**) is faster than step 5 (18 secs and 570362 microseconds) which is to be expected as 8 processors are used. This result is concordant with the **CPU usage, where 8 cores** are actively used (Appendix 1- step 6). However, compared to the later steps (7, 8), the runtime is still slower. This seems to be due to frequent call to pipe, read, write, which are all system calls. Clock cycle have decreased (**543**) compared to step 5 (**902**), which is to be expected as multiple cores are employed.

Memory usage seems to be concentrated on the parent process (**276.3MB**), which is expected as it depends on other processes to write to the pipe for it to read, and it persists from start to the end.

Process Name	Memory	Threads	Ports
a1.6	276.3 MB	1	10
a1.6	50.1 MB	1	8
a1.6	50.2 MB	1	8
a1.6	51.5 MB	1	8
a1.6	51.1 MB	1	8
a1.6	50.1 MB	1	8
a1.6	51.0 MB	1	8
a1.6	51.3 MB	1	8

Step 7

I initially make a copy of array in the shared memory using mmap. I had two processes. Parent process sorts left, and child process sorts right. Once child process dies, parent

process merges the result and copies the result in the shared memory back to the incoming block's data.

This step is similar to step 5 in that it uses two processes. However, this step uses a shared memory instead of piping. You can observe that step 7 (**5 secs and 186925 microseconds**) runs much faster than step 5 (18 secs and 570362 microseconds), most likely owing to the fact that there are no multiple read() and write() to be executed. However, it's still slower than task 1 and 3 (which uses threads instead of processes) by 0.4~0.7 seconds difference. This is probably due to the overhead in creating a process – creating process is much harder than creating a thread.

Cycles count (**499 cycles**) is much less than others, due to multiprocessing (cycles only counted in the parent thread). In fact, it's **almost half** the cycles count of task 1,3,5. On system monitor, we observe two processes being created, one having a much more memory than the other. This is because my implementation is such that the parent merges the result and copies result to the incoming block, so has quite a bit of more work than child process.

Process Name	Memory	Threads	Ports
a1.7	531.3 MB	1	10
a1.7	103.2 MB	1	8

We can also observe that each process makes a full use of CPU time.

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	Architecture	% GPU	GPU Time
a1.7	100.0	2.44	1	0	Apple	0.0	0.00
a1.7	100.0	3.05	1	0	Apple	0.0	0.00

Step 8

I used 8 processes. Each process mergesorts 1/8 of the array (in the shared memory). Particular process has to wait on particular child (with a specific pid), then execute merge.

This runs very fast **(2 seconds and 441233 microseconds**) compared to earlier ones (>4.5 seconds). This is unlike the previous tasks that run slower than thread implementations due to process creation overhead/pipe read and write. Number of cycles is much less too (**221 cycles**) reflecting that each clock of CPU does work in 8 cores and the CPU time count made on the parent thread is correspondingly less. Referring to Appendix 1 – step 7 shows that **all 8 cores become active**.

We observe from system monitor that there are 8 processes.

