Real Time Programming

Parallel Programming

Gordon Johnson
K1451760

Table of Contents

Introduction

I am going to create a prey/predator simulation, using cellular automata, to demonstrate the performance and behaviours of different parallel programming techniques. My goal is to deliver four implementations, in one application. To achieve this, an options menu will need to be implemented, this application will be coded in C++. A serial implementation, will be implemented to provide a base line. An OpenMP implementation, will be used to demonstrate the multi-threaded and an MPI implementation, will be used for multi-processor demonstrations. Finally, a hybrid implementation, will make use of both OpenMP and MPI. This will be done using ASCI text, this provides the application the opportunity to have customisation once launched. SDL, will be used to provide a graphical output for the simulation. This will provide an overhead on performance, so an option will be available to exclude this when testing.

Performance testing, will be conducted on each implementation, using grid sizes 100x100, 1,000x1,000 and 10,000x10,000. The first two grid sizes, will be tested over 10,000 iterations, with the final grid size being tested over 1,000 iterations. The OpenMP, will be tested using 2, 4, 6 and 8 threads. MPI, will be tested using 2, 4, 6 and 8 processes. The Hybrid, will be tested using (threads, processes), these will be performed using the following strategy (2, 2), (4, 4), (6, 2) and (2, 6).

The following, presents the specification of the device used, for all the performance tests:

CPU: Intel® Core™ i7-7700HQ Quad-Core Processor with Hyper-Threading 2.8GHz / 3.8GHz (Base/Turbo)

SSD: 512GB SSD (PCIe M.2)

Memory: 16GB dual-channel onboard memory (DDR4, 2400MHz)

GPU: NVIDIA® GeForce® GTX 1060 (6GB GDDR5 VRAM)

OS: Windows 10 (64-Bit)

GitHub:	https://github.com/LordGee/prey_pred
Box:	https://kingston.box.com/s/cp9oauh5qcui3y4j6e5g87lvxxfl3v3j
YouTube:	https://youtu.be/9KmIm2TVGWg

Serial Implementation

Algorithm / Strategy

The first priority, is to find the best process, to manage each cell within the grid, identifying the important attributes and how they are going to be represented. To do this a class is created called Cell, with two attributes that are both initialised to zero. These attributes are Type and Age. The attribute Type, indicates whether the cell is Prey, Predator or Empty. The mechanics for Type, are as follows:

- -1 = Predator
- 0 = Empty
- 1 = Prey

The Age attribute, will be used to store the incremental age of the cell. Unless the cell is empty, then it will remain at zero value.

The main loop, for the simulation, will have to iterate through every cell within the grid, whilst analysing all eight neighbouring cells. To accomplish this, the use of four 'if' statements will be used, to check if either the x or y coordinates, are out of bounds. If so, they will change the value to be checked to the opposite side of the grid, for example, if the y index returns a -1 value, then the overall height value will be added to this, to provide the highest possible index that y could be. As each neighbouring cell is checked, if the type of cell is not empty, then the type is accumulated within a variable, for later use. Also, when identifying if the cell is prey or predator, a check is performed to determine whether they are of breeding age, this is accumulated in an age variable.

Once all the neighbouring checks have been completed, it is then time to check what the new state of the cell will be. To accomplish this, the check statements will be broken down into three categories, depending on their current type, e.g. manage prey, manage predator and manage empty. These three checks, will contain additional 'if' statements, to determine if any of the set rule conditions have been met.

For a given cell containing a fish,

- Fish live for 10 generations
- If >= 5 neighbours are sharks, fish dies (shark food)
- If all 8 neighbours are fish, fish dies (overpopulation)
- If a fish does not die, increment age

For a given cell containing a shark,

- Sharks live for 20 generations
- If >= 6 neighbours are sharks, and fish neighbours = 0, then the shark dies (starvation)
- A shark has a 1/32 (.031) chance of dying due to random causes
- If a shark does not die, increment age

For a given empty cell,

- If there are >= 4 neighbours of one species, with >= 3 of them of breeding age and there are <= 4 of the other species, then create a species of that type.

The new state of the cell, will be recorded into a temporary grid. This, will be an independent array, of equal size to the actual grid. This is due to the main loop having not concluded through all cells yet, but the remaining checks still need to be based upon the original states. After all cells have been iterated through, the temporary grid containing the new states, will be copied back to the main grid, so the process can start again for the next iteration.

