Box-Link: https://kingston.box.com/s/piwl3i3i076el5d25gs99su3dmk3tw1c

PLEASE READ “READ ME” IN RELEASE FOLDER

Coursework Part B

Real Time Programming

“Cellular Automata: Prey/Predator”

Due date: 17/04/2017

**Student:**

Name: Sören Schreiber

Kingston ID: K1549360

Module leader: -Dr. Jean-Christophe Nebel (JCN)

Lecturers: -Dr. Vasilis Argzriou (VA)

- Dr. Jean-Christophe Nebel (JCN)

Marker: - Dr. Jean-Christophe Nebel (JCN)

**Table of Contents**

[Introduction 4](#_Toc480114878)

[Assignment 4](#_Toc480114879)

[PC configuration 4](#_Toc480114880)

[Serial Implementation 5](#_Toc480114881)

[Algorithm 5](#_Toc480114882)

[Boundary implementation 6](#_Toc480114883)

[Source-Code 6](#_Toc480114884)

[Results 7](#_Toc480114885)

[Screenshots 8](#_Toc480114886)

[Performance Analysis 10](#_Toc480114887)

[Conclusions 10](#_Toc480114888)

[Parallelisation Potential 10](#_Toc480114889)

[Sequential Sections 10](#_Toc480114890)

[Parallel Sections 11](#_Toc480114891)

[Load Balancing 11](#_Toc480114892)

[Critical Sections/Synchronisation Points 12](#_Toc480114893)

[Global Operations 13](#_Toc480114894)

[Parallelisation Method: OMP 13](#_Toc480114895)

[Algorithm 13](#_Toc480114896)

[Boundary implementation 14](#_Toc480114897)

[Source-Code 14](#_Toc480114898)

[Results 15](#_Toc480114899)

[Screenshots 15](#_Toc480114900)

[Performance Analysis 17](#_Toc480114901)

[Conclusions 21](#_Toc480114902)

[Parallelisation Method: MPI 22](#_Toc480114903)

[Algorithm 22](#_Toc480114904)

[Boundary implementation 23](#_Toc480114905)

[Source-Code 24](#_Toc480114906)

[Results 25](#_Toc480114907)

[Screenshots 25](#_Toc480114908)

[Performance Analysis 27](#_Toc480114909)

[Conclusions 31](#_Toc480114910)

[Parallelisation Method: Hybrid 32](#_Toc480114911)

[Algorithm 32](#_Toc480114912)

[Boundary implementation 33](#_Toc480114913)

[Source-Code 33](#_Toc480114914)

[Results 34](#_Toc480114915)

[Screenshots 34](#_Toc480114916)

[Performance Analysis 36](#_Toc480114917)

[Conclusions 40](#_Toc480114918)

[Conclusion 40](#_Toc480114919)

[Annex 42](#_Toc480114920)

[Table of Figures 42](#_Toc480114921)

[Table of Tables 42](#_Toc480114922)

[Table of Tables 42](#_Toc480114923)

# Introduction

The second coursework for the module “Real Time Programming” focuses on the parallel programming techniques taught during week 2 of lectures.

The assignment is to create cellular automata, implementing a Prey/Predator simulation with different parallel coding techniques.

## Assignment

The Prey/Predator model is to be implemented using serial, OMP, MPI and hybrid programming techniques. The cellular automata deal with the population of sharks and fish within an infinite Ocean, which will be represented in a finite 2d grid array.

To simulate an infinite ocean size, the program is designed to “wrap the established grid into a torus shape.

Any given cell can either hold a fish, a shark or be empty(ocean).

The given ground rules for the simulation are as follows:

* Breeding:
  + >=4 neighbours are of same species
    - >=3 neighbours are of breeding age
  + <4 neighbours are other species
* Fish:
  + Birth age: 1
  + Breeding age: >=2
  + Life span: 10 Generations
  + Dies if neighbours are >=5 sharks
  + Dies if neighbours are =8 fish
* Sharks:
  + Birth age: 1
  + Breeding age: >=3
  + Life span: 20 Generations
  + Dies if neighbours are >=6 sharks and =0 fish
  + Dies randomly with a change of 1/32

Visual output can be either in ASCII grid or other visual representation.

## PC configuration

PC configuration used for this project:

* Intel Core i5 (2 cores, 4 logical processors)-> up to 2.3/2.4 GHz
* 8gb Ram
* 64-bit Win 10 Home

# Serial Implementation

## Algorithm

For all the implementations, there have been a few changes to the originally imposed rules.

These changes were made to correct for occurring problems during runtime, which caused either the fish or sharks to die out within a couple thousand generations, or created to much empty space which never got filled again.

The changes are:

* Sharks die with a random chance of 1/48
  + This will prevent to many sharks from dying of random causes
* The breeding requirement has been changed to >=3 neighbours and >=3 in breeding age of one species as well as <4 of other species
  + This change primarily prevents sharp and straight edges from forming. If these edges would be allowed to occur with the old rule, segments of space would be created which would never be able to be filled again by any means.

These changes are implemented in every implementation.

The algorithm used for this coursework is a basic linear flow, starting with the creation of relevant variables and allocating memory for the current, new and past girds.

Next is the randomization of the start world from which the calculations begin (Generation 0). This world is set to be filled with roughly 50% fish, 25% sharks and 25% open ocean.

A random number generator is used to achieve randomisation. (seed provided with srand() for reproducible results).

