Games Engineering

Wator Project



A – Team

Patrick Doyle

James Browne

Dean Ryan

Martin McGuiness

# Work Log

# James Browne

* SDL setup and integration
* OpenGl setup and integration
* Co - wrote and optimised final version of wator
* Testing of Serial Code
* Testing of Optimised Code

# Patrick Doyle

**Serial implementation**

* Created Wator framework
* Initialization of sharks and fish
* Pair Programming (w/ James Browne)
  + shark movement
  + shark feeding, starving
  + Breeding function for sharks and fish

**Benchmarking**

* Created FPS calculator
* Ran tests for serial code
* Grid size VS FPS

**Optimization**

* Pair Programming (w/ James Browne)
  + Refactored serial code
  + Optimizing of serial code
* OpenMp
  + Implemented OpenMp into serial code
  + Implemented OpenMp into optimized code
* Benchmarking Optimized Code
  + Ran tests for Optimized code
    - Grid size VS FPS

# Martin McGuiness

* Optimised Outputted the FPS to a file.
* Added Functionality to run project for a fixed number of cycles
* Ran through the code to make sure none of the typical errors were present (e.g. making the inner for loops parallel, making sure there are no breaks or returns end a parallel for loop).
* Dean and I tried to use barriers to make the program faster, but this had no or very little effect.
* Dean and I tried to maximise the parallel regions by adding #pragma omp parallel areas with for loops that were parallel also. This was to optimise the opportunities for using the data in the cache.
* Co - wrote Fish and related fish functions
* Made Graphs
* Tried optimising Wator.c (first attempt)
* Testing of Serial Code
* Testing of Optimized code

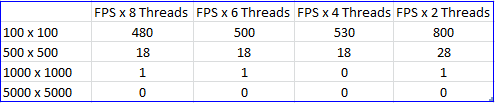
# Dean Ryan

* Optimised Outputted the FPS to a file.
* Added Functionality to run project for a fixed number of cycles
* Ran through the code to make sure none of the typical errors were present (e.g. making the inner for loops parallel, making sure there are no breaks or returns end a parallel for loop).
* Martin and I tried to use barriers to make the program faster, but this had no or very little effect.
* Martin and I tried to maximise the parallel regions by adding #pragma omp parallel areas with for loops that were parallel also. This was to optimise the opportunities for using the data in the cache.
* Testing of Serial Code
* Testing of Optimized code

## Results

**SERIAL IMPLEMENTATION**

Below are the results for the testing of our serial code implementation. The graph displays the resulting FPS (frames per second) against various grid sizes for the Wator world.

****

**Serial Code**

**Grid Size**

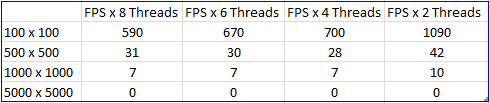
**F**

**P**

**S**

# SERIAL IMPLEMENTATION WITH OPENMP

The next set of results show the performance of the serial implementation with OpenMP added to the project. As you can see from the results there is an increase in frame rate in almost all cases. The most dramatic increase can be seen for grid sizes of 1000x1000, where the frame rate rose from 1 FPS to 7 FPS in one case, and from 1 FPS to 10 FPS in another.



**Serial Code with OpenMP**

**F**

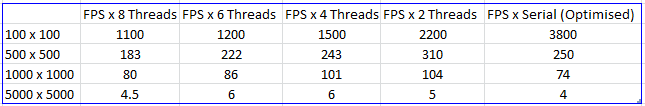
**P**

**S**

**Grid Size**

# OPTIMIZED CODE WITH OPENMP

The final set of results shows the performance of our Wator application after optimization of our serial code as well as OpenMp implemented. The implementation of OpenMP seemed appeared to have greater effect on the optimized code than it did on the serial implementation. Again there are dramatic improvements in frame rate across the board, with some results increasing tenfold. Some of the most noticeable results are those for grid sizes of 5000x5000, which before failed to reach 1 frame per second. The larger the grid sizes were, the greater the rates of improvements were outputted for the OpenMp optimized code.



**Optimized Code with OpenMP**

**F**

**P**

**S**

**Grid Size**

# Conclusions:

After running numerous tests on our application( with and without OpenMP included) and with help from the OpenMP manual, we have come to a number of conclusions:

* **For smaller grid sizes, OpenMp actually decreased the frame rate**.
* After a number of tests and some research we have found that when the work load is quite small, it is inefficient to include OpenMP into the application. The time taken for creating threads is counterproductive and slows down the simulation. So smaller applications run better in a single thread than splitting the small work load between multiple threads.
* **Multiple threads attempting to access the same memory locations dramatically slows down application speeds.** 
  + After Implementing OpenMP into our serial code, we noticed a dramatic decrease in frame rate. I did some research into this and found some information in the OpenMP manual which highlighted the point and explained the reason for it. In one of our update functions, which we had parallelized with OpenMP, there was quite a few calls to a number of different arrays. When multiple threads attempt to retrieve data from the same location, they tend to block each other out. When the number of threads gets larger, the effect this has on the application increases dramatically. This can be seen in the results for our implementation of OpenMP in our serial code above. To avoid this problem, we restructured our code in such a way as there is minimal calls to data structures within parallel regions.
* **Not all loops need to be parallelized.**
  + After reading a number of manuals and running a number of tests, we came to the conclusions that it is better to leave smaller loops to run in a single thread. The overheads involved in setting up threads and splitting the work load takes longer than it would for the loop to just run by itself.
* **Merging multiple loops increases Parallel efficiency**
  + If an application has a number of parallel regions in sequence, it is more efficient to merge these regions ( where possible ) than it is to run one parallel region followed by another. In other words, it is better to run one large parallel region that it is to run a number of smaller ones.
* **Order Nested loops for more efficient data access.**
  + When making calls to a memory location, in an array for example, a thread takes the whole row from memory and then selects the data from a specific position within this row as required. If you structure your nested loops in a parallel region in such a way so as the inner loop is working along the same row in the array, there will be significantly less calls to the array. This will decrease the change that multiple threads attempt to access the same memory address at the same time, which will in turn decreases the chance of thread blocking.