Implementation

As discussed in the strategy, the need for an implementation of an object to manage each individual cell was created. In hindsight, this could have been simplified in a struct as opposed to a class, eliminating the need to declare the attributes and operations as public.

```
class Cell {
public:
	int type;
	int age;

public:
	Cell() {
		type = 0;
		age = 0;
	};
};
```

When a new instance of the simulator is created, the constructor initialises the required variables and sets up appropriate size vector arrays, for both the main and temporary grid. This constructor, will persist for every implementation, as the parent constructor.

```
Simulator::Simulator(int width, int height, int preyPercent, int predPercent, int ran-
domSeed, int threads, int proc) :
width(width), height(height), seed(randomSeed), numThreads(threads), numProc(proc) {
	prey = (float)preyPercent / 100.0f;
	pred = (float)predPercent / 100.0f;
	mainGrid = std::vector<std::vector<Cell>>(width);
	copyGrid = std::vector<std::vector<Cell>>(width);
	for (int x = 0; x < width; x++) {
		mainGrid[x] = std::vector<Cell>(height);
		copyGrid[x] = std::vector<Cell>(height);
	}
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			mainGrid[x][y].type = 0;
			mainGrid[x][y].age = 0;
		}
	}
}
```

After the constructor, the next operation populates the main grid with starting values. The values entered are set randomly, using a predefined seed for the random generator and predefined percentages of prey and predator.

```
void Serial::PopulateGrid() {
	srand(seed);
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			float random = (float)(rand()) / (float)(RAND_MAX);
			if (random < prey) {
				mainGrid[x][y].type = 1;
				mainGrid[x][y].age = 1;
			} else if (random < prey + pred) {
				mainGrid[x][y].type = -1;
				mainGrid[x][y].age = 1;
			} else {
				mainGrid[x][y].type = 0;
				mainGrid[x][y].age = 0;
			}
		}
	}
}
```

Once the main array has been generated, there are three entry points to the main loop, depending type of display the user option chooses. The three options are, graphical output, or no display. The code below, is for ASCI statistics, it loops through the simulator and outputs the statistics, to the console window.

```
void Serial::RunSimNoDraw(const int COUNT) {
	int counter = 0;
	clock_t t1, t2;
	float timer;
	while (counter < COUNT) {
		t1 = clock();
		deadPrey = 0, deadPred = 0;
		livePrey = 0, livePred = 0, empty = 0;
		UpdateSimulation();
		for (int x = 0; x < width; x++) {
			for (int y = 0; y < height; y++) {
				if (mainGrid[x][y].type > 0) {
					livePrey++;
				} else if (mainGrid[x][y].type < 0) {
					livePred++;
				} else {
					empty++;
				}
			}
		}
		t2 = clock();
		timer = (float)(t2 - t1) / CLOCKS_PER_SEC;
		UpdateStatistics(timer, counter, livePrey, livePred, empty,
			deadPrey, deadPred);
		counter++;
	}
}
```

Within the main loop, the following operation is called. This checks the state of each cell and updates the simulator. The first step, is to check all neighbouring objects of the given cell. The cell, depending on its current state, is then checked to determine its new state.

```
void Serial::UpdateSimulation() {
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// loop each neighbouring cell
				// check neighbouring cell states
			// Manage Prey
				// Check new state of cell
			// Manage Predator
				// Check new state of cell
			// Manage Empty
				// Check new state of cell
		}
	}
}
```

The following, initialises new variables to perform a count on the neighbouring cells, depending on their state. This is done, by using a double loop, which provides a value either -1, 0 or 1. If both loop values are set to 0, then this is the current cell and is ignored. New x and y values, are created by adding $x + i$, and $y + j$. The new value is then checked, to ensure that it is not out of bounds of the grid. If it is, then the value is recalculated to the other side of the grid. For example, if $y + j = -1$, then $y = y + j + height$. So, if height were to equal 100, then this will result in the new y being the last index in the grid $(0 + -1) + 100 = 99$. If the new grid index is a prey or predator, then the relevant variable is accumulated, whilst also checking if that type is greater or equal to the breeding age. This is also accumulated to its respective variable.

```
int preyCount = 0, preyAge = 0, predCount = 0, predAge = 0;
for (int i = -1; i < 2; i++) {
	for (int j = -1; j < 2; j++) {
		if (!(i == 0 && j == 0)) {
			int newX = x + i, newY = y + j;
			if (newY < 0) { newY = newY + height; }
			if (newX < 0) { newX = newX + width; }
			if (newY >= height) { newY = newY - height; }
			if (newX >= width) { newX = newX - width; }
			if (mainGrid[newX][newY].type > 0) {
				preyCount++;
				if (mainGrid[newX][newY].age >= PREY_BREEDING) {
					preyAge++;
				}
			} else if (mainGrid[newX][newY].type < 0) {
				predCount++;
				if (mainGrid[newX][newY].age >= PRED_BREEDING) {
					predAge++;
				}
			}
		}
	}
}
```

