ENEL464 - Threading

1 Introduction

Often when writing computationally heavy applications, it is desirable to leverage the multicore nature of modern CPUs to run calculations in parallel. This can be done inside a single process by using *threads.* A *thread* starts at a specified function and continues to execute until that function returns. Your program will always have at least one, 'main' thread running the main() function. When this thread exits, all other threads are terminated. Threads are useful because they execute in parallel. The operating system can schedule threads to be run on different CPU cores simultaneously allowing for up to n times speed improvements (for n cores) [1].

The operating system kernel provides functions to create new threads in your program on the fly. In C on Linux, this is provided through the POSIX Threads (pthread) library, and in C++ using the standard library (thread). Both of these methods take a pointer to a function that is executed as soon as the thread is created. See 'threads.c' for an example using POSIX threads. It works by creating a collection of worker thread which solve the problem, and then waiting until they have finished. The waiting method is the 'join()' function which suspends the main thread and so doesn't waste extra CPU time, and then returns once the thread has finished (joined).

2 Dangers

Multithreading can be a very effective means of improving program performance, however, there are some risks associated with this. Different sections of code can be running simultaneously on different CPU cores in your computer. This means that two threads could try to modify the same piece of memory at the same time. Worse, thread behaviour can also depend on the order in which the threads are running. This is called a race condition: two threads, A and B, are racing to completion, but the behaviour changes if A wins or B wins. This can lead to very complicated and hard to diagnose bugs.

Some of these risks can be mitigated using *synchronisation primitives.* These are kernel constructs that guarentee behaviour across threads. For example, the *mutex* can be used to make a resource *mutually-exclusive*, that is, to allow acces to only one thread at a time. *Semaphores* and *condition variables* are further examples. In the context of the 464 assignment, care must be taken to ensure that all calculations are being done from the same iteration. This is because an iteration depends

[1]Note: threading (or multithreading) is different from Intel HyperThreading or AMD Simultaneous Multithreading. These are both CPU hardware architectures that allow for single physical core to appear as two logical cores to the operating system. This will affect your programs, but in a different way.

on the previous state, thus they cannot be calculated at the same time. This does not apply *within* the iteration though.

3 Troubleshooting

When running into trouble with multithreading, it is often helpful to scale back towards a minimum workable example until the problem disapears. For example, if running into issues with 10 threads, does running with 1 child thread reproduce the issue? Does running the same code in just the main thread work? Try to narrow down where the problem is occuring. Perhaps you are trying to simultaneously access a shared resource (remember: variables/memory are a shared resource too!) and either need to ensure they are not being simultaneously read and written to, or add a mutual exclusion barrier.

4 Further reading

- `http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html`
- `https://www.geeksforgeeks.org/multithreading-c-2/`
- `https://en.cppreference.com/w/cpp/thread/thread`
- `https://en.wikipedia.org/wiki/Thread_safety`