COMP5426

Assignment 2 – Pthreads Red Blue Computation

A. Problem definition and Requirements

In the Pthreads Red/Blue computation most requirements are the same as the requirements in the first assignment which requires us to use process to perform the computation. The basic requirements are listed as below.

First, we need to initialize a N by N size board with red, blue, and white colours separated in the board just like the assignment 1 required. Then, the red can move right if its right cell is in white, the blue can move down if the cell below it is white, and colours can wraparound to the opposite side when reaching the edge. Red and blue cells can only move one time in one iteration, we will first move the cells in red, and then move the cells in blue.

Third, the board is separated into T by T tiles. And the board must perfectly overlaid with these tiles, each tile will have N/T by N/T cells, if in any tiles, cells in one colour (red or blue) are more than the c% which is a input threshold value, the computation is going to be terminated.

Forth, we need to count the iterations we have performed. If the number of iterations is larger than the predetermined maximum iterations, the program must terminate the computation even no tiles reach threshold value.

Last, the most important one is that I should use Pthreads to implement the red/blue computation. As in Pthreads, there is no need to use MPI to communicate, data are in the shared memory, I need to focus on implementing the mutex lock to let every thread finish their computation correctly.

B. Parallel algorithm design

This section is focusing on the designing of the parallel algorithm of the computation. I am going to describe the details about how I distribute workloads to each thread and achieve the load balancing.

I need to think about task assignment in two ways, one is the assignment of rows, and the other is the assignment of tiles. In the assignment requirement Each thread can only handle at most one more row or tile than the other threads. Due to the nature of shared memory, I don't need to worry about if a row is assigned to a different thread than the tile it belongs to would cause problems. Therefore, the distribution method is very clear and simple. For rows, I just need to calculate the rows / t, and rows % t(rows represent the number of rows, t represents the number of threads). Then for every thread it can get the number of rows / t rows, if the rows % t does not equal to zero, then I just distribute the remaining rows one by one to each thread, in this case I can guarantee that every thread get nearly the same number of rows, the difference is no larger than one row per thread. For tiles, I use the same way to distribute the tiles to each thread.

By implementing this distribution method, I can make sure that every thread completes nearly the same amount of red/blue computation work, and this program does its best to achieve the load balancing goal. The Fig.1 shows implementation of the distribution function.

```
int *distributeWorkToThread(int threadsNumber, int workNumber) {
    int *work = (int*)malloc( size: sizeof(int) * (threadsNumber));
    int a =  workNumber / threadsNumber;
    int b = workNumber % threadsNumber;
    int i;
    for (i = 0; i < threadsNumber; i++) {
        work[i] = a;
    }

    for (i = 0; i < b; i++) {
        work[i] = work[i] + 1;
    }
    return work;
}
```

Fig.1 Distribution function

C Implementation and Test

C.1 Implementation

Struct for thread data: id to represent the thread id, start_row and end_row to represent the start location and the end location of the row a thread assigned, start_tile and end_tile to represent the start location and the end location of the tile a thread assigned.

```
// struct for thread data
struct thrd_data{
    int id;
    int start_row;
    int end_row;
    int start_tile;
    int end_tile;
};
```

Fig.2 Thread struct

Assigning rows and tiles to each thread, then using “redBlueComputation” to perform the computation work.

```
// distribute the workload among the threads
for (i = 0; i < n_threads; i++) {
    //assign id
    t_arg[i].id = i;
    if (i == 0) {
        //first thread start from row 0, tile 0
        t_arg[i].start_row = 0;
        t_arg[i].start_tile = 0;
    } else {
        //other thread start from the end of previous thread
        t_arg[i].start_row = t_arg[i - 1].end_row;
        t_arg[i].start_tile = t_arg[i - 1].end_tile;
    }
    //end = start + num of work assigned
    t_arg[i].end_row = t_arg[i].start_row + row_for_thread[i];
    t_arg[i].end_tile = t_arg[i].start_tile + tile_for_Thread[i];
    pthread_create(&thread_id[i], NULL, redBlueComputation, (void *) &t_arg[i]);
}
```

Fig.3 Work distribution

Mutex lock implementation: Containing barrier initialization, barrier function and barrier destroy function.

```
// barrier initialization
void mylib_barrier_init(mylib_barrier_t *b) {
    b->count = 0;
    b->flag = 0;
    pthread_mutex_init(&b->count_lock, NULL);
    pthread_cond_init(&b->ok_to_proceed, NULL);
}

// barrier function
void mylib_barrier(mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&b->count_lock);
    b->count += 1;

    if (b->count == num_threads) {
        b->count = 0;
        pthread_cond_broadcast(&b->ok_to_proceed);
    } else {
        pthread_cond_wait(&b->ok_to_proceed, &b->count_lock);
    }
    pthread_mutex_unlock(&b->count_lock);
}

// barrier destroy
void mylib_barrier_destroy(mylib_barrier_t *b) {
    pthread_mutex_destroy(&b->count_lock);
    pthread_cond_destroy(&b->ok_to_proceed);
}
```

Fig.4 Mutex lock

C.2 Test

I perform serval test cases to test my program.
1.Sequential
Two command to test sequential computation.
(1) ./pthreads 1 4 2 55 10
(2) ./pthreads 1 30 10 60 10
2.Parallel
Three command to test parallel computation.
(1) ./pthreads 5 30 10 60 10
(2) ./pthreads 7 30 10 60 10
(3) ./pthreads 10 50 10 60 10

I just use ./pthreads 1 4 2 55 10 to test. I use the initial grid, and the grid printed each iteration to check one by one to make sure red and blue are moving as required. After checking one by one, I can make sure that the red and blue move as required, so the computation has no problem.

Then I run the ./pthreads 5 30 10 60 10, in parallel computation, there is a self-check after the parallel computation, as the sequential computation has been proved is correct, the result of parallel computation is the same as the sequential computation, I can make sure the parallel computation is correct.

D. Possible improvement

I think my program meets the requirement of all listed points in the Assignment 2, there is no major improvement in my program.

E. Manual

The example steps to run the program.
1. unzip the a2_ls.zip file
2. cd a2_ls (Make sure change to the directory contains file A2_Pthreads.c)
3. gcc -o pthreads A2_Pthreads.c
4. ./pthreads 7 30 10 60 10
5. Then can see the output of the program.

Input:
The 5 inputs we need to input are the n_threads which represents the number of the threads, the n which represents n*n grid, t which represents t*t tile, c which represent threshold, and maximum iterations.

Output:
When the number of threads is one, we will run sequential computation, we can see the results of each iteration printed. If there are any tiles exceed the threshold, we can also see which tiles exceed and the detailed information in those tiles. If the number of threads is larger than one, the output will have the results of parallel computation, and followed by a self-check sequential computation.