CS402 Coursework 1

George Field, u2013389

1 Introduction

In this report I will use OpenMP to introduce multithreading into DEQN to attempt to improve the program speed. The success of the changes will be measured solely on the run time of the program although memory usage comprimises will be discussed.

2 OpenMP Overhead Analysis

Before any changes were made to the DEQN codebase, I first wanted to gather data on how much overhead is needed when creating a thread in OpenMP. Below is the initial test code I used and a sample output.

```
for (int i = 0; i < 5; i++){
    double parallelStart = omp_get_wtime();
    int sum = 0;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp atomic
        sum += omp_get_thread_num();
    }
    double parallelEnd = omp_get_wtime();

    std::cout << parallelEnd - parallelStart << std::endl;
}
```

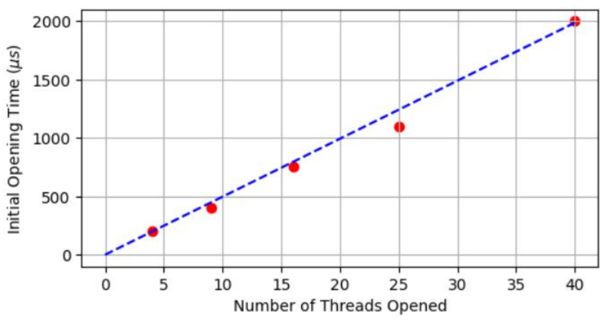
```
0.000168358
4.01901e-06
3.52901e-06
3.847e-06
3.40102e-06
```

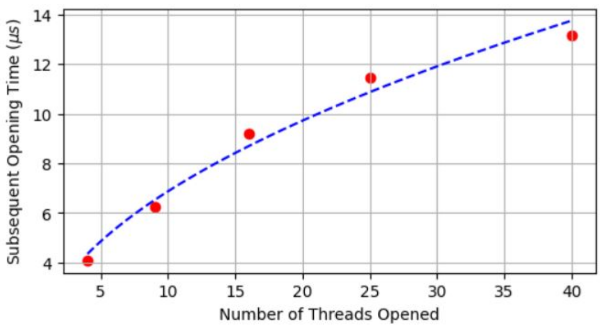
The sum variable was needed so that the compiler didn't optimise away the parallel section. I assumed the single addition operation had a negligible effect on the test outcome.

Strangely, the initial time to open 4 threads was about 50 times longer than with subsequent iterations of the loop. To investigate further, I collected the data for different amounts of threads opened on Warwick's Kudu supercomputer. The subsequent opening times were averaged over 1000 loop iterations. Collecting the initial opening times data was difficult as I had to rerun the executable on each time so they are approximated from inspection of a few runs.

Threads	Initial opening times	Average of reopening times
4	$\sim 200\mu s$	$4.072\mu s$
9	$\sim 400\mu s$	$6.250\mu s$
16	$\sim 750\mu s$	$9.189\mu s$
25	$\sim 1100\mu s$	$11.46\mu s$
40 (max)	$\sim 2000\mu s$	$13.18\mu s$

When plotted, the initial opening times (left) show a strong linear relationship between threads opened with $R = 0.996$. Fitting a line $y = a \cdot x$ using a least squares regression I get $a = 49.6$ indicating that for every thread it takes $\sim 50\mu s$ to initially open it.





From inspection, I predicted the subsequent opening times (right) would fit closely to a $y = a \cdot x^{0.5}$ regression. This proved successful as $R = 0.981$ with $a = 2.17$.

The creation of the above models for initial thread opening time and thread reopening time allowed for a guided approach to how I would parallelize DEQN.

3 Original Benchmarking

I measured timings for 8 aspects of the DEQN code. This test along with all subsequent tests in this report were performed on Kudu. The input file was a modified *square.in* with an *inital_dt* = 0.04, and an *end_time* = 8.00 implying there would be 200 total 'frames'. All further benchmarking times will use this input file. The results are tabulated below.

Diffusion	Bound. update	Cycle	Temp. calc	driver.run()	Write	Init	Total
$23.14\mu s$	$1.014\mu s$	$32.12\mu s$	$21.65\mu s$	$3.054s$	$3.036s$	$0.044s$	3.099

The first 4 columns measure timings from parts of the mesh update code. These are reported as the average over each of the 200 frames calculated. *Diffusion* measures the time taken to update each pixel in the mesh to the next frame using the diffusion equation, *Boundary update* measures the time taken to reflect the boundaries of the mesh, and *Cycle* measures the total time to update the mesh ready for the next frame (the *.doCycle()* function).

driver.run() is the function that both calculates all the mesh element temperatures and writes them to the *.vtk* and *.visit* files for visualisation. *Write* is the isolated time taken to write all the visualisation files. *Init* measures the Driver constructor function that allocates the memory for all the meshes, gathers data from the input file, and writes the initial output file.

4 Implementing OpenMP

Through benchmarking, it became evident that to optimise the program's run time required a significant focus on parallelising the file-writing process since it currently accounted for 97.9% of the total program time. However, this meant that the current method of swapping data between two blocks of memory would have to be replaced by a system storing the mesh at each time step so that they could all be written to *.vtk* files synchronously.

Even if I neglected the write time of the program and chose to only parallelize the diffusion algorithm, I found that the OpenMP threads would take the large initial opening time if the program wrote to a file in between cycle calls.