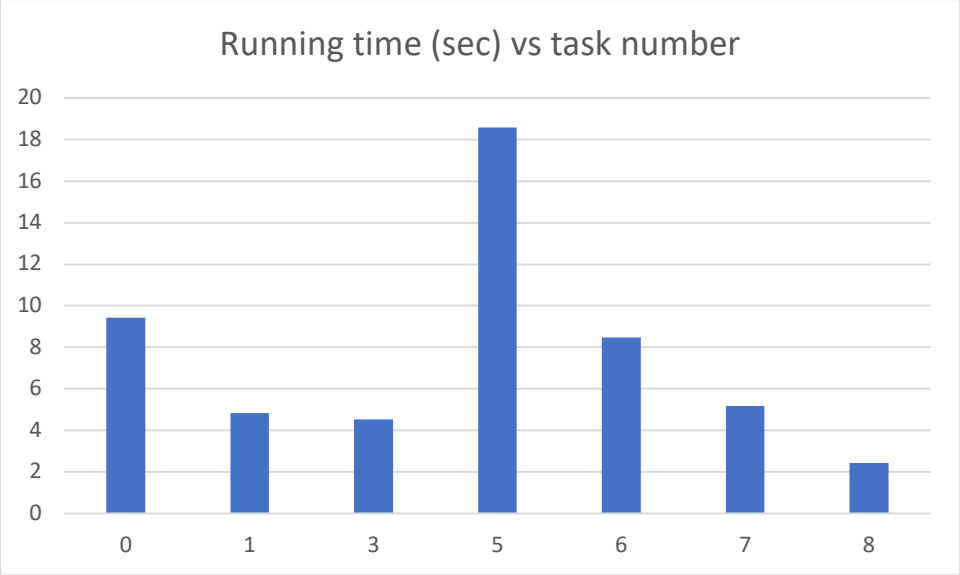
Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	Architecture	% GPU	GPU Time
a1.8	86.6	15.01	1	0	Apple	0.0	0.00
a1.8	86.2	4.06	1	0	Apple	0.0	0.00
a1.8	85.4	4.12	1	0	Apple	0.0	0.00
a1.8	85.2	4.10	1	0	Apple	0.0	0.00
a1.8	85.1	4.04	1	0	Apple	0.0	0.00
a1.8	84.8	4.08	1	0	Apple	0.0	0.00
a1.8	84.0	4.05	1	0	Apple	0.0	0.00
a1.8	83.0	4.07	1	0	Apple	0.0	0.00

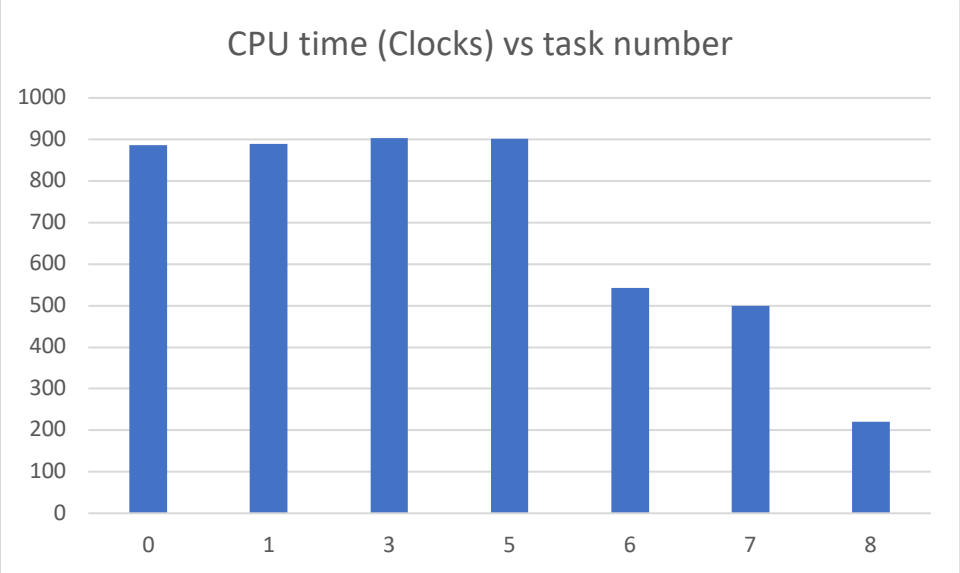
It was interesting noting that 7 processes occupy only ~45 MB of memory while the last process runs with 554 MB of memory. Those 7 processes probably just have that much memory because processes don't copy the whole resource from the parent – they copy on demand, which means they only copy when they modify variables.

