CS402 Coursework 1

u1805023

1 Introduction

The heat diffusion is simulated by running a plus shaped kernel across the matrix which simulates the loss and gain of heat energy from each of the neighbouring four tiles with the change in temperature dependent on the size of the time step. The new temperature is stored in a new array as to not affect later computations. Array indexes are calculated by using a small calculation inserted by the C preprocessor.
After storing the diffused results in the second matrix, the matrices are reset by copying the updated data from the second matrix back into the original matrix. Only the actual data of the matrix needs to be copied as the boundary values will be overwritten in the next step.
In the final step the boundary data is updated. To handle the 3x3 kernel being applied to cells on the edge the matrix contains boundary values which need to be updated to ensure no heat energy is lost via the borders. This is done by copying the values on the edge of the valid data matrix to the boundary for each of the four edges of the 2D matrix.
To determine which changes result in a performance gain, timing measurements were made to identify how long each of the tasks took in each loop. Time differences were measured in microseconds using OpenMP's *omp_get_wtime*. The timings were written out to standard out where they can be read normally or read in by another program to process. In this case *deqn* is called by a Python program which stores the timing information to generate graphs.

2 Optimisations

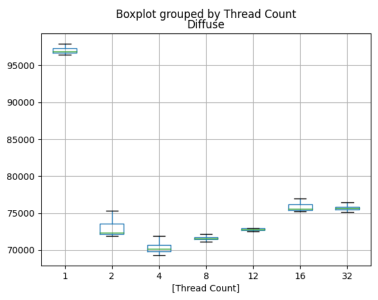
Optimisations were applied to loops measured as being expensive and also slow functions as detected with the valgrind profiler and kcachegrind. Mesh initialisation was accelerated using a parallel for loop and mesh *getTemperature* was parallelised with a for loop reduction. As the mesh initialisation took up less than 3% of the runtime for a 20 step simulation its graphs have not been included. As *getTemperature* can be considered debug output it has not been included in the timing measurements which focus on the diffusion *doAdvance* step, however it is very similar to the diffusion and reset loops with a reduction rather than writing to an array.

The following optimisations assume that the number of rows in the mesh is larger than the number of threads as tasks are allocated by row. However as will be shown later no noticeable performance loss will occur as long as two threads have work.

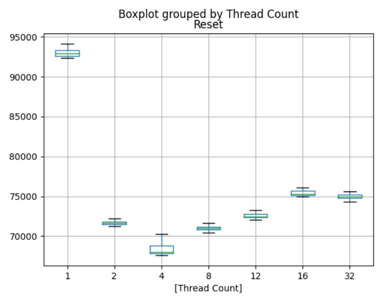
2.1 Diffuse

Diffusion is accelerated by parallelising the outer loop which iterates over the rows. Parallelising the inner loop would not result in noticeable performance gains as the number of threads almost certainly is smaller than the number of rows and iterating over a row enables better cache locality and easier to predict memory accesses than splitting the same row across multiple processor cores and having each core iterate over more rows.

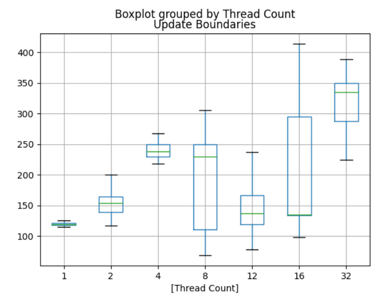
Tiling the matrix was also tested as an optimisation with the matrix being broken into a number of smaller square tiles. The idea behind this is that the diffusion equation requires the rows above and below to also be loaded from main memory along side the row currently being updated. In the normal update mechanism the entire row is updated before moving onto the next, meaning that for long rows the next row may no longer be in the cache once it is time to be updated. By splitting the rows into smaller sections the tiling method effectively reduces the row length, theoretically allowing the next row to remain in the cache, avoiding a costly trip to main memory. Additionally if the tiles fit within the lower cache levels not shared across processor cores, adding more threads would allow this otherwise unused cache to be utilised.



(a) Diffuse



(b) Reset



(c) Update boundaries

Figure 1: Individual function timings against 10000x10000 grid (microseconds)

2.2 Update boundaries

To parallelise the boundary update all four edges are copied in parallel. Only one thread is used for copy each row as the length of each edge will be small compared to the overhead of starting OpenMP processes and the size of the matrix itself. The processes are started as OpenMP tasks within an OpenMP parallel scope such that each boundary starts at most one thread.

2.3 Reset

The *reset* function has a very effective and simple optimisation, eliminating the copy entirely. As the original *u0* matrix data is not used after the reset the *u0* and *u1* matrix pointers can be swapped, resulting in the same effect. However as this is not a clear OpenMP optimisation it has not been used in this coursework.

