COMPSCI 340 Assignment 1

10% of your grade

Due date: 9:30 pm Monday 24th August

Introduction

We no longer live in a world where computing speeds increase along with Moore's Law. Instead we maintain increased throughput in our computers by running more cores and increasing the amounts of parallelism. Parallelism on a multi-core (multi-processor) machine is looked after by the operating system.

In this assignment you have to parallelise a simple algorithm in a number of different ways. You then need to compare the different ways and write a report summarising your findings.

This assignment is to be done on a Unix based operating system. I recommend a version of Ubuntu either directly or on a Virtual Machine (as described in the first tutorial). The machine (real or virtual) must have at least two processors or cores. macOS or WSL are also appropriate environments.

Quicksort

The algorithm you have to parallelise is quicksort. Quicksort is *relatively* easy to parallelise. Just to remind you, quicksort recursively finds a pivot and splits the data into values below that value and above that value and then sorts those halves. One of the advantages of quicksort is that it is commonly done in place. No extra space is required to sort the data.

The version of quicksort you have to use is written in C and available on the assignment Canvas page as `a1.0.c`.

Always compile using:

`cc -O2 filename.c -o filename` (you will need to add `-pthread` for some programs).

Do all of the timings on the same (or very similar) machine and don't watch videos or do similar things while you are taking the timings. Take the timings at least 3 times and average them. Carefully record all of the timings in a chart.

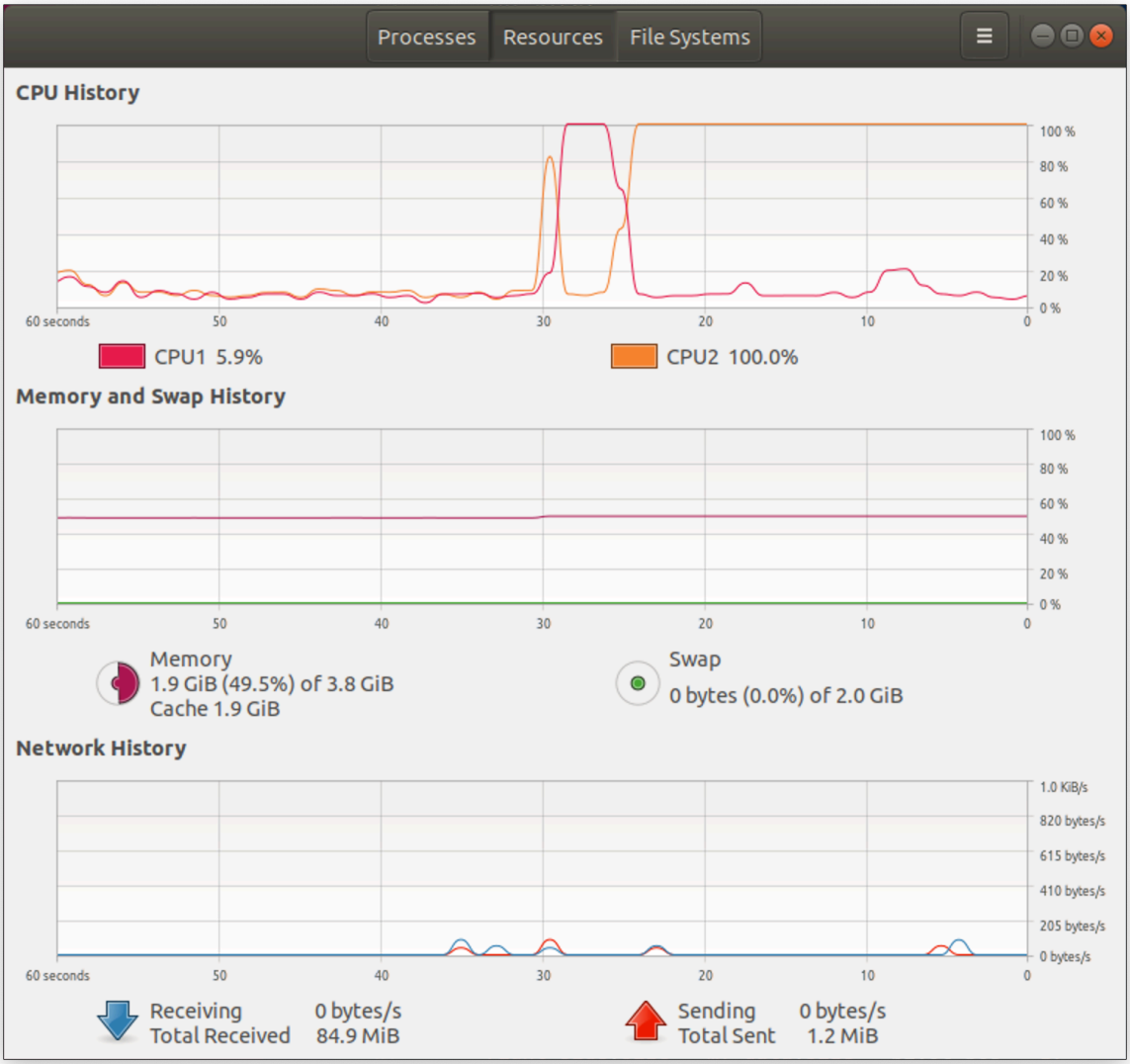
Everyone's results will be different because of the different environments, especially the number of CPUs. So specifying the environment is important: question 1.

