CM30225 Parallel Computing
Shared Memory

Rowan Walshe

November 19, 2017

Chapter 1

Code

1.1 Introduction

While working on this coursework, I attempted a number of different solutions for parallelising the problem. In this section I will discuss the different approaches to parallelism for each version, along with the advantages and disadvantages.

1.2 Version 1

In the sequential program of my attempt, the program moves through the 2D array from the top left down to the bottom right. At each point it temporally stores the value at the point, then calculates the new value based on the average of the surrounding four values. It then checks to see if the absolute difference between the new and old value is greater than the given precision. If the value at any point changes more than the given precision, then at the end of this iteration of averaging values, it must complete at least one more iteration. This is illustrated in the code bellow.

```
while (1) {
    endFlag = true;
    iterations++;
    for(i=1; i<sizeOfPlane-1; i++) {
        for(j=1; j<sizeOfPlane-1; j++) {
            pVal = plane[i][j];
            plane[i][j] = (plane[i-1][j] + plane[i+1][j]
                + plane[i][j-1] + plane[i][j+1])/4;
            if(endFlag && tolerance < fabs(plane[i][j]-pVal)) {
                endFlag = false;
            }
        }
    }
    if(endFlag) {
        return iterations;
    }
}
```

To parallelise this, each thread is given a certain number of rows that it has calculate. The threads work out which rows of the 2D array they have to work on based on the size of the 2D array, the total number of threads and their thread ID. For example if the 2D array is of size 10 by 10 and there are four threads total, each thread will work on exactly two rows. If the number of rows does not divide

evenly between the number of threads, and there are n rows remaining, then one extra row is given to the thread with ID's less than the number of remaining rows. This means that at most, each thread has to calculate the averages for one extra row. The logic for this can be found in the code bellow.

```
unsigned int sizeOfInner = threadData->sizeOfPlane-2;
unsigned int rowsPerThreadS = sizeOfInner/threadData->threadCount
    +1;
unsigned int rowsPerThreadE = sizeOfInner/threadData->threadCount;
unsigned int remainingRows = sizeOfInner - threadData->threadCount
    *rowsPerThreadE;

unsigned int startingRow, endingRow;
if(threadData->id < remainingRows) {
     startingRow = threadData->id * rowsPerThreadS + 1;
     endingRow = startingRow + rowsPerThreadS;
} else {
     startingRow = threadData->id * rowsPerThreadE + remainingRows
         + 1;
     endingRow = startingRow + rowsPerThreadE;
}
```

After a thread has done their portion of the work, they barrier once to wait for all of the threads to finish doing their work on the 2D array. Then each thread checks to see if they need to do another iteration or not. Once they have checked barrier again. The main thread then resets the finished flag, followed by one more barrier. If the threads did not barrier to wait for all the threads to finish their calculations, you end up with a race condition, and some threads may attempt to prematurely return, which would cause the program to freeze. If the threads did not barrier before reseting the finished flag, the flag may be reset before all of the threads have checked whether or not they have finished or not which would be a race condition, and would lead to a similar situation as before. Finally, they barrier once after the flag has been reset, so that the threads do not start work early, which would be a race condition, that may lead to incorrect results. The code for the main loop of the main thread can be found bellow. The only difference from the main loop of the child threads is that it counts the number of iterations, and it is the only thread that attempts to reset the finished flag.

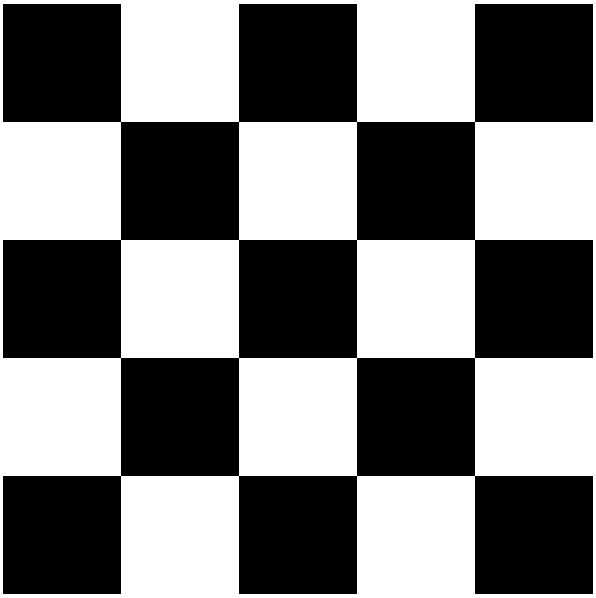
```
// Main Thread
while (1) {
     iterations++;

     relaxPlaneRows(threadData->plane, threadData->sizeOfPlane,
         threadData->tolerance, startingRow, endingRow);

     pthread_barrier_wait(&barrierGeneric);
     if(finishedFlag)
          return iterations;
     pthread_barrier_wait(&barrierGeneric);
     finishedFlag = true;
     pthread_barrier_wait(&barrierGeneric);

}
```

1.3 Version 2



1.4 Version 3

Chapter 2

Correctness Testing

Chapter 3

Scalability Testing