SOFTENG 370 Assignment 1

10% of your grade

Due date: 9:30 pm Friday 26th March

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. 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.

The hybrid sort

The algorithm you have to parallelise is a hybrid merge sort. Read the distributed source code to understand the algorithm.

The version of the sort 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 -lm` (you will need to use `-pthread` for some programs, it is slightly different in MacOS).

Do all of the timings on the same machine and don't watch videos or do similar things while you are taking the timings. Take timings from each Step until you are confident they are consistent.

You will notice that the whole program could be much more efficient than it is, e.g. the merge function allocates an array every time it is called. For our purposes this doesn't matter.

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.

Determine how large an array can be dealt with by this program before you get into memory difficulties. This will depend on the amount of RAM your machine has. N.B. It is possible for your machine to become unusable while doing this, so I strongly recommend saving any work before running the program, and only incrementing the parameter by one each time until things get pretty slow.

On my VM with 3 GiB of memory I hit this with a parameter of 24 (i.e. 25 was too much).

If you find the sort is taking a long time e.g. more than a minute, even before running into memory problems you can stop and use the current parameter instead.

Record the number you find, you will use it for all Steps which follow. You must write the programs and run them with this parameter, recording the times reported.

You should also take screen shots of the `gnome-system-monitor` program (or equivalent) showing the Resources tab as your programs run. This will provide information on the processor and memory usage.

Step 1

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.

Hint: `man pthread_create`, `pthread_join`

Of course you only need to create one extra thread to use two threads to perform the sort.

Step 2

Modify the program (call it `a1.2.c`) to use a new thread every time you call `merge_sort`.

If every time you call the `merge_sort` function you create a new thread to deal with half of the data you may find that you rapidly run out of threads. The solution to this is to try to create a new thread every time you call `merge_sort` and check the result of the call to `pthread_create`. If the result indicates failure then just sort both halves with the current thread as in Step 0. 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 3

Modify the program (call it `a1.3.c`) to use 8 threads to perform the sort.

Step 4

This is the most complicated step, and so you may want to complete the others first.

Modify the program (call it `a1.4.c`) to use 8 threads from a thread pool. Create a thread pool of 7 extra threads and whenever you are about to call `merge_sort` if there is a free thread in the pool allocate that to sort half of the data, otherwise sort both halves in the current thread. When a thread has finished its current work it goes back into the thread pool available to be used again.

For this step you MUST use some shared state to keep track of the thread pool. You should use condition variables to start and stop threads. All shared state must be protected with locks (mutexes).

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 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;
```

Do not use an existing thread pool implementation.

Step 5

Go back to step 1 and modify the program (call it `a1.5.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.

Step 6

The same as step 3 but use 8 processes. Call the program `a1.6.c`.

Step 7 & Step 8

These are similar to steps 5 and 6 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 `a1.7.c` and `a1.8.c`.

Bonus step

Write the quickest version of the 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 your previous programs. Show the results in a table.

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. **Do not show the output of these commands.** Summarise the information on the number of CPUs, the amount of memory, the version of the operating system and anything else you consider important. Also mention whether you were using a virtual machine and if so say which one. [1 mark]
2. What is the parameter number you found in Step 0? Was this value chosen because of time or memory pressure? [1 mark]
3. In your own words explain the different times reported by the `times` and `gettimeofday` functions. [1 mark]
4. On a virtual machine with 3 GiB of memory, I ran the a1.0.c program with input of 24 and then 25. This is the output I received:

   ```
   robert@ubuntu:A1 2021$ ./a1.0 24
   start time in clock ticks: 20
   finish time in clock ticks: 187
   wall time 2 secs and 90922 microseconds
   sorted
   robert@ubuntu:A1 2021$ ./a1.0 25
   start time in clock ticks: 36
   finish time in clock ticks: 396
   wall time 111 secs and 614577 microseconds
   sorted
   ```
   Explain these timing values. [2 marks]

Report [10 marks]

Write a report (max. 6 pages, including diagrams but excluding appendices e.g. tables) which summarises what you have found out about the different ways of parallelising the sort program. You must only include results from the programs you have written. The report mark will be scaled according to the number of steps you completed.

Include a table ordering the techniques from slowest to fastest.

Provide a brief explanation of what is happening in each step 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 8 (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).