ENCE360 Assignment 2024

Introduction

This assignment comes with a single program, `serial.c`, that completes a menial task - numerical integration. As is, the program prompts the user for a number range, a number of slices, and the index of the function it should integrate over this range.

We will modify this file in two main ways:

- We will multithread the integration. This will simply make it run faster.
- After reading the user's prompts, the calculation will be handled by a child process. The parent process will take user requests, and spawn a child process to handle the task. This is similar to how an operating system's terminal works.

We will then combine these two modifications together.

Notes:

- All programs will need to be compiled with the `-lm` flag on the end. All programs besides `serial.c` will also need the `-lpthread` flag.
- All programs should compile with `-Wall -Werror`, so fix those warnings!

1. Multithreaded Execution

Your first task is to make a faster version of the supplied program. Make a copy of the original `serial.c`, this time named `thread.c`.

Your program should contain a `#define` for the number of threads to split the work between. Split the work of the integration equally between a fixed number of threads by splitting up the range. For example, the prompt `gauss -1 1 16` run with 4 worker threads should split the 16 slices evenly, 4 slices per thread. You can ignore the case where there are fewer slices than threads.

You ***must*** declare a variable within `main()` that all threads write to when completed, the access to which is controlled with a mutex. While this is not strictly necessary for a simple integration, it is for the general case where the threads take different times to execute, or the computation has more complex results.

I suggest you define a `struct` containing the arguments we want each thread to have access to, and make an array of these within your `main()` function. The `integrateTrap()` function will need to be modified to take a pointer to this `struct` as its argument, as a `void*`.

Be sure to check your implementation for correctness. You should notice a significant speedup roughly proportional to the number of threads used - e.g., using 4 threads, the program should run 4 times faster than `serial.c`.

2. Spawning Task Processes

Make another copy of the original `serial.c`, this time named `process.c`. This program will run the (serial) function `integrateTrap()` inside a child process. The parent process should then re-prompt for input. This means you'll be able to have several queries running at once. Your code should have a maximum number of worker processes - if the number of children is already at this maximum, it should not prompt for input until one of the children finishes.

As a starting point, I suggest you `#define` a `MAX_CHILDREN` macro, and declare a `static` global semaphore `numFreeChildren`. The next step would be to register a signal handler that responds appropriately when a child completes - you may just use `signal()` rather than `sigaction()` if you prefer. There are more steps beyond this, but hopefully it's enough to get you started.

3. Multithreaded Worker Processes

The final step is to combine these two modifications into a third new file, `processThread.c`. This should allow multiple jobs to be run at once, *and* be faster than `process.c`, especially when handling one task at a time.

This one should only require a few modifications given the code you've already written. Again, make sure you check for correctness by comparing it to the supplied code's answers.

4. Report

Once you've written these three programs, also write a short, 3-4 page (excluding title/references) report describing and explaining the performance differences you see between them. In order to test this, you'll want to remove the delay caused by a human typing in the tasks. You can do this by feeding an input file in as follows:

```
./a.out < testCases.txt
```

You should test with more than just the sample `testCases.txt` provided. Your report should contain at least two graphs showing how the program performances compare. You should also include reputable references (in IEEE format) backing up your explanations of the observed behaviour.

Depending on the computer you're running your tests on, varying both the number of threads and the maximum number of child processes may produce very different results, which could be good to talk about.

Avoid testing your code on a virtual machine (for example, though WSL on Windows) - running the program natively (for example, on the Linux lab machines) will give you much more to discuss in your report.

You will submit both this report and your source files. Make sure your code is laid out sensibly and is easy to follow.

UC Dishonest Practice Policy:

Plagiarism, collusion, copying and ghost writing are unacceptable and dishonest practices.

- Plagiarism is the presentation of any material (test, data, figures or drawings, on any medium including computer files) from any other source without clear and adequate acknowledgment of the source.
- Collusion is the presentation of work performed in conjunction with another person or persons, but submitted as if it has been completed only by the named author(s).
- Copying is the use of material (in any medium, including computer files) produced by another person(s) with or without their knowledge and approval
- Ghost writing is the use of another person(s) (with or without payment) to prepare all or part of an item submitted for assessment.
- LLMs (e.g. ChatGPT, Claude, Llama) and other similar AI tools *may* be used, but nothing they generate should be submitted. You must have manually written your submitted code and report yourself. Doing otherwise will be treated as both plagiarism and ghost writing.

Do not engage in dishonest practices. The Department reserves the right to refer dishonest practices to the University Proctor and where appropriate to not mark the work.