CS402 High Performance Computing
Assignment 1: OpenMP with deqn code

Michael Boyle
1403116

February 2018

Introduction

The *deqn* code is designed to simulate the diffusion of heat through a two dimensional material. The explicit scheme which performs this looks at the discrete areas surrounding an individual 'cell' and factors in each of these as an element in the next iteration of the materials heat distribution. At the end of each diffusion iteration, the 'stepforward' array (u1) is moved back into the base array (u0) in order for another time unit to be stepped forward.

The task for this assessment is to take this sequential code and find ways to improve the running time making use of OpenMP directives. For the most part, this takes the form of parallelizing the loops within the code. Comparison between runtime at various thread counts will be drawn, with analysis on the expected and actual results.

Code Breakdown

deqn takes in an input file specifying a region of an abstract material, with an optional subregion included that is heated. It also specifies how to discretized the area, which is essential for the machine runnable program that is produced. The implemented explicit schemes makes sure that there is no heat loss across the entire running of the code, as the boundary reflects any potential loss of heat, as can be seen in Figure 1. The heated region may diffuse across the remaining space, but the sum of the discrete space's temperature will remain the same across each executed cycle of the code.

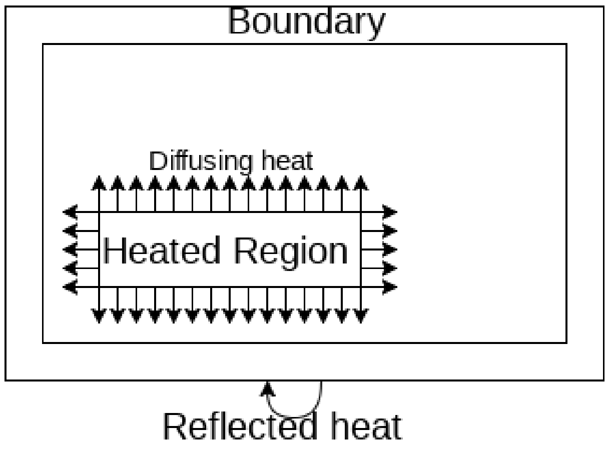

Figure 1: Special processing occurs at the boundary which results in heat being reflected back in to the area

Initial valuations of running times for various processes showed that the majority of the time was spent on writes within the vtk writer. The vtk writer was one of the few classes that could not be parallelized due to the writing out of the files needing to be sequential. It was also not possible to write a file while performing the next diffusion cycle, since the writer class pulled values from the mesh, which has a dependency within the Explicit Scheme, as it is changed upon reset. This meant for the most part that test data was run with a $vis\_frequency$ value of -1, so no writes were performed and the diffusion code could be tested faster.

For some factorisations of the space (ie selected values of nx, ny) it was possible to construct a diffusion which diverges from the starting temperature, often to values magnitudes different from the original. *large.in* is an example of this, which reaches a value of $1.02765e^{14}$ by the end of 20 steps. It should be noted that it is the number of cycles allowing for the small errors to build up exponentially, rather than the value being accurate at a time value of 0.8, since the dt value can be changed to 0.08 (meaning there is 10 incremental steps), resulting in a smaller diversion of 889059 from the original value of

900000. It may also be the case that these values are different depending on which machine the program is run on, as floating point errors can be architecture dependent. To avoid these floating point errors influencing the running time of the code in any way, specific testing values which avoid these errors have been chosen, such as giant.in and massive.in.

Optimisations

Outside of the writer, there are a few loops scattered across the program which are potentially optimizable.

```c
#pragma omp parallel for schedule(static)
for (int j = 0; j < y_max+2; j++) {
  for (int i = 0; i < x_max+2; i++) {
    if (celly[j] > subregion[1] && celly[j] <= subregion[3] &&
      cellx[i] > subregion[0] && cellx[i] <= subregion[2]) {
      u0[i+j*nx] = 10.0;
    } else {
      u0[i+j*nx] = 0.0;
    }
  }
}
```

Listing 1: Diffusion initialization with subregion

Listing 1 demonstrates how static scheduling is used for initializing the cells in the defined region [1]. It should not be necessary to make use of dynamic scheduling in this program as there will not be any loops which require additional execution time, except from the potential small differences in fetching different memory locations.