Things to do

For the following I recommend keeping the System Monitor running on the screen, displaying the CPU usage. You can start this from the command line as `gnome-system-monitor&`. This won't work in macOS or WSL (you can use their tools instead). In the Resources tab you can see the CPU utilisation for all of your cores, as in the figure on the next page. In the Processes tab you can see the list of processes (which becomes important later in the assignment).

Step 1

Read through and understand the code in the file `a1.0.c`.



Compile the program.

```
cc -O2 a1.0.c -o a1.0
```

An important part of the program is the `is_sorted` function before the end. In all of your programs you will need to call this to ensure that any changes you make to the implementation do in fact return the sorted data.

If you run the program without any command line parameters e.g.

```
./a1.0
```

it will use a default array of only 10 values.

Run it with a larger amount of random data by including the size parameter on the command line e.g.

```
./a1.0 1000
```

runs the program with an array of 1000 random values.

For all of the steps time how long it takes the program to sort 10,000,000 numbers and record the results. The program reports the number of clock ticks the process ran for (sort of). You

should also get **real elapsed time** either by adding system calls to do so or by running the program with the command:

```
time ./a1.0 10000000
```

Repeat this for all of the following steps.

Step 2

Modify the program (call it `a1.1.c`) to use two threads to perform the sort.

You will need to make sure you are running on a machine with at least 2 cores. If you are using a virtual machine you may need to change the configuration to use at least 2 cores.

Read: `man pthread_create`, and `man pthread_join`

On Linux compile with:

```
cc -O2 a1.1.c -o a1.1 -pthread
```

Only start up one extra thread (the main thread is already running). Use one thread to sort the values below the pivot and one thread to sort the values above the pivot. Do NOT recursively create threads. Only make one more thread and when it has finished its job it should exit.

Ensure the array gets sorted. You should use `pthread_join` at some stage.

In your report you must explain the results you see here and compare them to step 1. Did you get much speed up by running the second thread? If not, explain why not. You should refer to what you see happening in the system monitor. What are the cores doing, explain why they are behaving this way.

The value of the pivot is important in quicksort.

Step 3

Modify the program (call it `a1.2.c`) to use as many threads as you can make.

If every time you call the `quick_sort` function you create a new thread to deal with half of the data you will find that you rapidly run out of threads. The solution to this is to *try* to create a new thread every time you call `quick_sort` and check the result of the call to `pthread_create`. If the result indicates failure then just sort both halves as in Step 1. If the call succeeds, sort half in the new thread and half in the existing thread. You should print out the number of threads you have created to prove to yourself that you made a lot.

In your report describe what happens and provide an explanation for the times recorded and any other behaviour you find significant.

Step 4

Repeat step 2 but reuse the second thread. Call the program `a1.3.c`. Create the second thread when your program starts and whenever you call the `quick_sort` function check to see if the thread is busy.

In the second thread you must wait on a condition variable. The main thread can check to see if the other thread is busy (by some value in your code). If it is not busy the main thread should signal the second thread to continue working on the data below the new pivot.

Read: `man pthread_cond_wait`, and `man pthread_cond_signal`

In order to use a condition variable you will also need a mutex lock.