The following three operations, manage the new state of each cell. Depending on the cells current type, the new values are created and placed in a temporary grid. This ensures, the remainder of the simulation update, is based on the current values, not the new values.

```
if (mainGrid[x][y].type > 0) {
	//manage prey
	if (predCount >= 5 || preyCount == 8 || mainGrid[x][y].age > PREY_LIVE) {
		copyGrid[x][y].type = 0;
		copyGrid[x][y].age = 0;
		deadPrey++;
	} else {
		copyGrid[x][y].type = mainGrid[x][y].type;
		copyGrid[x][y].age = mainGrid[x][y].age + 1;
	}
}
```

This manages the predator cells, by performing a check based on the rules for if the predator dies. If any of these conditions are true, then the cell is set to a zero value, this indicates the new cell type is empty. As before, the predator remains as the type and the age is incremented.

```
else if (mainGrid[x][y].type < 0) {
	// manage predator
	float random = (float)(rand()) / (float)(RAND_MAX);
	if ((predCount >= 6 && preyCount == 0) || random <= PRED_SUDDEN_DEATH ||
			copyGrid[x][y].age > PRED_LIVE) {
		copyGrid[x][y].type = 0;
		copyGrid[x][y].age = 0;
		deadPred++;
	} else {
		copyGrid[x][y].type = mainGrid[x][y].type;
		copyGrid[x][y].age = mainGrid[x][y].age + 1;
	}
}
```

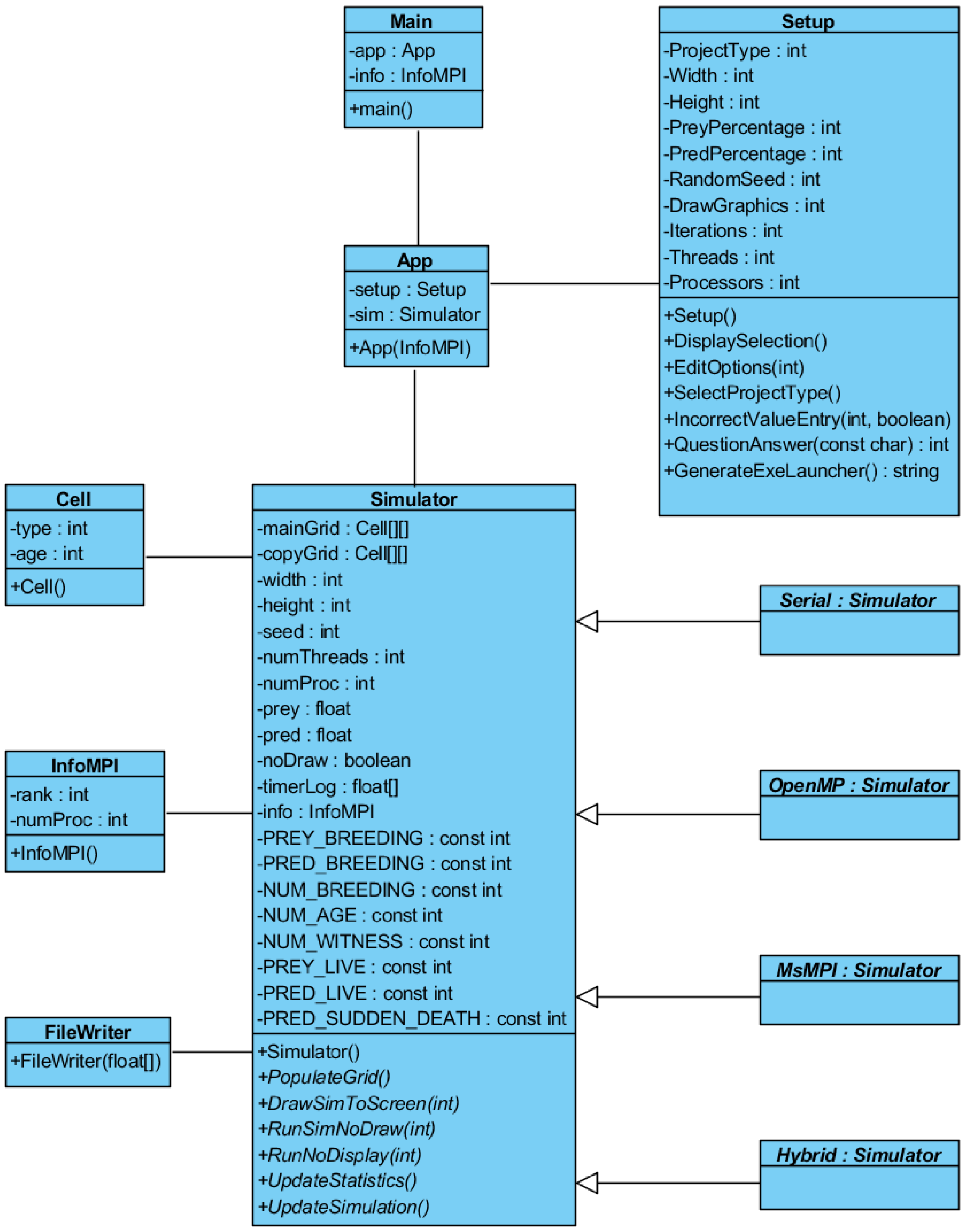
The final checks, are to manage the current empty cells, dependant on the surrounding cell types and age count, potentially a new type, either prey or predator, could be generated. If neither condition is met, then the cell remains empty.

```
else {
       // manage empty space
       if (preyCount >= NO_BREEDING && preyAge >= NO_AGE &&
                     predCount < NO_WITNESSES) {
              copyGrid[x][y].type = 1;
              copyGrid[x][y].age = 1;
       } else if (predCount >= NO_BREEDING && predAge >= NO_AGE &&
                     preyCount < NO_WITNESSES) {
              copyGrid[x][y].type = -1;
              copyGrid[x][y].age = 1;
       } else {
              copyGrid[x][y].type = 0;
              copyGrid[x][y].age = 0;
       }
}
```