Once all these setup parts have been completed the start time will be saved for calculating run time. (This implementation is only focused on calculating the time needed for the actual work throughout the different generations. This means that the time given at the end does not match the actual runtime of the program. This is done because he setup could be skipped by providing pre-set data into the array via manual input in the source code. This would reduce computational time if so desired, while the calculations of the different generations will always stay the same no matter how the starting point is acquired.

The version provided here should make results more compatible if used outside of this application (all versions, serial and parallel, are in OOP compliant header and cpp files to be reused in different projects if so desired.

For each cell during each generation, the application runs the same sequence of commands:

1. Reset all necessary values to their default
2. Calculate neighbour positions (left, right, up, down)
3. Check and count all neighbours for species and if in breeding age
4. Depending on current call value (empty, shark, fish) decide what the new cell should look like

After all cells have been calculated, the girds will be shifted though, meaning the current grid will be placed in the buffer and the just calculated grid will be placed in the current grid for use during the next generation.

When all generations are calculated, the timer is stopped and the application will display the elapsed time in minutes on the screen.

If the code segments relating to the visual representation are used, the time needed will increase depending on grid size and how often as visual feedback is printed onto the screen.

[NOTE: The Visual output is commented within the source code, the only way to activate visual printouts is by uncommenting desired output areas and recompiling the application]

## Boundary implementation

Dealing with boundaries is a key part of the implementation. Since the assignment specifies that an infinite ocean is to be simulated, a solution must be found to “warp” the given grid into a torus shape therefor allowing for free and unlimited movement across its surface without ever reaching an end.

There are 3 possible ways of dealing with this issue.

1. Ghost-Cells:

With this method, each border of the grid gets an extra row at the edge. These rows represent the opposite edge of the original grid.

Using this method means that each Generation must copy and fill these cells before the next generation can be calculated. If this is not done the computation will create false outputs based on outdated border data.

Depending on grid size and computer power this can slow the overall runtime.

1. Ghost-Grids:

A similar method to “1”, but instead of just copying one row the entire grid is copied into a 3x3 grid (the middle grid being the “active” one.

The rest of the process is identical to “1”.

This is the easiest solution to implement, but it is also the costliest.

1. “Wrapping”:

“Wrapping” is the fastest and most efficient solution for the torus problem.

Here the code will “calculate” the positions of neighbour cells before neighbours are checked. While this requires more lines of code, especially when checking neighbours, it is the fastest since no memory needs to be allocated for rows are grids and no time is wasted copying those cells into their proper positions.

Here the code calculates the values for “left”,” right”,” up” and “down”. Those values will be used to represent all 8 neighbours of the current cell.

This version is used in the implementation of the serial and OMP version, the MPI solutions utilize the “Ghost cell” method. (more in the MPI section).

## Source-Code

The source code for the serial version is written in a separate .h and .cpp file, which are included in the main selection Hub-program.

Inside the serial module is a function which run the entire cellular automata from memory allocation to timing of the runtime.

It consists of 1 function with the following parameters:

* Void runSerialVersion(int GridSizeX, int GridSizeY)

In addition, it utilizes a function from the ConsoleIO.h which allows for visual or ASCII output of the grid in question.

This function either prints an ASCII representation of the grid into the console or paints each individual pixel, starting in the top left, in a specified colour representing the three different value types.

The source code requires multiple definitions for its operation.

* maxLoops -> number specifying how often the cellular automata should run (in case the user wishes to average the needed time)
* maIterations -> Number representing the Generations to be run (here set to 11000)
* fishAge -> max Generation lifetime of fish (represented in >0 numbers)
* sharkAge -> max Generation lifetime of Sharks (represented in <0 numbers)
* fishBreedingAge -> Generation lifetime from which the fish can reproduce
* sharkBreedingAge -> Generation lifetime from which the shark can reproduce

At the beginning of the function, an output is given to the users console displaying the settings of the current program (such as grid size, number of threads, etc.)

The grid is being represented by a double pointer of type int.

After allocating memory the grid is filled with a random placement of fish, sharks and empty ocean. This is achieved via a random number generator and seed value (as long as seed remains the same the “random” sequence will stay the same.

During the calculation of a generation, each cell is being viewed individually giving the algorithm x\*y\*z {x &y are the grid specs, z is the amount of generations to be simulated} calculations to run.

If the cell is a shark, the program uses the same random number generator from the grid randomization to calculate the 1/48 chance of death. This is done by taking the modulus of the generated number over the base of 48, producing a number between 0-47.

This number is then compared to the selected value, if identical the shark will be considered dead.

The timer for this application is implemented using the clock\_t function provided by cpp.

The current system time is taken at the start of the generation process and again after all generations have been simulated. The two numbers are then subtracted revealing the elapsed time in milliseconds.

Here the output is being displayed in minutes, therefore the result is being divided by 60000.

If the user defines more than 1 loop, the program will return the amount of time used for each of the runtimes.

## Results

The following section shows results produced by the algorithm.

In the serial version, the output picture will always be the same if the randomization seed isn’t changed.

Runtime on the other hand will vary depending on what else the PC used has to handle as well as processing speeds and available cores.

