COMPSCI 340 Assignment 1

10% of your grade

Due date: 11:00 pm Monday 16th 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 program in a number of different ways and answer questions summarising your findings.

This assignment is to be done on a Unix based operating system. The distributed code runs on Linux, but you can modify it to run on any Unix based operating system such as MacOS. It should run on the Windows subsystem for Linux. To run your code the markers will use the Ubuntu image provided on flexit.auckland.ac.nz. So please ensure that your submitted programs run in that environment. To get the results for this assignment you will need a machine (real or VM) with 4 or more cores.

When you come to time your programs run them all on the same machine.

The incredibly slow insertion sort

The algorithm you have to parallelise is an insertion sort. As you will recall from COMPSCI 130 the insertion sort is an order $O(n^2)$ sort, but it is one of the better such sorts.

The original version for the assignment is `a1.0.c`.

Always compile using:

`gcc -O2 filename.c -o filename -lm` (you will need to use `-lpthread` for some programs, it is slightly different in MacOS).

All the programs (except the first version of `a1.1.c`) should be written to correctly sort the data. If the markers find that a program does not sort the data, you will not get the marks for that step.

Things to do

Step 0

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

Compile and run the program. An important part of the program is the `is_sorted` function before the end. You will always 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 16 values.

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

```
./a1.0 10
```

runs the program with an array of 1024 random values. This is the largest size which will print all of the values.

Find a parameter size which makes the program take more than 30 seconds to complete. On one of my VMs this was a value of 20.

Question 1 [1 mark]:

Describe the environment you used, operating system, number of cores, amount of real memory. What parameter gave a time of > 30 seconds? How long did it take to complete in clock ticks and elapsed time?

You don't need to submit `a1.0.c`.

Step 1

The `a1.0.c` program has several stages. The first stage splits the array of data into 4 roughly equal size bins. (This of course is totally pointless before we start to parallelise the program.)

This step is based on trying to split the data faster. The `split_data` function runs through the whole array and deposits values into one of 4 bins, with values from 0 to 249, 259 to 499, 500 to 749, and 750 to 999.

The file `a1.1.c` contains an attempt at doing this.

Run the `a1.1.c` program with the same parameter as Step 0.

Question 2 [2 marks]:

What happens when the program runs? Explain why?

Modify the program so that the 4 threads successfully split the data into the 4 bins and the data is sorted properly. **Submit this program as `a1.1.c`.** Put your login/UPI in the file comments.

Question 3 [2 marks]:

What did you do to enable the parallel `split_data` function to work properly?

Question 4 [2 marks]:

Is it a good idea to split the data using multiple threads? Explain why or why not, using data you collected. (I recommend you time only the splitting of the data not the sorting in order to answer this question.)

Step 2

From these steps onwards you should split the data using the best technique from Step 0 and Step 1.

Now we want to parallelise the sort. Use one thread to sort each bin using the insertion function. i.e. Four threads each running the insertion sort on a different bin. **Submit this program as `a1.2.c`.** Put your login/UPI in the file comments.

Question 5 [3 marks]:

Explain how you did this, and compare the times you observe with the times from Step 0. Explain why the times are different.

Step 3

Similar to step 2 but use four processes rather than four threads. **Submit this program as `a1.3.c`.** Put your login/UPI in the file comments.

Processes normally don't share memory with each other and so there will have to be some communication between the processes. Hint: `man fork`, `pipe`.

One of the interesting things is that the `fork` system call copies the data in the parent process so that the child can see the data from the parent (at the time of the `fork`). This means the child process does not need to copy data from the parent to the child. However after the child has sorted the data the resulting sorted values have to be sent back to the parent in order for it to do the merge.

Question 6 [3 marks]:

Explain how you set up the pipes and send the sorted data back to the parent. In your answer include how the parent process knows when a child process has completed its sorting, and how it puts the received sorted data into the correct place in the main array.

Question 7 [3 marks]:

Compare the times and memory usage of this program with the earlier steps. Explain the results. N.B. You will need to add the `cutime` to the `utime` to include the time taken by child processes. This also means you have to call `waitpid` before reading the final times.

You can use the `gnome-system-monitor` to visually see the processor (CPU) and memory usage. You can also use the command line `htop` command (not as nice, but works in the terminal).

Step 4

The same as step 3 but rather than passing information back to the parent process we share the memory to be sorted in the processes. Hint: `man mmap, wait`. **Submit this program as `a1.4.c`.** Put your login/UPI in the file comments.

Question 8 [2 marks]

Explain how the parent process sets up the shared memory for the child processes. In particular say whether or not the parent is sharing the same data with all the children, and why you did it that way.

Question 9 [2 marks]

Explain how the parent process knows when a child process has completed it sorting.

Question 10 [3 marks]:

Compare the times and memory usage of this program with the earlier steps. Explain the results.

Question 11 [2 marks]:

Which of the steps do you consider gives the best solution? You need to take into account the overall time taken, the amount of memory used and the ease of programming. This does not need to be the fastest.

Submission

1. Enter your zipped source files in the "Assignment 1 Programs" Assignment on Canvas. Remember to include your login in each file.
2. Enter your answers to the questions in the Canvas Assignment 1 Quiz "Assignment 1". As this is set up as a quiz, I recommend you write your answers in a file and prepare them before running the quiz and pasting the answers in to the corresponding question field.

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).