Finally, once the main update loop has concluded, the temporary array is then copied, cell by cell, into the main array, ready for the next iteration.

```
for (int x = 0; x < width; x++) {
       for (int y = 0; y < height; y++) {
              mainGrid[x][y] = copyGrid[x][y];
       }
}
```

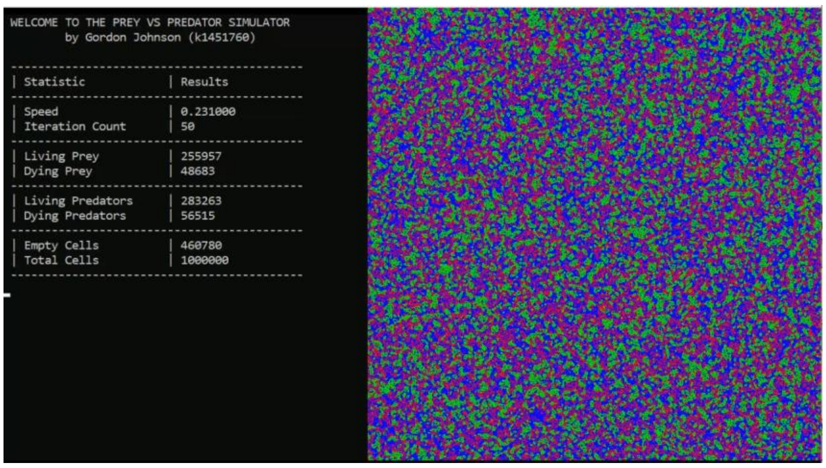
Class Diagram



Results

After some trial and error, the simulator worked as desired. The screen shots, provided below, were taken at certain iteration counts, to demonstrate the evolving simulation.

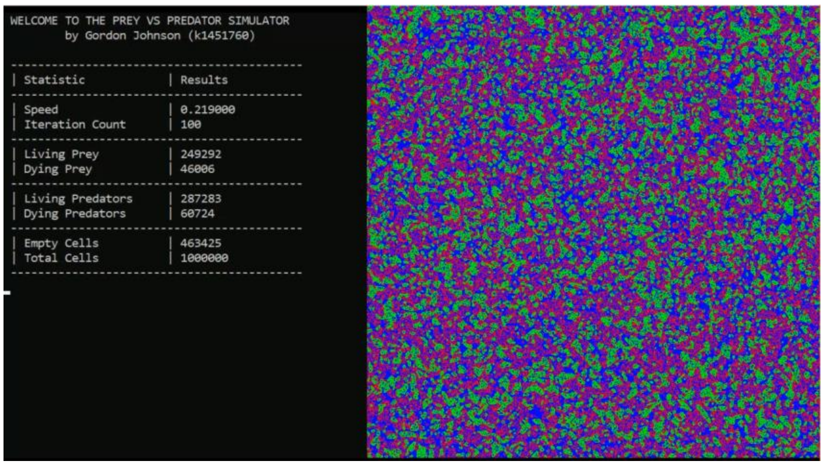
50 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.231000
| Iteration Count        | 50
---------------------------------------------
| Living Prey            | 255957
| Dying Prey             | 48683
---------------------------------------------
| Living Predators       | 283263
| Dying Predators        | 56515
---------------------------------------------
| Empty Cells            | 460780
| Total Cells            | 1000000
---------------------------------------------
```

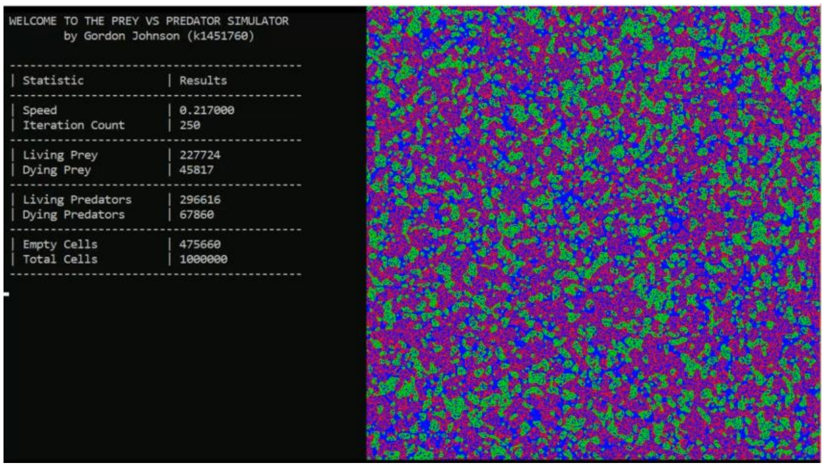
100 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.219000
| Iteration Count        | 100
---------------------------------------------
| Living Prey            | 249292
| Dying Prey             | 46006
---------------------------------------------
| Living Predators       | 287283
| Dying Predators        | 60724
---------------------------------------------
| Empty Cells            | 463425
| Total Cells            | 1000000
---------------------------------------------
```