### Screenshots

|  |  |
| --- | --- |
| Screenshot | Description |
| Figure 1 - Serial Generation 0 | Screenshot of a 500x500 Grid at generation 0.  Random distribution of Fish (~50%), Sharks (~25%) and ocean (~25%). |
| Figure 2 - Serial Generation 100 | Screenshot of a 500x500 Grid at generation 100.  Sharks and Fish start grouping together. |
| Figure 3 - Serial Generation 1000 | Screenshot of a 500x500 Grid at generation 1000.  Sharks and Fish forming bigger formations.  The edges of the grid math the corresponding opposite position indication that the torus wrapping forms an infinite ocean. |
| Figure 4 - Serial Generation 5000 | Screenshot of a 500x500 Grid at generation 5000.  Fish starting to form in a massive field and smaller patches within sharks. |
| Figure 5 - Serial Generation 10000 | Screenshot of a 500x500 Grid at generation 10000. |
| Figure 6 - Serial Generation 11000 | Screenshot of a 500x500 Grid at generation 11000. |

Table 1 - Serial implementation 500x500 Grid

### Performance Analysis

Graph 1 - Serial Automata

### Conclusions

The serial version of the implementation produces an exponential curve as runtime. If the grid size doubles, so does the runtime (+- an error margin; this margin depends on the runtime itself, the larger the runtime the larger the fluctuation).

# Parallelisation Potential

For this project, the runtime is being calculated from the start of the generations until all generations are completed. Therefore, this section will only describe and cover code segments which are being timed. All other segments are being considered sequential and completed prior to application start.

## Sequential Sections

Parts of the serial source code cannot be parallelised due to dependencies or computational limitations such as processors or development tools.

In case of this application, the following sections are not parallelisable:

* Generations:

Since each generation is being calculated from the results of the previous, there is no way to parallelise the 11000 generations run by this application.

* Grid Analysis:

The grid (ocean) has a cell count of X\*Y. Due to computational limitations (4 logical cores) the only feasible parallelisation method, is to split one of the sides (x or y) across multiple threads. Therefore, at least half of the Gird size will have to be run in a sequential manner within each thread.

* Grid Shifting:

See explanation “Grid Analysis”

* Generation operations:

After each generations grid has been updated and shifted, there are a few sequential commands left to execute, such as increasing generation count, printing the new grid if applicable or calculating and printing the total runtime after all generations have been simulated.

With a speed-up of just under x3, indications are that close to 90% of the serial code have been successfully parallelised.

[NOTE: this number has been extrapolated from the speed up for ~81 to ~30 min using OMP at the greatest tested grid size and the graph displaying Amdahl’s law introduced in the Introduction lecture slides [graph point for 4 cores]].

## Parallel Sections

As already somewhat explained in the previous section, the only parallelised parts of the serial code are the two outer for loops in the grid calculation and grid shifting.

This will cut these loops into X segments (depending on number of threads used) and distribute them across all available processors.

In addition, there are other segments that could be parallelised, but would gain next to nothing in terms of runtime, either due to neglectable runtimes of these segments or not available processing power (more in the Load Balancing section of this report).

These sections are:

* Inner for loop
* If very high processor count would be available a collapsed for loop could spread the load more efficiently, this would require more processors then the large side of the grid to have a noticeable effect.
* Neighbour checking
* Printing of visual output (ASCII output would have to run in a sequential manner)

When using MPI or Hybrid parallelisation, there is a possibility (not tested in any way) to parallelise the communication between the different processes, possibly cutting needed runtime in half (if each circle [Left and right communication] is run in a separate thread)

[NOTE: more information on the communication scheme can be found in the MPI section of this report]

## Load Balancing

The problem of load balancing is mostly relevant when using MPI or high number of threads/processors in OMP.

In general, the parallel segments of the source code should be distributed across all needed/available processors in such a way that neither has a higher or lower load then the other. (this can further be optimised in giving smaller bits to slower processors and larger bits to faster ones)

While OMP will cut for loops into roughly equal pieces depending on threads used, a load balancing problem could occur if multiple parallel sections are overlapping. In that case it is possible for the communication, requisition of resources and CPU scheduling delays to outweigh any gain the runtime might have from parallelisation. (more information can be found in the conclusion sections of each parallelisation method)

When using MPI it is up to the programmer to make sure the load is distributed equally across all processes. It is undesirable to just divide the grid by the number of processes. This could lead to a situation where 1 processor has only 1 or 2 rows of information to run while the rest run 10x as many, effectively wasting a processor and slowing calculations with idle resources.

IT is a better practice to give all processes the same amount of work and split the remaining work across multiple processes, effectively reducing overhead and idle time to a minimum.

For MPI and Hybrid practices it is also best to not overload the capabilities of the given machine. If the number of started processes exceeds the total number of processors available, a communication overhead will be created, slowing runtime and possibly negating any gain of parallelisation. (more to this in the MPI and Hybrid sections).

Since the PC used for testing had 4 logical processors available, the best possible setup is by splitting load across these 4 processors:

* 4 Threads (Using 8 threads would sometimes produce slightly faster speeds then using 4)
* 4 Processes
* In case of Hybrid, testing showed that using 2 or 4 threads with 2-4 processes ran for about the same timeframe (slight diversions are probably caused by other tasks interfering with processing resources)

## Critical Sections/Synchronisation Points

For this project, each of the cells are being treated as individual computations, making them perfect for parallelisation. This has the advantage that splitting the cell calculation across processors does not create any critical sections within the code. Process creates needed variables for each single cell therefore eliminating race conditions and making variable calculations accurate and reliable.

The only exception to this is the use of a random number generator for the occasional death of sharks. Since the used random generator is not thread safe, a race condition is created. This race condition has no critical effect on the execution of the application.

The only problem arising from this is the fact that the use of parallelisation, unlike the serial version, will not give the same grid results.

Any attempt to use a thread safe method of sorts has resulted in unacceptable outcomes, where fish or sharks die out again or the computational time went through the roof. Therefore the non-thread safe method was left in place to create running and stable versions of parallelisation methods.