To improve the copy performance the standard OpenMP pragma is used to parallelise the outer loop of the copy i.e. each row is copied by only a single thread. Additionally *std::memcpy* is used to perform the copy which doesn't provide a noticeable increase in performance but simplifies the code and may provide an improvement if the implementation were modified to use a faster copying mechanism. All variables inside the loop were initialised locally to any threads using *firstprivate* apart from u0 and u1 as the matrices need to remain valid once to loops exit.

Performing a parallel copy using a single combined loop across the array resulted in a performance regression relative to the outer loop and memcpy implementation.

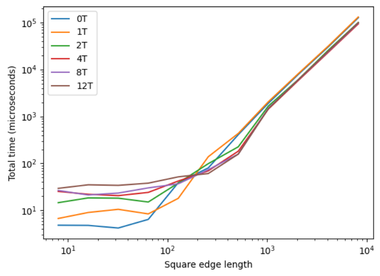
3 Results

The timings of the individual OpenMP parallelised functions for varying numbers of threads can be seen in Figure 1. Increasing the thread count only reduces the time for a single diffusion iteration up to a point and the addition of a thread only reduces the time required by approximately 25%. For this system scaling beyond 4 threads results in a slight increase in the time required. Scaling the number of threads beyond the number of processors available also results in a performance decrease but the loss is less than 5%.

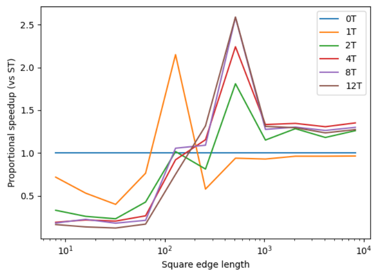
The situation is worse for *Update Boundaries* where the time required can increase up to 2-3x as the thread count increases. This is likely due to the lower cost of this function relative to the other two at $100\mu s$ vs $70000\mu s$.

The overhead caused by OpenMP is relatively low overall, taking approximately 3% longer with a single thread compared to the original implementation. For the smaller *Update Boundaries* function the overhead is similar for a single thread, which is expected as with a single thread OpenMP doesn't necessarily need to spawn new threads and can run the computations on the main thread. However with more threads the overhead increases and the time taken and the variance actually increases. This doesn't effect the overall time taken due to the low cost of the function and only 4 calls made.

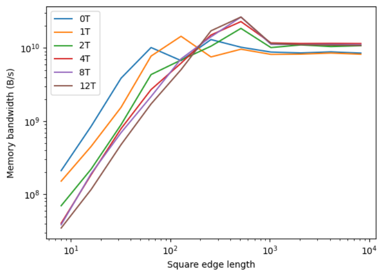
Figure 2 shows the effect of varying the thread size for various thread counts. As can be seen from Figure 2a for small square sizes the performance of OpenMP is significantly behind that of the original implementation, and the difference becomes greater as more threads are added. However once the square edge length increases beyond approximately 128 the time taken by the multithreaded implementation grows slower than both the



(a) Time taken

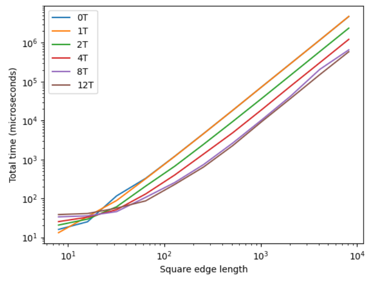


(b) Speed up relative to no OMP

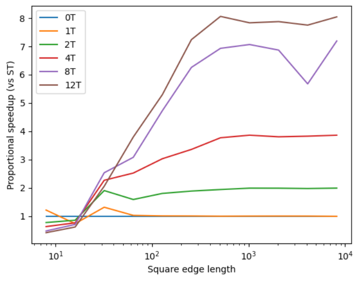


(c) Minimum memory bandwidth

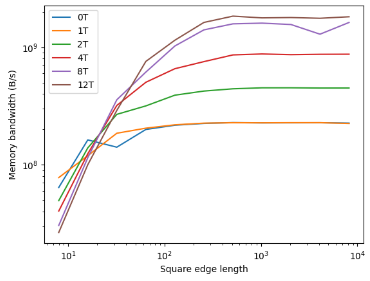
Figure 2: Effects of increasing square size for various thread counts. 0T = no OMP



(a) Time taken



(b) Speed up relative to no OMP



(c) Minimum memory bandwidth

Figure 3: Effects of increasing square size for various thread counts with expensive trig functions. 0T = no OMP

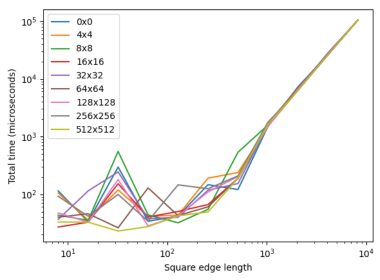
original implementation and the OpenMP version with a single thread. The gap increases up to around an edge size of 512 after which it shrinks, however the multithreaded version remains slightly faster.