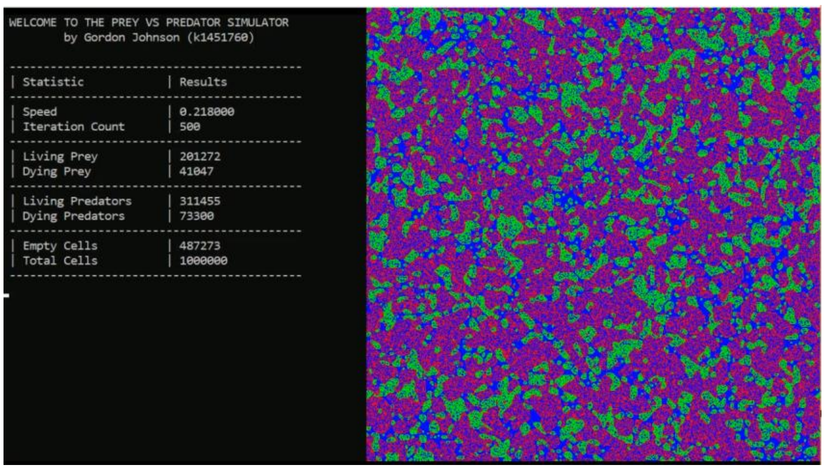
250 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.217000
| Iteration Count        | 250
---------------------------------------------
| Living Prey            | 227724
| Dying Prey             | 45817
---------------------------------------------
| Living Predators       | 296616
| Dying Predators        | 67860
---------------------------------------------
| Empty Cells            | 475660
| Total Cells            | 1000000
---------------------------------------------
```

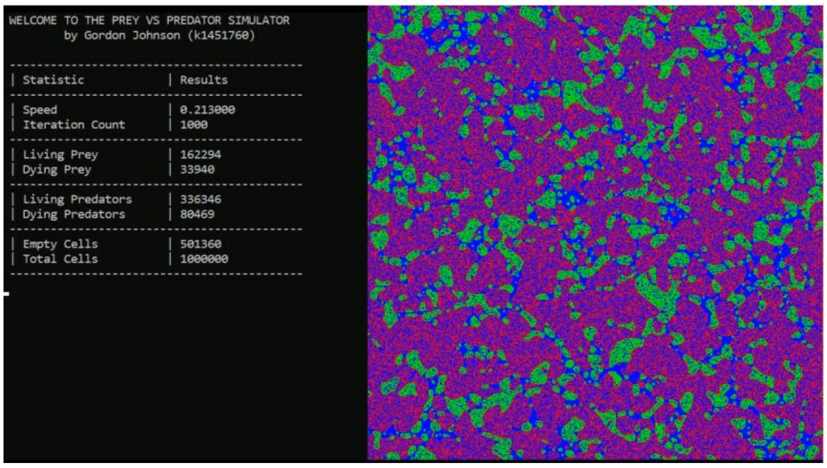
500 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.218000
| Iteration Count       | 500
-------------------------------------------
| Living Prey           | 201272
| Dying Prey            | 41047
-------------------------------------------
| Living Predators      | 311455
| Dying Predators       | 73300
-------------------------------------------
| Empty Cells           | 487273
| Total Cells           | 1000000
-------------------------------------------
```

