Distributed and Paralel Tech: Pthreads Lab

The goal of this lab is to refamiliarise yourself with the basics of multithreaded programming: threads, mutexes (locks), and condition variables. We will use C (and this is also a chance to refamiliarise yourself with that and ask any questions) and the pthreads library.

Remember when compiling to link in pthread with `-lpthread` (and for the heat eqn you might need mathlib with `-lm`).

Parallel 1D Heat Equation

In this part we will look at parallelising a classic problem: discrete solutions to differential equations. This type of workload is extremely common in high performance computing (which does a **lot** of fluid dynamics).

It is not important you understand the physics behind the heat transfer and I give starter code below.

Here we consider a bar with a given initial temperature profile. At either end of the bar (the *boundaries*), we assume there is a fixed `0` temperature.

To computationally solve differential equations (that describe heat diffusion), we usually use Euler's method (or similar) that discretises the problem: e.g. we ask what the next value should be for a point over a (ideally very small) time step given the current conditions.

The main idea is to divide the bar into grid cells (slices), and calculate empirically what the expected temperature after t time units will be. The smaller the grid cells the better accuracy we can have, but this results in a lot more compute. Each cell requires its left-and-right cell values (or boundary) in order to compute the value for the cell (**Think about**: why might this be challenging for parallelism?).

Here we use the simple update function:

$$t_{new} = t_{cell} + (K \times time\_step/cell\_size \times (t_{left} - 2t_{cell} + t_{right}))$$

What we expect to see is that the heat normalises over the bar and stays warmest in the middle while gradually getting colder towards the boundaries (since these are fixed to 0).

Parallel Heat Equation

Starter code is available on Moodle and below. Please check you understand the code and ask for help if you need a refresher on C programming.

Your task is to use pthreads to parallelise this code. Think about:

1. Where to introduce parallelism? Is it in the spatial direction, e.g. different threads handle different parts of the bar? or in the time direction, e.g. can we compute multiple time steps at once?
2. What sort of thread coordination might be required. How will we ensure values are synchronised (if required)?
3. How you will divide the work between threads: how much work are they doing?
4. How and when you will start and stop threads?

You should check your parallel version gets the same final state as the sequential version and see if there are any speedups (you might need to move to a larger initial state depending how many threads you use).

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

// Model parameters
static float BARLEN = 100.0f;
static unsigned NCELLS = 100;
static float TIMESTEP = 0.05f;
static float MAXTIME = 200.0f;
static float TEMP_COEFFICIENT = 1.0f;
static unsigned PRINTEVERY = 20;

// Calculate the next temperature for a cell of size "dist", over timestep ts
float nextTemp(float ts, float dist,
               float currentT, float leftT, float rightT) {
    float f = (ts * TEMP_COEFFICIENT) / dist;
    float adjustment = f * (leftT - 2 * currentT + rightT);
    return currentT + adjustment;
}

void initialiseTemp(float* cells) {
    for (unsigned i = 0; i < NCELLS; i++) {
        cells[i] = 0.5 + 0.2 * sin(20 * i);
    }
}

void printBar(float* cells) {
    printf("-------\n");
    printf("|");
    for (unsigned i = 0; i < NCELLS; i++) {
        printf(" %0.3f |", cells[i]);
    }
    printf("\n-------\n");
}

int main() {
    float buf1[NCELLS];
    float buf2[NCELLS];
    float* cells = buf1;
    float* prev = buf2;

    initialiseTemp(cells);
    printBar(cells);

    unsigned steps = 0;
    for (float t = 0; t < MAXTIME; t += TIMESTEP, steps++) {
        // Flip current and prevous time step buffers
        float* tmp = prev;
        prev = cells;
        cells = tmp;

        for (unsigned i = 0; i < NCELLS; i++) {
            // Handle left boundary
            if (i == 0) {
                cells[i] = nextTemp(TIMESTEP, BARLEN/NCELLS, prev[i], 0, prev[i+1]);
                continue;
            }

            // Handle right boundary
            if (i == NCELLS-1) {
                cells[i] = nextTemp(TIMESTEP, BARLEN/NCELLS, prev[i], prev[i-1], 0);
                continue;
            }

            // Everything else
            cells[i] = nextTemp(TIMESTEP, BARLEN/NCELLS, prev[i], prev[i-1], prev[i+1]);
        }

        if (!(steps % PRINTEVERY)) {
            printBar(cells);
        }
    }

    printBar(cells);

    return 0;
}
```

Implementing Async + Futures

Just like sequential programming, a large part of parallel programming is building useful abstractions that are easy to reason about. One abstraction is the `future` which represents a computation that might still be happening (concurrently). The idea is that a user would write code such as:

```
Future f = async(fn);
...
void* ret = future_get(f);
```

If the value for `f` is produced by the time the code gets to the `future_get` then the value is read and the code can move on. Else the thread waiting for the value `blocks` and is woken when the value becomes available. This is a very useful construct since you do not need to handle thread creation, explicit value synchronisation, or thread deletion.

In this part of the lab we will build our own asyc + future structure (to show that threads, locks and condition variables are a great basis for other abstractions). This is not expected to be a fully fledged implementation, e.g. we don't need to handle all possible error cases and if you forget to tidy up some resources then don't worry too much.

Future Data Type

We will start with a future datatype following the template below. Think about how to implement the blocking on `get_future` if the value isn't available, and how you will know when the value becomes available.

Write a simple test program to check your future works. This does not need to be parallel at this stage.

```
typedef struct future {

} Future;

Future* new_future() {
}

void* get_future(Future* f) {
}

void set_future(Future* f, void* val) {
}
```

Async Function

Extend your implementation with function that simplifies thread creation and waiting for completion: It should take a function pointer and an args pointer (`void*`) and create a thread to execute the function that will write into the future when it is done.

A template is provided below as a hint for how this program could be structured.

A starter program is below:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* foo(void*) {
    unsigned* myInt = malloc(sizeof(unsigned));
    myInt = 42;
    return myInt;
}

// Arguments to pass to the thread
struct targs {
    Future* f; // Future to write the results to
    void*(*func)(void*); // Function to execute
    void* fnargs; // Arguments for the function
};

void* thread_runner(void* args) {
    struct targs* as = args;
    // What else does the thread need to do
}
```

```
Future* async(void*(*fn)(void*), void* args) {
    // TODO Careful with memory management: async might return before the thread
    // executes so anything passed to it needs to remain alive (i.e. on the heap)
}

int main() {
    Future* f = async(foo, NULL);
    int* v = get_future(f);
    printf("Got %d\n", *v);
    free(v); free(f);
}
```