Synchronisations points for OMP are in place by creating and closing parallel sections around:

* Grid outer for loop
* Grid shifting outer for loop

When using MPI synchronisation points have been set after:

* Grid shifting

This barrier is in place to make sure that each processor has completed its calculations and copy cycle and is ready to start the next generation.

* Visual output (if active)

Makes sure the user sees a complete visual representation of the current grid on the screen

* Displaying runtime on the user console

Clustering the runtime output for simpler reading purpose

## Global Operations

For the use of MPI a communications system must be in place to fill needed information at the beginning of each generation cycle. This system represents the only global operations used within the parallel code.

# Parallelisation Method: OMP

## Algorithm

For the implementation of OMP parallelisation, the serial source code serves as the template to be parallelised.

Using the same set of rules and methods as in the serial implementation, the runtime will be speed up by parallelising certain aspects of the template code to generate a faster more efficient execution and resource management.

For this implementation and due to PC limitations, only one segment is parallelised at the same time. This means that if a section is running in parallel no other internal part of these parallel threads will be run in any other way then sequential execution.

If enough resources would be available, it is possible to increase efficiency by a minor factor by additionally parallelising thread internal segment for a small decrease in runtime.

Here the parallelisation focuses on reducing big loads onto different threads, spreading them across several logical processors (test PC with 4 logical processors).

The segments of the serial template that have been parallelised are:

* Memory allocation for “ocean” grid:

This segment is not part of the runtime calculation but has been parallelised to speed up non-critical sections of the preparation code for overall faster executing time (changes in this area will not affect the calculated runtime, it will only speed up the calculation of the simulation execution.

* Grid calculation:

The grid calculation has a total amount of X\*Y (grid size) cells to cycle through individually. This represents a perfect opportunity for parallelisation. By splitting the Outer or inner for loop across multiple threads the calculation count that must be processed gets cut to GridSize/Threads (assuming Threads=logical processors of target computer).

It is also possible to run a segment on a higher amount of threads than the number of processors available. This will force communication and sharing of resources between the individual threads. Should the grid be cut into small enough pieces with as little as possible overhead on thread count, a similar or even slightly faster executing time can be achieved. IF the overhead exceeds the potential gain the runtime will be quite a bit slower (more on this in the conclusion section).

* Grid shifting:

The grid shifting has the same parallelisation as described in the grid calculation.

Other possible parallelisation segments include (not been parallelised for stated reasons):

* Randomization of start grid:

If the randomization of the grid would have been parallelised, another race condition for the random number generator would have been created. This would have further changed the visual outcome of the simulation. Since the randomization is not included in the calculation of the runtime it was not parallelised to generate a constant starting point.

* Printing of visual output (NOT ASCII):

Since the printing of using pixels is a thread safe algorithm, it would have been possible to parallelise the generation of the print out. The print out is not recorded within the runtime calculated during testing, for this reason no parallelisation took place to create a universal printing algorithm to be used in the serial as well as the OMP implementation.

* Neighbour checking within grid calculation:

Neighbour checking is a small and relatively cheap process. Since the outer for loop of the grid calculation would already take up most if not all processors. Parallelising this segment would create an overhead that would most likely slow down runtime by forcing resource management to spread to many tasks across not enough processes. This would risk a slower overall runtime with the only gaining a small amount of time in a best-case scenario.

## Boundary implementation

The way grid boundaries of this implementation of OMP parallelisation is the same as used in the serial implementation.

The code will “calculate” the positions of neighbour cells before neighbours are checked. While this requires more lines of code, especially when checking neighbours, it is the fastest since no memory needs to be allocated for rows are grids and no time is wasted copying those cells into their proper positions.

Here the code calculates the values for “left”,” right”,” up” and “down”. Those values will be used to represent all 8 neighbours of the current cell.

## Source-Code

The source code for the OMP implementation utilizes the serial implementation as a template, therefore all the code is identical with a few minor changes to allow for parallelisation.

* OMP.h:

OMP header file, which allows access to OMP functions and tools

* Threadnumber:

A predefined number of threads which are used to create the parallelisation.

OMP will create this number in parallel threads.

Since this number is coded as a definition into the source code, the project has to be rebuild when the number of threads is going to change. (This could be outsourced into a parameter given to the application on start-up, but has not been implemented in this case)

* Parallel sections:

The implementation uses 3 parallel sections to enhance the runtime of the overall program. Only two of those sections are considered when calculating the runtime of the generation calculation.

* + Memory allocation:

Speed up for the overall application runtime, by parallelising the outer for loop of the grid. (no changes to the calculated results)

* + Grid Calculation:

Enhancing the runtime by splitting the outer for loop across given amount of threads.

* + Grid shifting:

Enhancing the runtime by splitting the outer for loop across given amount of threads.

## Results

The following section shows results produced by the algorithm.

Using the OMP implementation of the cellular automata, will provide the user with a possibly different Visual output every time it will be run. This is due to the race condition created by using a random number generator inside omp parallelised sections of the code.

The result depends on the timing that each thread reads an information from said generator.