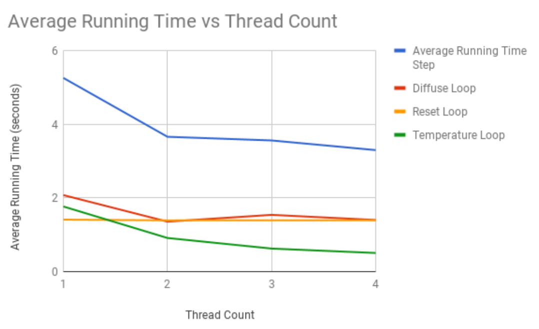

Figure 2: Average Running Time for a single diffusion cycle over the mesh for specific loops with parallelization. Considers for 1-4 threads, due to the test machine having 4 cores.

Interestingly, despite this apparent instruction parallelism, OpenMP's Single Instruction Multiple Data directive does not improve the running time in any of the loops within this program. During testing this resulted in either no change or even slowdown in some cases. This may be due to accessing of data slowing down a grouping of instructions being performed in parallel, when threads may have otherwise completed an iteration and moved onto the next index.

collapse(n) is another keyword that can be used with the parallel directive to allow parallelization across nested loops, however in tests it also failed to improve the running time of the program. This may just be the implementation of OpenMP that is being used not making use of the directive optimally, however it is more likely that the compiler automatically collapses the loops upon seeing the OpenMP 'for' directive.

The most complicated loops to optimize are the temperature, initialization, reset and diffuse loops which are all nested for loops. The temperature loop calculates the total temperature across the whole mesh, by summing each of the values at every position in the array (outside of the boundary values, which act as reflectors). This allows for a reduction [2] to be used across the team of threads, which is one of the best speedups across each of these optimizations since it makes use of the thread pool most effectively, as can be seen from Figure 2.

The reset loop is a more basic directive usage as all it does is move the newly calculated values into the base mesh. If the diffusion cycles were implemented in a parallel way, this is the process which would need to be blocked until the writer released a lock, allowing new data to be written over. Only after the

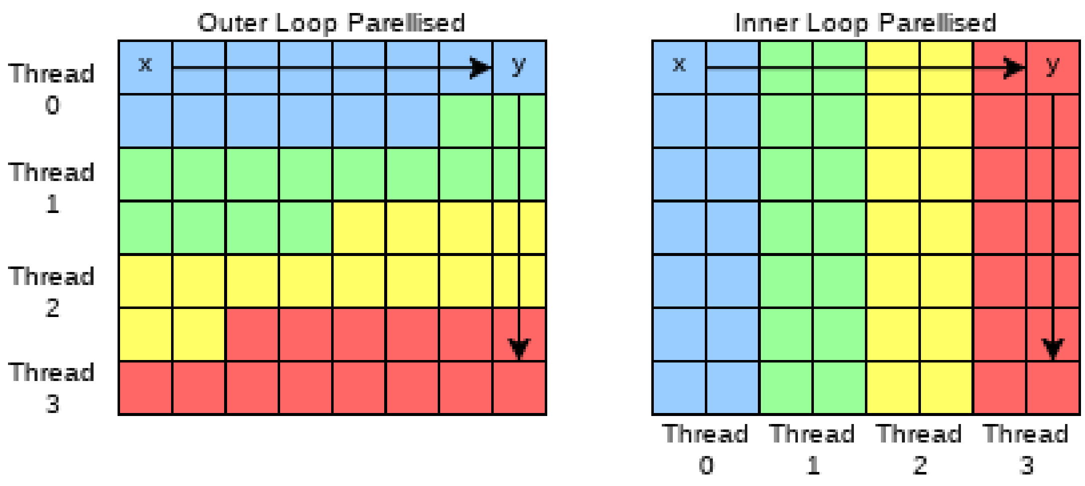

Figure 3: Temporal locality can be exploited by reusing values when calculating on mesh cells which are next to a cell which has been previously used.

reset method is called can a new write be performed, however any write would mean that the next reset must be held until the write is complete.