Therefore, I chose to restructure the program so the meshes are calculated consecutively with each stored in memory; only afterwards are the meshes written to the *.vtk* output files. This meant a huge increase in memory usage but I couldn't see another solution.

4.1 Diffusion Algorithm

Reviewing the OpenMP overhead analysis it was clear employing a parallel approach to the diffusion and temperature calculation would be worth it if threads took the reopening time to start. I chose a spatial partitioning approach where the mesh is split into cells that perform their diffusion calculations simultaneously and then rejoin the main thread to share boundary data.

Each cell was given a 1 element wide halo. Outward facing cell sides are updated via reflection and inward facing sides are sent the neighbouring cell's relevant information. Meshes are stored in memory in a 2D array where index 0 refers to the cell number and index 1 refers to the element in that cell. This allowed for easy multithreading:

```
#pragma omp parallel num_threads(NUM_CELLS)
{
    int cellNum = omp_get_thread_num();
    double* u0cell = u0[cellNum];
    //... [diffusion code] ...
}
```

Part of DEQN is the temperature calculation. This was multithreaded in the same way. Each cell's temperature was calculated in parallel and stored in a local temperature variable that was then added to the global temperature variable within the thread using OpenMP's atomic flag.

4.2 Writing the Output Files

Dividing the write time by 201 from the benchmark (200 *.vtk* files and 1 *.visit* file), it took an average of 15.10ms to write one file. This meant that, even if opening a thread took the initial opening time (50μs), it would still be better than a single thread write multiple files. Therefore I decided to use the maximum number of threads available (40 on Kudu) to minimise the amount of files each thread had to write.

5 Multithreaded Benchmarking

5.1 Diffusion algorithm

Given a number of divisions, the mesh is split into a *divisions* $\times$ *divisions* matrix of cells with the number of individual cells equal to $divisions^2$. Below are the timing results for the relevant aspects of the program given an amount of divisions.

Divisions	Diffusion	Bound. update	Cycle	Temp. calc.
2	28.99μs	4.187μs	33.21μs	5.976μs
3	21.68μs	8.954μs	30.71μs	7.411μs
4	15.05μs	11.56μs	26.67μs	8.524μs
5	12.59μs	16.48μs	29.14μs	10.50μs

The ultimate measure of diffusion performance is the cycle time. In this case 4 divisions performed the best averaging 26.67μs per frame. This is an improvement over the 32.12μs original cycle time but not a large improvement. This is because reopening the 16 threads necessary costs 9.189μs as well as the boundary update function now having to share data between neighbouring cells as well as reflect against the mesh boundaries.

In fact, the cycle time is only faster due to the original having to copy the stored values from one array back to the other in the *reset()* function. However, even if this copying in the original code was removed (by swapping pointers instead of copying data), it does not mean my cycle code is redundant. For larger meshes the advantages of my parallel code become more pronounced. Below are results for increasing mesh sizes.

Mesh Size	Original Cycle Time	4 Divisions Cycle Time	Speed Increase Factor
100×100	$32.12\mu s$	$26.67\mu s$	1.204
200×200	$123.2\mu s$	$64.54\mu s$	1.909
300×300	$201.4\mu s$	$100.8\mu s$	1.998

The OpenMP overhead remains constant when compared to the mesh size and the impact of boundary sharing increases linearly with the mesh size. However, the amount of diffusion calculations grows proportionally to the mesh size squared. Since the diffusion calculations are the aspect of the code that is sped up by multithreading, the benefits become more apparent on larger meshes.

5.2 Surrounding Program

Below are the results for 4 other timings that were also recorded from the original code. I chose to compare against my implementation running 4 divisions as that performed best in the diffusion benchmark.

Divisions	driver.run()	Write	Init	Total
4	$0.2528s$	$0.2433s$	$0.025494s$	$0.2803s$

The use of multithreading to write the *.vtk* files simultaneously significantly improved the write time. The initialisation time remained similar to the original program despite the amount of memory allocated for the meshes being much greater and the opening of the initial threads being included in the *Init* time. This is due to the majority of the initialisation time comes from opening and writing the initial mesh to a file, the cost of which far outweighs anything else.

6 Possible Improvements

My solution incurs a significant memory cost. I attempted to mitigate this by computing meshes and writing *.vtk* files in batches such that when the writing function is called there is one thread allocated per file. Theoretically, this approach would limit memory usage, with a worst-case scenario on Kudu of the 40 threads taking $2000\mu s$ to open for file writing and less than $2000\mu s$ to open for another batch of mesh calculation, resulting in a runtime increase of under 7%.

Unfortunately, this strategy unexpectedly led to a 50% increase in runtime causing me to remove this change. I hypothesise that due to possible branching into the file writing function in the diffusion calculation loop, the compiler could not use the quicker reopening time when opening threads in the diffusion code. This could potentially by fixed by rewriting the code so that the compiler knows it will only write to files every 40 iterations but I did not implement this.

7 Conclusion

Overall, the program (using 4 divisions) ran 11.06 times faster. The vast majority of the speed increase was because of the parallelisation of the *.vtk* file writing but, through altering the structure of my program to use an optimised reopening time for threads, I also managed to achieve speed up in the diffusion algorithm. My program uses a factor of $\frac{end\_time}{2\times initial\_dt}$ more memory than the original code but neither the diffusion algorithm nor the file writing system could have been improved so significantly without this change to storing all the meshes at once.

I believe my implementation of OpenMP was successful and highly efficient not only compared to the original, but compared to any naive approach where *omp parallel* flags are haphazardly used on any large loop.