### Screenshots

|  |  |
| --- | --- |
| Screenshot | Description |
| Figure 7- OMP Generation 0 | Screenshot of a 500x500 Grid at generation 0.  Random distribution of Fish (~50%), Sharks (~25%) and ocean (~25%). |
| Figure 8 - OMP Generation 100 | Screenshot of a 500x500 Grid at generation 100.  Sharks and Fish start grouping together. |
| Figure 9 - OMP Generation 1000 | Screenshot of a 500x500 Grid at generation 1000.  Sharks and Fish forming bigger formations.  The edges of the grid math the corresponding opposite position indication that the torus wrapping forms an infinite ocean. |
| Figure 10 - OMP Generation 5000 | Screenshot of a 500x500 Grid at generation 5000.  Fish starting to form in a massive field and smaller patches within sharks. |
| Figure 11 - OMP Generation 10000 | Screenshot of a 500x500 Grid at generation 10000. |
| Figure 12 - OMP Generation 11000 | Screenshot of a 500x500 Grid at generation 11000. |

Table 2 - OMP implementation 500x500 Grid

### Performance Analysis

Graph 2 - OMP 100x100

Graph 3 - OMP 200x100

Graph 4 - OMP 200x200

Graph 5 - OMP 500x200

Graph 6 - OMP 500x500

Graph 7 - OMP 500x1000

Graph 8 - OMP 1000x1000

Graph 9 - OMP 2000x1000

Graph 10 - OMP 2000x2000

### Conclusions

Using a PC with 4 logical cores, would seem to provide the best results in total runtime, when using the same number of threads.

As seen in the graphical representations, the biggest increase in terms of difference between runtimes happens on the very first attempt (2 threads instead of 1). The graph form 1-4 threads resemble a parabolic function.

Once the number of threads exceed the maximum number of logical processors, an increase in runtime can be observed. This is due to the PCs resource manager kicking in, trying to schedule resource across multiple threads. In that case, the communications and wait times negate some of the gains from having smaller grid sizes to calculate in each of the individual threads.

The tests have shown that after the first increase in runtime (here 5 threads) the runtime slowly begins to decrease again, due to smaller and smaller grid sizes being calculated.

It can also be observed that when 8 threads (highest tested thread count) is reached, the grid will be cut in such small pieces that the gain negates the cost of resource management, thereby running at roughly the same speed, or in some cases even slightly faster, as the 4 threads version.

# Parallelisation Method: MPI

## Algorithm

The algorithm for the MPI implementation, consists out of parts form the serial implementation:

* Source code definitions
* Generation calculation
* Grid shifting

In addition, the MPI solution has multiple new or changed approaches to create a working solution.

These changes deal with:

* Grid creation and memory allocation

The grid creation for the MPI implementation involves two steps

* + Original grid orientation:

To determine the grid orientation the code checks for the larger side of the provided grid size, if the grid is squared it will default to the X-side.

This determination is also later used to determine the loop starting points for the grid analysis and shifting.

* + Partitioning of the grid size:

After the grid orientation has been determined, the next step is the creation of a partial grid for each of the processes. This is done but dividing the longer side of the array into equal pieces for each process. If the division leaves remaining cells they’ll be split among all available processes to counter act and prevent a load imbalance.

Each individual grid is then assigned two ghost rows representing cells from neighbouring processes (the surrounding process IDs).

When the size is determined, the memory is allocated accordingly.

* Printing a visual output

For the visual output, a new function has been added to the ConsoleIO.h.

This function is specifically designed for MPI use. Feeding information such as grid size and process rank to the function will allow the application to draw a coherent and correct picture onto the screen when using Pixel output.

* Communication between processes

To enable the algorithm to calculate each cell correctly, the outer cells need to know their neighbours form the other processes. This happens at the beginning of each generation.

For this each process has to communicate with its neighbour process. This is done by using the MPI functions:

* + MPI\_Send
  + MPI\_Recv

Both functions are blocking functions, meaning that they will hold execution of further lines until a corresponding send or receive happened.

It is possible to create a deadlock scenario where the program will stall out and never continue.

To prevent this from happening, the code assumes a circle layout of all processes in which each will communicate to its direct neighbours. Here one process (in this case ID0) will start the communication by sending out its information to its neighbour and then waiting to receive what it needs. This triggers a snowball effect in which each process sends its information to its neighbour.

Once the communication reaches ID0 again, that process will start the communication the other way to fill missing cells.

Once both loops are completed the algorithm proceeds with the generation calculation

It might be possible to speed up this process but determine multiple start points. So instead of having just one process initiate communication between 4 processes, a second id (lets say id2) could also start sending, possible cutting needed time in half for even lower depending on number of starting points and number of processes run.

This implementation only uses ID0 as starting point for the communication.

* Barrier points

To ensure that each of the processes is ready before performing critical sections of the code, mpi barriers are in place waiting for each process to reach this point before continuing.

The barriers are located:

* + After printing the current grid on the screen
  + After shifting the array in preparation for the new generation
  + Before calling MPI finalize

## Boundary implementation

Using MPI as the parallelisation tool, presented a new kind of boundary problem. Since every process only and exclusively knows his own little segment of the total grid, each process must communicate its border cells to the appropriate process for use.

This must happen before a new generation can be simulated.

To handle this boundary issue, the MPI version utilizes “wrapping” as well as ghost cells.

Each Process has its unique id which positions it at a specific section of the total array. In addition to its segment size (calculated from the number of processes and the total grid size), the side, which was the original longest, will receive an additional row at the start and end (Ghost cells for border calculation).

Should the original Grid have been a square, the X side will be used by default.

Now each process sends its outer cell rows (first and last, not the ghost cells) to the appropriate process. [left to ID-1 and right to ID+1].

When these cells are in place the normal wrapping algorithm from the serial and OMP variant of the application will be used to create a segment of the total torus which will allow for infinite ocean simulation.

## Source-Code

As with the OMP implementation, the MPI source code uses the serial solution as the basic template.