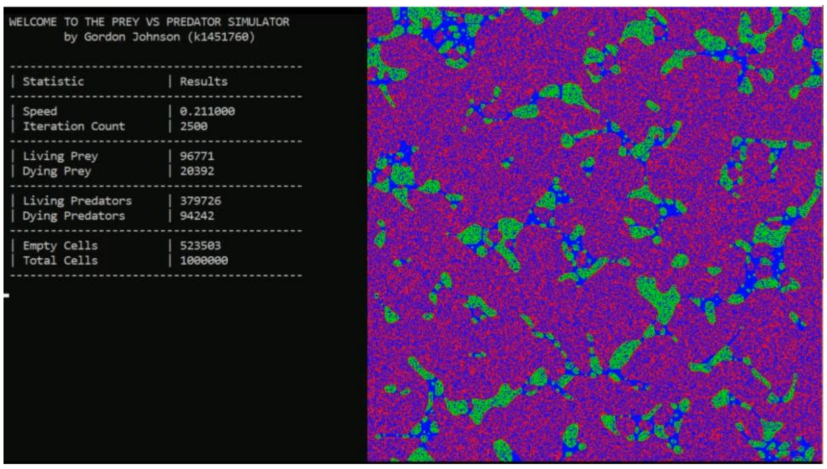
1000 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.213000
| Iteration Count       | 1000
-------------------------------------------
| Living Prey           | 162294
| Dying Prey            | 33940
-------------------------------------------
| Living Predators      | 336346
| Dying Predators       | 80469
-------------------------------------------
| Empty Cells           | 501360
| Total Cells           | 1000000
-------------------------------------------
```

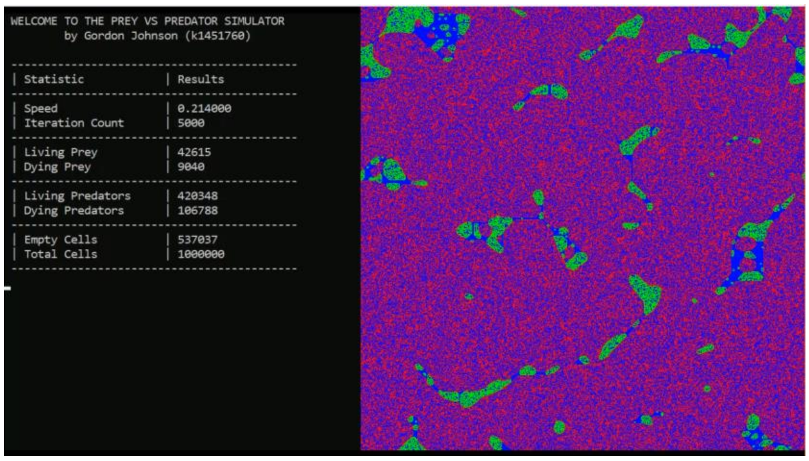
2500 iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.211000
| Iteration Count       | 2500
-------------------------------------------
| Living Prey           | 96771
| Dying Prey            | 20392
-------------------------------------------
| Living Predators      | 379726
| Dying Predators       | 94242
-------------------------------------------
| Empty Cells           | 523503
| Total Cells           | 1000000
-------------------------------------------
```

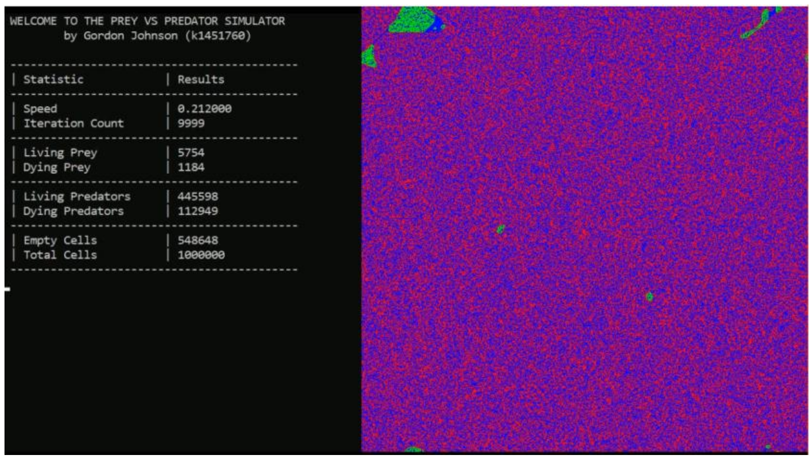
5000 Iterations



```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.214000
| Iteration Count       | 5000
-------------------------------------------
| Living Prey           | 42615
| Dying Prey            | 9040
-------------------------------------------
| Living Predators      | 420348
| Dying Predators       | 106788
-------------------------------------------
| Empty Cells           | 537037
| Total Cells           | 1000000
-------------------------------------------
```