Read: `man pthread_mutex_lock`, and `man pthread_mutex_unlock`

You can easily set up the lock and condition variables using the following code (at the top level, i.e. outside of functions):

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

In your report describe what happens and how this compares to the results in Steps 2 and 3.

Step 5

Rather than using a second thread (as in Step 2) start a second process. Call the program `a1.4.c`. Sort the numbers below the first pivot in the child process and pass the sorted information back to the parent process. You must pass the sorted data back from the child process to the parent using a pipe.

Read: `man fork`, and `man pipe`

In your report explain how this differs from Step 2, both in what is happening and in the timings you collect.

Step 6

It is tricky to keep both processes working in a two process version of quicksort. If we don't share memory between the processes there can be lots of message passing which slows things down.

So instead - determine a minimum size for each quicksort split. If the number of elements above or below the partition is greater than the minimum size create a new process to sort one section, otherwise sort in the current process. And of course the sorted data must be passed back from any child process to its parent process (or other ancestor).

Have each child process report when it starts running and when it finishes, so that you can keep track of the number of processes created. You can also see this in the Processes tab of the system monitor.

Call this program `a1.5.c`.

You should attempt to optimise the minimum size value for your environment. Do this by testing different values. Because of the variance in results, close enough will be good enough.

In your report describe how you determined the minimum size value and comment on how you get the sorted information all the way back to the original process so that the array can be checked for correctness. Say how many processes were created in the optimal case. Also compare this to previous steps.

Step 7

Repeat Step 6 (including finding the optimal split size) but rather than using message passing to send sorted data back, have the entire array in shared memory. i.e. The data is shared amongst all of the processes.

Read: `man mmap`

In your report compare this to step 6 and steps 3 and 4.

Bonus step

Write the quickest version of quicksort that you can based on the original code, i.e. it must use the same pivot function. Call the program `a1.bonus.c`. In your report include a section describing how your program works with its timing results. This section will get you an extra mark if it is faster than the previous programs.

Questions to answer

Include the answers to these questions at the start of your report document. You do not need to include the questions ("TurnItIn" will flag them as copies).

1. What environment did you run the assignment on? Hint: `man uname`, `man free` and `man lscpu`. The output of `uname -a` provides some of this information, `free` provides information on the amount of memory, and `lscpu` provides information on the number of CPUs. Also mention whether you were using a virtual machine and if so say which one. [1 mark]
2. In some of the steps you should find the number of clock ticks recorded similar to other steps and yet the real time is much faster than the corresponding step. Explain why this is so. [2 marks]
3. When developing `a1.4.c`, the first version with 2 processors, I received the following output. Explain what happened. [2 marks]

```
robert@jim:$ ./a1.4
383 886 777 915 793 335 386 492 649 421
start time in clock ticks: 0
finish time in clock ticks: 0
386 383 335 421 492 649 777 793 886 915
not sorted
finish time in clock ticks: 0
335 383 386 421 915 793 777 886 492 649
not sorted
```

Report

Write a report (max. 4 pages) which summarises what you have found out about the different ways of parallelising the quicksort program. You must only include results from the programs you have written (i.e. if you didn't do steps 6 and 7 you should not refer to results from those steps). The report mark will be scaled according to the number of steps you completed.

At the end of the report you should order the techniques from slowest to fastest and include a brief explanation of what is happening in each of them and how that relates to their performance. Include relevant timing information and screen shots from the `System Monitor` (if that helps your discussion).

This report and the questions above must be submitted in a text readable pdf or Word document. This will automatically be sent through TurnItIn when you submit it and it will be checked for uniqueness.

Submission

1. Submit your report (including your chart of timings and the answers to the questions) as a pdf or Word document in Canvas under "Assignment 1 Report". The report must be readable by "TurnItIn" i.e. don't submit an image of your report. If TurnItIn cannot process your document you may get no marks for the assignment.
2. Submit the source code of all the programs from steps 1 to 7 (and the bonus if you did this) in a zip file on Canvas under "Assignment 1 Programs". Your report will not be marked unless you submit your programs and the marker is able to open and inspect them.

The report and every source code file must include your name and login. There is a mark for this.

By submitting a file you are testifying that you and you alone wrote the contents of that file (except for the component given to you as part of the assignment).