To be able to use MPI successfully multiple changes have been made to make the template MPI compatible:

* Grid rotation:

The first change, is the analysation of the given grid size. The application has to determine which side of the array is bigger (default is X), this is important when using the communication between processes.

* Grid partitioning:

Once the rotation of the grid has been determined, the next step is to split the grid size across the given processes.

For this, the grid side is being divided by the given process count.

* + If modulus of GridSide%ProcCount equals 0 then the internal grid will be set to Side/count +2 (the +2 will add the ghost cells needed for the simulation)
  + If modulus of GridSide%ProcCount is not equal 0, then the internal grid will be Side/count +2 and in addition the remaining cells will be split with 1 each across the first x processes. This will balance the workload equally over all processes.
* Communication:

MPI needs inter process communication to relay essential information across multiple processes. Here, border cells must be send to the surrounding processes to enable a correct calculation of each generation.

This communication takes place at the beginning of each generation cycle before the new grid is to be calculated. Each process will send its outer right and left cells (not the ghost cells) to the right and left (higher and lower process id) processes respectively.

Blocking communication MPI\_Send and MPI\_Recv is used for communication.

* Cell parsing:

This segment of the code is only relevant if Y is the large side of the original grid.

Since the implementation utilizes a double pointer array to represent the grid, a parsing method has to be implemented if a row marked with [x][y] {where y is the same number} is to be send to other processes.

For this a for loop is run before and after the communication takes place.

* + The first loop will copy the selected row into a 1d buffer array which will then be send to the desired process
  + The second loop will take the information contained within the received array and parse it ack into the ghost cells of the new process.
* Loop Starting points:

Since the internal grid of each processor contains ghost cells, dynamical starting and end points for the loops have to be created in order to not check and change these cells.

This is done by utilizing the orientation of the original grid. Each loop, inner and outer) have an extra +- event within their limits which is represented by a converted true false using X as the reference side {is X the bigger side, for loop for x would start with original +1 and end at original -1 while Y is unchanged, is Y the larger side Y will start with the changes while X remains original}

* MPI Barriers:

MPI Barriers are in place to shield critical sections from being executed before they are ready. These barriers are located after:

* + After printing the current grid on the screen
  + After shifting the array in preparation for the new generation
  + Before calling MPI finalize

## Results

The following section shows results produced by the algorithm.

As with the serial implementation, using only MPI will generate the same output every time the application is run, even though it is calculating the simulation in a parallel manner.

This output will differ from the serial version, because each process has its own unique randomization seed (here the process ID has been used to initialize the random number generator).

### Screenshots

|  |  |
| --- | --- |
| Screenshot | Description |
| Figure 13 - MPI Generation 0 | Screenshot of a 500x500 Grid at generation 0.  Random distribution of Fish (~50%), Sharks (~25%) and ocean (~25%). |
| Figure 14 - MPI Generation 100 | Screenshot of a 500x500 Grid at generation 100.  Sharks and Fish start grouping together. |
| Figure 15 - MPI Generation 1000 | Screenshot of a 500x500 Grid at generation 1000.  Sharks and Fish forming bigger formations.  The edges of the grid math the corresponding opposite position indication that the torus wrapping forms an infinite ocean. |
| Figure 16 - MPI Generation 5000 | Screenshot of a 500x500 Grid at generation 5000.  Fish starting to form in a massive field and smaller patches within sharks. |
| Figure 17 - MPI Generation 10000 | Screenshot of a 500x500 Grid at generation 10000. |
| Figure 18 - MPI Generation 11000 | Screenshot of a 500x500 Grid at generation 11000. |

Table 3 - MPI implementation 500x500 Grid

### Performance Analysis

Graph 11 - MPI 100x100

Graph 12 - MPI 200x100

Graph 13 - MPI 200x200

Graph 14 - MPI 500x200

Graph 15 - MPI 500x500

Graph 16 - MPI 500x1000

Graph 17 - MPI 1000x1000

Graph 18 - MPI 2000x1000

Graph 19 - MPI 2000x2000

### Conclusions

The recorded data shows that the runtime behaves in a comparable way as the OMP version when using 1-4 Processes (equal or less than the available logical processors on the testing machine).

Once this threshold is crossed the communication between the different processes as well as the constant activation and deactivation from the cpu manager, drastically increase the time needed to complete the simulation.

When used with small grid sizes this increase in runtime even exceeds the needed time for the serial application. If the grid size is getting bigger, it is shown that the runtime will still increase quite a lot but it will remain under the serial threshhold, providing a limited acceleration of the calculation process.

# Parallelisation Method: Hybrid

## Algorithm

The algorithm created for the hybrid parallelisation, is a combination of the OMP and MPI version of the program.

The MPI version is used to create a template which is then enhanced with OMP parallelisation methods to create a hybrid solution.

The algorithm follows the same set of rules as the serial implementation.

In addition to the parallelisation, which is provided by splitting the grid across multiple processes, OMP techniques re used to increase the runtime gain within each of the processes.

OMP is applied to the following parts of the source code:

* Memory allocation for “ocean” grid:

This segment is not part of the runtime calculation but has been parallelised to speed up non-critical sections of the preparation code for overall faster executing time (changes in this area will not affect the calculated runtime, it will only speed up the calculation of the simulation execution.

* Grid calculation:

The grid calculation has a total amount of X\*Y (grid size) cells to cycle through individually. This represents a perfect opportunity for parallelisation. By splitting the Outer or inner for loop across multiple threads the calculation count that must be processed gets cut to GridSize/Threads (assuming Threads=available logical processors of target computer).