Figure 2b shows the proportional speed up relative to the original implementation which provides a clearer look at the performance differences OpenMP causes. The cost of initialising many processes is clearly visible for small square sizes while the speed up for mid sized squares is greater than for larger squares. To explain the spike and the convergence for large square sizes, Figure 2c is required, showing the minimum memory bandwidth required to process the array (number of cells with 8 bytes each, read from *u0*, write to *u1*). Using this graph it is possible to see that the greatest performance gains occur while the problem size is large enough to take advantage of multiple cores but small enough to still fit in the cache. However once the problem size requires trips to main memory the program becomes memory bandwidth limited and the multithreaded implementations converge in performance, slightly ahead of the original implementation and OpenMP with one thread which are not quite memory bandwidth bound. The move from cache to main memory is visible in the drop in minimum bandwidth past a square size of 1000. Further support is provided by Figures 1a and 1b for the timings of diffuse and reset for large squares. Note that the difference in time required is small despite diffuse having to perform multiple calculations while reset performs a copy with few unpredictable branches. This indicates the performance is mostly limited by the ability to copy the array back and forth rather than the computation itself.

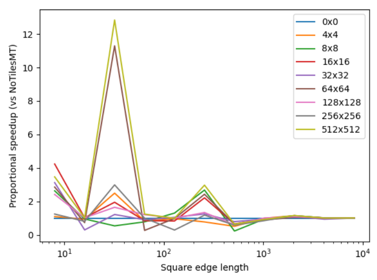
Tests with a more complex kernel also support this theory. Wrapping the diffusion calculation with multiple expensive trigonometric functions $cos^{-1}(cos(sin^{-1}(sin(x))))$ which have latencies of 40-100 cycles compared to 3-5 cycles for the floating point addition and multiplication used normally in the diffusion kernel. [1] This results in much better scaling at around 7 times on 8 threads, indicating good scaling can be achieved if other bottlenecks are not encountered first.

3.1 Tiling

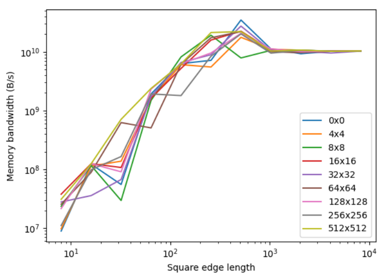
As memory bandwidth was identified to be the limiting factor, tiling was tested to determine if better cache utilisation could improve performance relative to the normal OpenMP implementation. As can be seen from



(a) Time taken

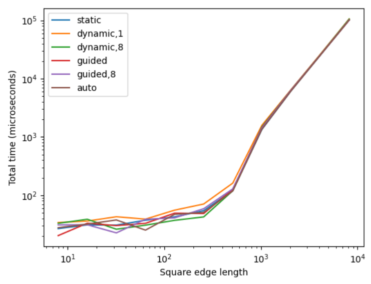


(b) Speed up vs normal OMP

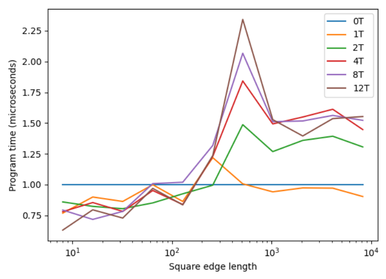


(c) Minimum memory bandwidth

Figure 4: Effects of increasing square size for various tile sizes. 0T = no OMP



(a) Effect of runtime schedule settings



(b) Overall program speed up

Figure 5: Schedule effects and overall program speed up

Figure 4 there are no large gains overall. There are some spikes around square sizes of 64 and 128 in Figure 4b but these appear to be caused by random noise when looking at Figure 4a rather than a consistent advantage and the tiles sizes which perform better vary from run to run. Overall tiling does not seem to provide a clear advantage over the OpenMP implementation and comes at a cost of increased code complexity.

3.2 Scheduling

The effect of scheduling was relatively minimal, as can be seen in Figure 5a. This is likely due to each of the work units consisting of an entire row and each unit of work taking approximately the same amount of time. This results in a low overhead for dynamic scheduling as each work unit is relatively large and minimal disadvantages for static scheduling as each thread will finish its set of work at approximately the same time.

3.3 Overall performance

Overall performance scales similarly to the main loop of the program as can be seen in Figure 5b. This is as expected as the diffusion step is the most costly section of the program followed by the total temperature calculation. However if visualisation were enabled the scaling would be reduced as the proportion of the program which is multithreaded would then decrease.

References

[1] Agner Fog. *Instruction tables.* URL: https://www.agner.org/optimize/instruction_tables.pdf. (accessed: 02.02.2022).