The Diffuse Loop

Temporal and Spatial Locality comes heavily into play within the diffuse loop and it is possible to gain a slight increase in computation through exploiting what is retained in cache for each thread [3]. From Figure 3, it is shown how the instructions are broken up when the directive is specified outside of the outer loop. While this effectively captures many values, it is possible to improve cache accesses minutely by parallelizing the inner loop instead. The second part of Figure 3 shows this as the diffuse operation looks at the cells to the right, left, above and below the cell being calculated. By parallelizing so that each value of y (ie any one column) is kept to a single thread, temporal locality can be exploited along the vertical as well as the horizontal directions of the mesh. This becomes more effective the larger the input, as the temporal locality (accessing the same data again) can be repeated more often, however there is likely a threshold where this data is no longer kept in cache when progressing to the next 'row' of the mesh, and thus will need to request the above 'values' again.

This is only a minimal improvement, rarely producing more than 5% speedup for the diffuse loop alone since there is also plenty of spatial locality to exploited through parallelizing the outer loop as moving along entire rows on one thread will allow for values to be reused along each horizontal.

Testing

As the development of this project was iterative, version control was useful to continually test the effectiveness of various inputs, even when a new input was created late into development [4]. Initial development failed to show any real improvement through use of the OpenMP directives, but after starting to work with larger input files, the speedups became more obvious. In all of the tests performed, however, there was never a speedup which matched the expected running time given the number of threads that were being used.

As Figure 4 shows, the speedup from 1 thread to 2 is proportionality much larger than for 3 or 4 threads. Considering the expected speedup would optimally be $\frac{1}{n}$ for n threads, not even halving the running time after quadrupling the thread count is not particularly impressive. With the relatively small difference in actual time between running time for the different thread counts (between 2-4), this suggests that the overhead for running OpenMP is actually a fixed amount of time (relative to the input), irrespective of the number of threads. This could well correspond to the initialisation of the thread pool [5], which is clear to see from Figure 5 between the first iteration of diffusion versus every other iteration once the program is running, which on average takes about 50% longer.

Thread Count	Step	Diffuse Loop	Reset Loop	Temperature Loop	Speedup
1	5.25567	2.07619	1.41138	1.7672	N/A
2	3.65877	1.35577	1.39024	0.912249	30.38%
3	3.55921	1.54009	1.39338	0.62514	32.28%
4	3.29778	1.40028	1.3911	0.505844	37.25%

Figure 4: Breakdown of Average Running Time for massive.in across the main loops which run.

File	First Diffuse	Second Diffuse	Ratio
massive.in	1.94719	1.36902	1.422323998
large.in	0.00536724	0.0032367	1.658244508
giant.in	0.128932	0.0887196	1.45325272
Increments.in	0.833293	0.528537	1.576602963
Average	N/A	N/A	1.527606047

Figure 5: Ratio of First Diffuse loop to second diffuse loop

Since this overhead tends to occur only on the first running of diffuse, the overhead will tend to be negated by any task which runs for a very large number of cycles.

Conclusion

The OpenMP directives used in these optimizations are statically scheduled due to the consistent computational cost across each of the loops. Through extensive tests, this implementation has demonstrated how useful parallelization can be, but has also shown that the overheads can be a major factor if the input is too small in terms of computation, or lacks the cycle count for the initial startup cost of threads to be countered by the improved running time within the loops.

Vectorisation could be incorporated to allow for Single Instruction, Multiple Data principles to be applied to this code, which would more effectively make use of the parallelism explored in this project. This could, however, come with the possibility of losing accuracy to floating point errors, which has already been shown to be a problem for some inputs. Additionally, this would usually be machine dependent, so would not be as portable as the OpenMP directives.

Further improvements could be developed around allowing the writer class to handle files externally from the program, possibly through forking. This would allow the mesh data to be copied, which would remove the problem created by the dependency between diffuse, reset and write. While this might be slower for the entire program, it would allows the program to continue on to further cycles without waiting for writes to complete and this could also free the user up to be doing other things while files are being written, since file writing is bound by the disk writing speed rather than CPU time.

References

[1] Intel. Openmp loop scheduling. https://software.intel.com/en-us/articles/openmp-loop-scheduling.

[2] Openmp reductions. http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-reduction.html.

[3] William Stallings. *Computer Organization and Architecture: Designing for Performance.* Pearson Education Limited, 10 edition, 2015.

[4] Assignment version control. https://github.com/Wellwick/cs402.

[5] Openmp reference sheet. http://www.plutospin.com/files/OpenMP_reference.pdf.