10000 Iterations



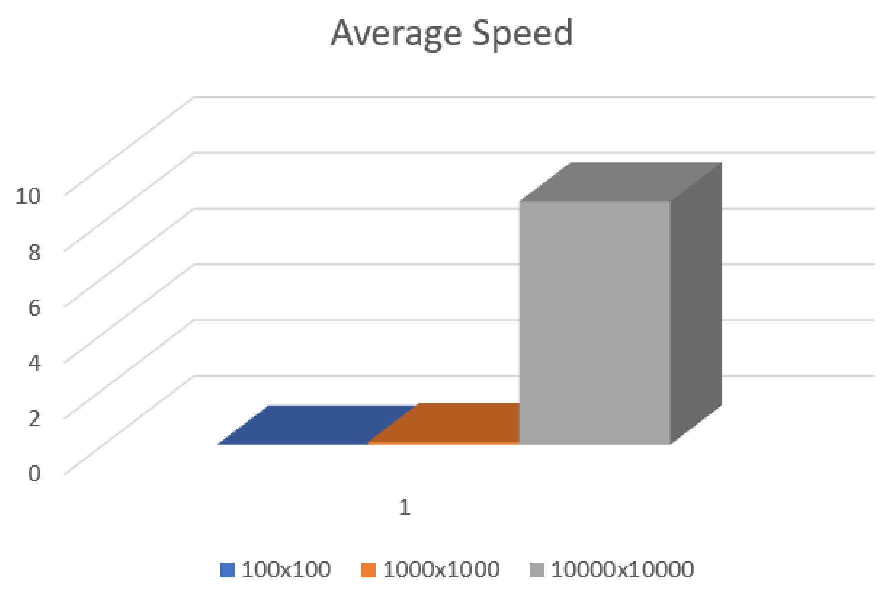
```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.212000
| Iteration Count       | 9999
-------------------------------------------
| Living Prey           | 5754
| Dying Prey            | 1184
-------------------------------------------
| Living Predators      | 445598
| Dying Predators       | 112949
-------------------------------------------
| Empty Cells           | 548648
| Total Cells           | 1000000
-------------------------------------------
```

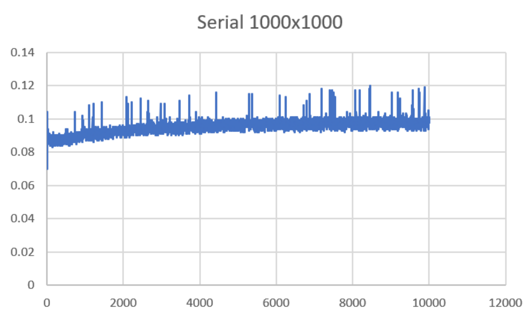
Performance

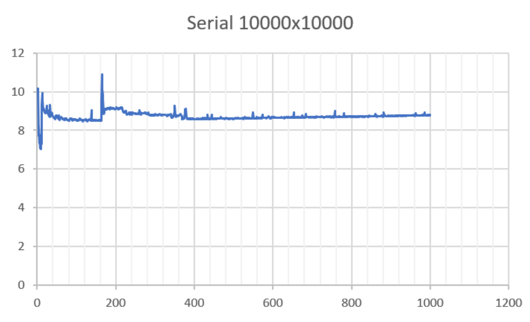
For this serial implementation, the following tests were conducted.

Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
Average Speed per Iteration	0.0005381	0.0951833	8.722674



As can be seen by the average iteration speed, the smaller grid sizes present a reasonable average time. However, for the largest grid size, each iteration is taking over 8 seconds to complete.





As can be seen from these charts, taken from the raw data, there is a gradual increase in seconds taken, indicative of a decline in performance over time.

Study of the potential for parallelisation

Sequential Section

From executing the application, the Main method, App and Setup classes, will remain sequential. Adding parallelisation to these areas, will provide no performance improvements to the application, as they are mainly user dependent. Any display methods or file writing should also remain sequential, this will ensure no duplicate outputs are displayed and like the previous methods, will provide no benefit to the simulation process of the application.

Potential Parallel Sections

The main three sections, that should be targeted for parallelisation, are all within the main Update Simulation loop.

```
void Serial::UpdateSimulation() {
	/* Target for Parallelisation */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// Prepare temporary grid
		}
	}
	/* Target for Parallelisation */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// loop each neighbouring cell
				// check neighbouring cell states
			// Manage Prey
				// Check new state of cell
			// Manage Predator
				// Check new state of cell
			// Manage Empty
				// Check new state of cell
		}
	}
	/* Target for Parallelisation */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// Copy temporary grid back to main
		}
	}

}
```

Additional consideration during the MPI implementation, is going to be required when performing the check of the neighbouring cells. Each process, will need to be prepared, prior to going into the evaluation loop. This ensures sufficient values are held, to enable all the neighbouring states to be checked.