It is also possible to run a segment on a higher amount of threads than the number of processors available. This will force communication and sharing of resources between the individual threads. Should the grid be cut into small enough pieces with as little as possible overhead on thread count, a similar or even slightly faster executing time can be achieved. IF the overhead exceeds the potential gain the runtime will be quite a bit slower (more on this in the conclusion section).

* Grid shifting:

The grid shifting has the same parallelisation as described in the grid calculation.

In addition to these three locations, there are a few other locations which could be parallelised:

* Communication:

The communication between the different processes might be able to be parallelised for the clockwise and anti-clockwise communication cycle. This could have been done by enveloping these segments in OMP Sections.

This was not done to make sure that no deadlock will occur if something went wrong with the threads (no time to be tested properly, but is a possibility, to reduce runtime by a relatively small amount of time)

* Cell parsing:

The parsing of cells, which must take place in case of Y being the bigger side of the grid, could be parallelised by splitting the for loop across threads. This has not been implemented due to time constraints and negligible time gain. (considering that runtimes for the tested rotated array seems similar to the other grids tested)

* Printing of visual output (NOT ASCII):

Since the printing of using pixels is a thread safe algorithm, it would have been possible to parallelise the generation of the print out. The print out is not recorded within the runtime calculated during testing, for this reason no parallelisation took place to create a universal printing algorithm to be used in the serial as well as the OMP implementation.

## Boundary implementation

The implemented boundary handling, is identical to the way used in the MPI version.

It combines:

* “Wrapping”
* Ghost-Cells

To create a process which will work when using different processes.

By communicating the outer cells to the neighbouring process ids before each generation is being simulated, each process knows all cells needed to create and “unlimited” ocean.

The torus segment is then completed by checking the neighbours using the calculated positions left, right, up, down.

## Source-Code

The source code for the hybrid implementation adds some minor changes to the MPI template, to allow for OMP parallelisation besides what MPI offers.

* OMP.h:

OMP header file, which allows access to OMP functions and tools

* Threadnumber:

A predefined number of threads which are used to create the parallelisation.

OMP will create this number in parallel threads.

Since this number is coded as a definition into the source code, the project has to be rebuild when the number of threads is going to change. (This could be outsourced into a parameter given to the application on start-up, but has not been implemented in this case)

* Parallel sections:

The implementation uses 3 parallel sections to enhance the runtime of the overall program. Only two of those sections are considered when calculating the runtime of the generation calculation.

* + Memory allocation:

Speed up for the overall application runtime, by parallelising the outer for loop of the grid. (no changes to the calculated results)

* + Grid Calculation:

Enhancing the runtime by splitting the outer for loop across given amount of threads.

* + Grid shifting:

Enhancing the runtime by splitting the outer for loop across given amount of threads.

## Results

The following section shows results produced by the algorithm.

Working from the base generation from the MPI version of the parallelisation, the results will change as they did when working with the OMP variant.

Since using a random number generator is not thread safe, the results change depending on which thread wins the existing race condition.

### Screenshots

|  |  |
| --- | --- |
| Screenshot | Description |
| Figure 19 - Hybrid Generation 0 | Screenshot of a 500x500 Grid at generation 0.  Random distribution of Fish (~50%), Sharks (~25%) and ocean (~25%). |
| Figure 20 - Hybrid Generation 100 | Screenshot of a 500x500 Grid at generation 100.  Sharks and Fish start grouping together. |
| Figure 21 - Hybrid Generation 1000 | Screenshot of a 500x500 Grid at generation 1000.  Sharks and Fish forming bigger formations.  The edges of the grid math the corresponding opposite position indication that the torus wrapping forms an infinite ocean. |
| Figure 22 - Hybrid Generation 5000 | Screenshot of a 500x500 Grid at generation 5000.  Fish starting to form in a massive field and smaller patches within sharks. |
| Figure 23 - Hybrid Generation 10000 | Screenshot of a 500x500 Grid at generation 10000. |
| Figure 24 - Hybrid Generation 11000 | Screenshot of a 500x500 Grid at generation 11000. |

Table 4 - Hybrid implementation 500x500 Grid

### Performance Analysis

Graph 20 - Hybrid 100x100

Graph 21 - Hybrid 200x100

Graph 22 - Hybrid 200x200

Graph 23 - Hybrid 500x200

Graph 24 - Hybrid 500x500

Graph 25 - Hybrid 500x1000

Graph 26 - Hybrid 1000x1000

Graph 27 - Hybrid 2000x1000

Graph 28 - Hybrid 2000x2000

### Conclusions

For the representation of the results gathered during testing of the hybrid parallelisation method, a 3-d surface diagram is used.

Using this representation allows for a fast and overall view of which setting (process number vs thread number) will provide the best result.

Analysing the results shows, that the setting of splitting the available logical processors across both processes and thread number will result in the best possible speed up.

A second observation indicates, that using more threads than available cores is less runtime impairing then running the application on more than 4 processes. This is due to resource management for threads being relatively cheap when compared to running cpu management as well as overhead communication.

# Conclusion

In conclusion, the tests have shown that each of the parallelisation methods have their own best application depending on the power of the selected hardware.

* OMP:

OMP parallelisation shows the most promising results provided by the testing.

In comparison to the other methods it has the lowest runtime with the least amount of additional work.

When compared to MPI, it stands put that when the thread count exceeds the maximum number of logical processors, there is only a slight increase, compared to the overall results, to the runtimes gathered when using equal or less threads.

