SOFTENG 370 Assignment 1

10% of your grade

Due date: 9:30 pm Monday 19th 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 the Ubuntu image in the labs (the CS lab version is more recent than the SE lab version).

The merge sort

The algorithm you have to parallelise is the merge sort. The reason the merge sort was chosen is that it is almost the perfect algorithm to parallelise. Just to remind you, the merge sort breaks the data to be sorted into two halves, sorts the halves recursively and merges the two sorted halves back together to give the answer.

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

In all of the work which follows do NOT optimise the C compilations. Always compile using:

`cc filename.c -o filename` (you will need to use `-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.

Things to do

Step 1

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 two 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.

Determine how large an array can be dealt with by this program before you get a segmentation fault.

Segmentation faults happen when something goes wrong with memory accesses or allocations. (A segment is an area of memory.) In our case the program allocates space for the array of values, and extra working space, on the stack, and as the size of the array increases eventually we run out of stack space.

Modify the program (call it `a1.1.c`) to increase the amount of stack space so that the program can at least deal with 100,000,000 numbers. Hint: `man getrlimit` and `setrlimit`.

Time how long it takes the program to sort 100,000,000 numbers and record the result.

```
time ./a1.1 100000000
```

Step 2

Modify the program (call it `a1.2.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. The lab machines have at least 4 cores. If you are using a virtual machine you may need to change the configuration to use at least 2 cores.

Hint: `man pthread_create`, `pthread_join`

Each pthread has its own stack and the standard pthread stack size is very limited (why do you think this is so?). So you will need to increase the stack size. You need to change the pthread attributes to do this.

Hint: `man pthread_attr_init`, `pthread_attr_setstacksize`

Time how long it takes the program to sort 100,000,000 numbers and record the result.

Step 3

Modify the program (call it `a1.3.c`) to use a new thread every time you call `merge_sort`.

This is really not a good idea. Use what you find out to answer question 4.

Step 4

Modify the program (call it `a1.4.c`) to use as many threads as there are cores to perform the sort, and no more.

You will need to make sure you are running on a machine with at least 4 cores. The lab machines have at least 4 cores. If you are using a virtual machine you may need to change the configuration to use at least 4 cores (if your computer can do this).

First you need a way to determine from within the program how many cores you have in the environment you are using. Hint: `man sysconf`.

You don't have to worry about the other work going on in the computer just proceed as if all cores are available for your sorting program.

For this step you MUST use some shared state to keep track of how many threads are currently active. Every time a new thread is started you should add one to a counter and every time a thread stops running you should reduce that counter. Whenever you are about to call `merge_sort` you either start a new thread (if you haven't reached the maximum number of cores yet) or use an ordinary function call to `merge_sort` from within the current thread).

The problem with this approach is that multiple threads can be accessing the shared state at the same time, so you will need to provide mutual exclusion over the thread counter. Hint: `man pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`.

Time how long it takes the program to sort 100,000,000 numbers and record the result. You should also take a screen shot of the `System Monitor` program showing the `Resources` tab as your program runs. This will prove that all cores are being used.

Step 5

You may have been tempted to ignore locking the shared state in step 4. That is not a good idea. One way you can improve performance is to move to spin locks instead of the heavy weight mutexes.

Modify the program (call it `a1.5.c`) and use spin locks to protect the shared state. Hint: `man pthread_spin_init`, `pthread_spin_lock`, `pthread_spin_unlock`.

Time how long it takes the program to sort 100,000,000 numbers and record the result. You should also take a screen shot of the System Monitor program showing the Resources tab as your program runs. This will prove that all cores are being used.

Step 6

Go back to step 2 and modify the program (call it `a1.6.c`) to use two processes rather than two threads.

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.

Time how long it takes the program to sort 100,000,000 numbers and record the result.

Step 7

The same as step 6 but use as many processes as the machine has cores. Call the program `a1.7.c`. There is a difficulty here as after a `fork` each process has its own values for how many cores are currently being used. How will you deal with this?

Time how long it takes the program to sort 100,000,000 numbers and record the result. You should also take a screen shot of the System Monitor program showing the Resources tab as your program runs. This will prove that all cores are being used.

Step 8 & Step 9

These are similar to steps 6 and 7 in that they both use processes rather than threads. However rather than passing information back to parent processes we share the memory to be sorted in all of the processes. Hint: `man mmap`. Call the programs `a1.8.c` and `a1.9.c`. In `a1.9.c` you have to worry about the shared state as in step 7.

Time how long it takes the program to sort 100,000,000 numbers and record the result. You should also take a screen shot of the System Monitor program showing the Resources tab as your program runs.

Bonus step

Write the quickest version of merge sort that you can based on the original code. Call the program `a1.bonus.c`. In your report include a section describing your program 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. What approximate size for the array can you get to in the original program before a segmentation error occurs? [1 mark]
3. Why is the limit on the stack size smaller than the amount of memory actually available to the stack (as you proved in step 1)? [1 mark]
4. Why is it not a good idea to start a new thread every time `merge_sort` is called? You should mention the things which go wrong. [1 mark]

5. Normally spin locks are regarded as wasteful. Why is the use of spin locks in Step 5 acceptable? [1 mark]

Report

Write a report (max. 4 pages) which summarises what you have found out about the different ways of parallelising the merge sort program. You must only include results from the programs you have written (i.e. if you didn't do steps 8 and 9 you should not refer to results from those steps). [10 marks]

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`.

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 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 9 (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.

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).