Process Name	Memory	Threads	Po
a1.8	44.2 MB	1	
a1.8	48.1 MB	1	
a1.8	42.9 MB	1	
a1.8	42.0 MB	1	
a1.8	45.1 MB	1	
a1.8	44.2 MB	1	
a1.8	45.6 MB	1	
a1.8	533.9 MB	1	
accountsd	11.3 MB	2	

Appendix 0 - Assignment 1 task results

Task	Wall time	Clock ticks	Notes
0	9 secs and 44074 microseconds	929-42 = 887	
1	4 secs and 834071 microseconds	932-42 =890	
2	653 secs and 957782 microseconds	60737-41=60696	2596588 threads created in total
3	4 secs and 538147 microseconds	945-42=903	
4	todo	Todo	
5	18 secs and 570362 microseconds	944-42=902	
6	8 secs 473823 microseconds	582-39 = 543	
7	5 secs and 186925 microseconds	541-42=499	
8	2 secs and 441233 microseconds	263-42=221	

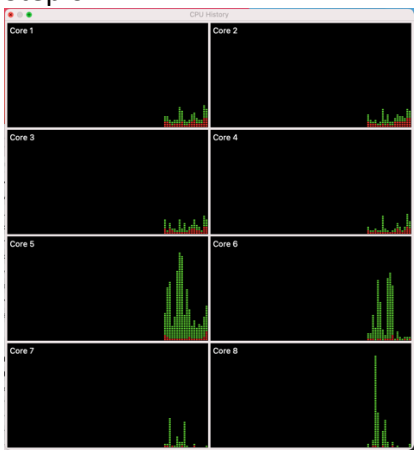




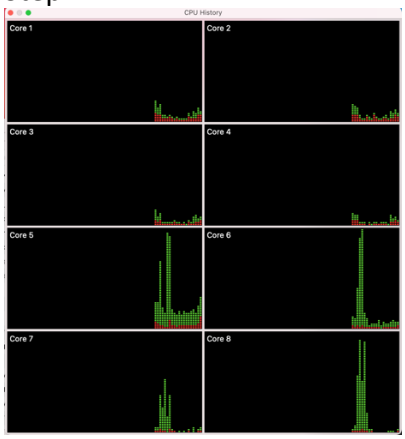
Note: task 2 result has been omitted for easier visual comparison

Appendix 1

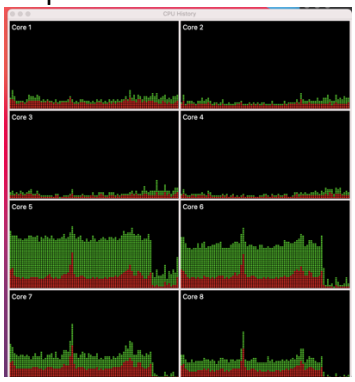
Step 0:



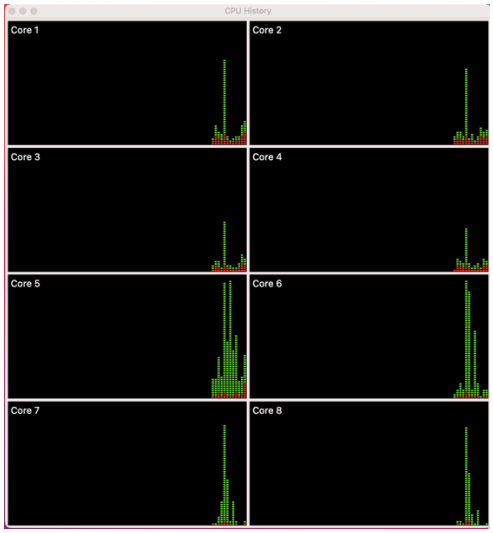
Step 1:



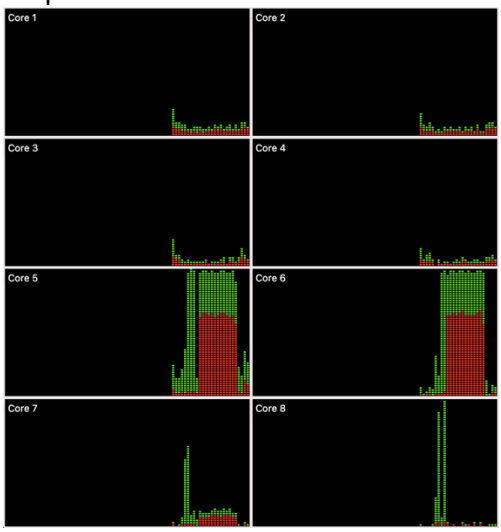
Step 2



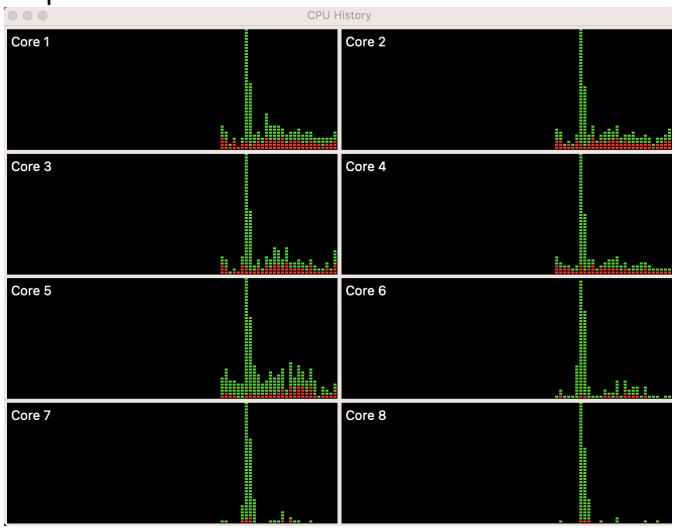
Step 3



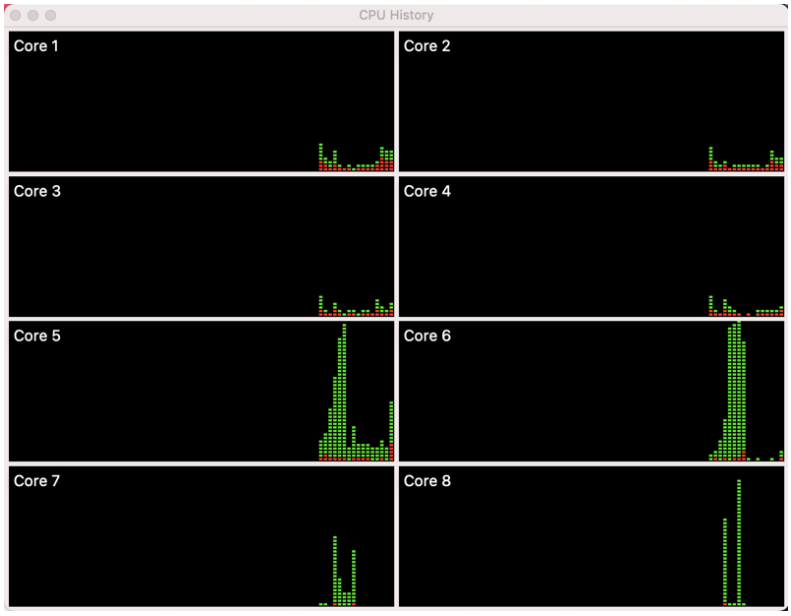
Step 5:



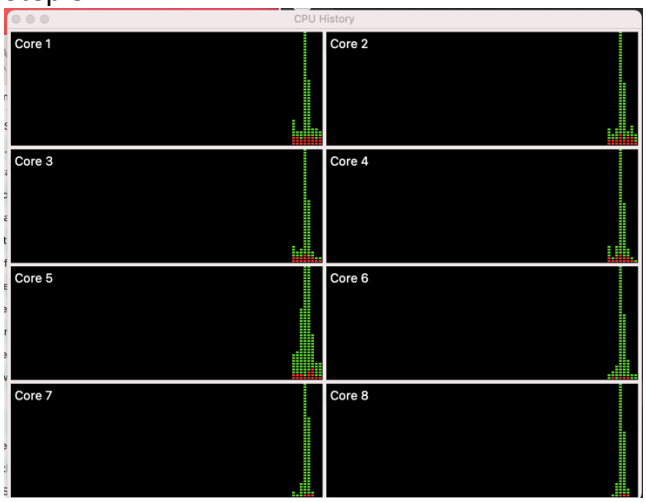
Step 6:



Step 7:


CPU History
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8

Step 8:


CPU History
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8