The best recorded % gain, happened when using 2 threads almost cutting the needed runtime in half.

* MPI:

Using MPI parallelisation has a much larger code altering requirement, then OMP. This is due to having to manually program the divided cell grid into each processor.

While the work load is higher the gain in comparable to the results when using OMP parallelisation (using less or equal threads and processes than logical processors).

Should the number of run processes exceed the number of processing units, the loss of runtime is significantly higher, than the loss produces by using OMP.

This makes OMP the better choice if a single pc is used to run the proposed simulation.

As with the OMP parallelisation the most % gain, from the last set up, is recorded when using 2 processors, almost cutting runtime in half.

* Hybrid:

The hybrid parallelisation method combines OMP and MPI making it the perfect choice if the simulation can be run across a network of computers each with at least 2 logical processors.

By using Hybrid parallelisation, the workload can be split across multiple pc each running a process of the total simulation. This will split the calculated grid size across N networked PCs. In addition to this individual gain by running smaller sections at the same time, the use of OMP will further enhance the processing gain by dividing the internal grid into parallel parts splitting it across M logical processors.

The results of testing would highly depend on the number of PC within the network on their cpu count and speed, meaning that this method of parallelisation is most efficient within a computer network.

# Annex

## **Table of Figures**

[Figure 1 - Serial Generation 0 8](#_Toc480114808)

[Figure 2 - Serial Generation 100 8](#_Toc480114809)

[Figure 3 - Serial Generation 1000 8](#_Toc480114810)

[Figure 4 - Serial Generation 5000 9](#_Toc480114811)

[Figure 5 - Serial Generation 10000 9](#_Toc480114812)

[Figure 6 - Serial Generation 11000 9](#_Toc480114813)

[Figure 7- OMP Generation 0 15](#_Toc480114814)

[Figure 8 - OMP Generation 100 16](#_Toc480114815)

[Figure 9 - OMP Generation 1000 16](#_Toc480114816)

[Figure 10 - OMP Generation 5000 16](#_Toc480114817)

[Figure 11 - OMP Generation 10000 16](#_Toc480114818)

[Figure 12 - OMP Generation 11000 17](#_Toc480114819)

[Figure 13 - MPI Generation 0 25](#_Toc480114820)

[Figure 14 - MPI Generation 100 26](#_Toc480114821)

[Figure 15 - MPI Generation 1000 26](#_Toc480114822)

[Figure 16 - MPI Generation 5000 26](#_Toc480114823)

[Figure 17 - MPI Generation 10000 26](#_Toc480114824)

[Figure 18 - MPI Generation 11000 27](#_Toc480114825)

[Figure 19 - Hybrid Generation 0 34](#_Toc480114826)

[Figure 20 - Hybrid Generation 100 34](#_Toc480114827)

[Figure 21 - Hybrid Generation 1000 35](#_Toc480114828)

[Figure 22 - Hybrid Generation 5000 35](#_Toc480114829)

[Figure 23 - Hybrid Generation 10000 35](#_Toc480114830)

[Figure 24 - Hybrid Generation 11000 35](#_Toc480114831)

## **Table of Tables**

[Table 1 - Serial implementation 500x500 Grid 9](#_Toc480114804)

[Table 2 - OMP implementation 500x500 Grid 17](#_Toc480114805)

[Table 3 - MPI implementation 500x500 Grid 27](#_Toc480114806)

[Table 4 - Hybrid implementation 500x500 Grid 35](#_Toc480114807)

## **Table of Tables**

[Graph 1 - Serial Automata 10](#_Toc480114776)

[Graph 2 - OMP 100x100 17](#_Toc480114777)

[Graph 3 - OMP 200x100 18](#_Toc480114778)

[Graph 4 - OMP 200x200 18](#_Toc480114779)

[Graph 5 - OMP 500x200 19](#_Toc480114780)

[Graph 6 - OMP 500x500 19](#_Toc480114781)

[Graph 7 - OMP 500x1000 20](#_Toc480114782)

[Graph 8 - OMP 1000x1000 20](#_Toc480114783)

[Graph 9 - OMP 2000x1000 21](#_Toc480114784)

[Graph 10 - OMP 2000x2000 21](#_Toc480114785)

[Graph 11 - MPI 100x100 27](#_Toc480114786)

[Graph 12 - MPI 200x100 28](#_Toc480114787)

[Graph 13 - MPI 200x200 28](#_Toc480114788)

[Graph 14 - MPI 500x200 29](#_Toc480114789)

[Graph 15 - MPI 500x500 29](#_Toc480114790)

[Graph 16 - MPI 500x1000 30](#_Toc480114791)

[Graph 17 - MPI 1000x1000 30](#_Toc480114792)

[Graph 18 - MPI 2000x1000 31](#_Toc480114793)

[Graph 19 - MPI 2000x2000 31](#_Toc480114794)

[Graph 20 - Hybrid 100x100 36](#_Toc480114795)

[Graph 21 - Hybrid 200x100 36](#_Toc480114796)

[Graph 22 - Hybrid 200x200 37](#_Toc480114797)

[Graph 23 - Hybrid 500x200 37](#_Toc480114798)

[Graph 24 - Hybrid 500x500 38](#_Toc480114799)

[Graph 25 - Hybrid 500x1000 38](#_Toc480114800)

[Graph 26 - Hybrid 1000x1000 39](#_Toc480114801)

[Graph 27 - Hybrid 2000x1000 39](#_Toc480114802)

[Graph 28 - Hybrid 2000x2000 40](#_Toc480114803)