Critical sections and points of synchronisation

In all instances of parallelisation, using a barrier will ensure synchronisation, for example, prior to the evaluation loop. It will be important to ensure all processes contain the relevant information, especially in the MPI implementation, ready for checking the neighbouring states, particularly if a neighbouring state is managed by a different process.

In addition, the OpenMP implementation will require a barrier, before the temporary grid is copied back to the main grid. This is to ensure that all threads, have evaluated all states, before continuing.

Potential issues of load balancing

There are two different methods that can be used to manage load balancing, these are Static and Dynamic. As highlighted (Manekar et al., 2012), the dynamic approach adds a greater level of complexity, making the process more unpredictable and adding a greater overhead to the application. Although this approach, could potentially improve overall performance due to the workload being allocated at run time, I have opted for the static approach initially, so the workload is allocated within the code and is known at compile time. This will also allow easier debugging of the application and ensure stability during the simulation process. If time allows, I will attempt a dynamic approach, to potentially improve performance and reusability further.

Global Operations

The only section targeted for parallelisation, where global variables will become an issue, is a loop that is only relevant in one selection of user defined options, when the display mode is set to “ASCI Statistics Only. The variables, will be used to accumulate the current statistics, so they can be displayed accurately.

```
        livePrey = 0, livePred = 0, empty = 0;
        UpdateSimulation();
#pragma omp parallel num_threads(numThreads) shared (livePrey, livePred, empty)
        {
#pragma omp for reduction (+: livePrey, livePred, empty)
                for (int x = 0; x < width; x++) {
                        for (int y = 0; y < height; y++) {
                                if (mainGrid[x][y].type > 0) {
                                        livePrey++;
                                } else if (mainGrid[x][y].type < 0) {
                                        livePred++;
                                } else {
                                        empty++;
                                }
                        }
                }
        }
```

Fortunately, some of the variables, e.g. the Main Grid array, that are global, won’t be affected from the parallel process. This is because access to them is determined by the local variables within the loop, that determine the index of the array element. This ensures, that only one thread is accessing that shared element of the array, at any one iteration.

OpenMP Implementation

Algorithm / Strategy / Implementation

For the OpenMP implementation, there was no need to change the algorithm itself compared to the Serial implementation. Upon evaluation of the main class, there were five main sections of code that I felt would benefit from the OpenMP implementation.

The first was the Populate Grid method. Although this does not affect the simulation, as it is only ever called once per execution, I felt that this would speed up the initial start-up of the simulation, as this needs to be actioned before it can begin.

The next three targeted sections, are all within the main Update Simulation loop, indication of areas to be targeted for OpenMP can be seen in the comments below.

```
void Serial::UpdateSimulation() {
	/* Target for OpenMP */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// Prepare temporary grid
		}
	}
	/* Target for OpenMP */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// loop each neighbouring cell
				// check neighbouring cell states
			// Manage Prey
				// Check new state of cell
			// Manage Predator
				// Check new state of cell
			// Manage Empty
				// Check new state of cell
		}
	}
	/* Target for OpenMP */
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// Copy temporary grid back to main
		}
	}

}
```

The first section is to prepare the temporary grid, ready to store new information.

```
#pragma omp parallel num_threads(numThreads)
       {
#pragma omp for
              for (int x = 0; x < width; x++) {
                     for (int y = 0; y < height; y++) {
                            copyGrid[x][y].type = 0;
                            copyGrid[x][y].age = 0;
                     }
              }
       }
```

Evaluating the next loop for global and local variables, revealed only one issue, which was within the set random value operation. Initially this was inside the loop, but this presented undesirable results. Next, I tried to take this outside the declaration for OpenMP as a global declaration, this did not work, as this type of method cannot be shared. Finally, I added this, just after the OpenMP declaration, but just before the OpenMP for loop.

```
#pragma omp parallel num_threads(numThreads)
       {
              srand(time(NULL));
#pragma omp for
              for (int x = 0; x < width; x++) {
                     for (int y = 0; y < height; y++) {
                            // Evaluate neighbours

                     }
              }
       }
```

Other variables that are used within this section of code, are local and declared within the loop. Regardless, they needed to be reset throughout each iteration, so this presented no need for concern.

The final loop within this section of code, is to copy the temporary grid details back into the main grid. However, for peace of mind, an OpenMP Barrier, was included before this loop starts, ensuring all threads have completed the previous evaluation loop before starting.

```
#pragma omp barrier
#pragma omp parallel num_threads(numThreads)
       {
#pragma omp for
              for (int x = 0; x < width; x++) {
                     for (int y = 0; y < height; y++) {
                            mainGrid[x][y] = copyGrid[x][y];
                     }
              }
       }
}
```

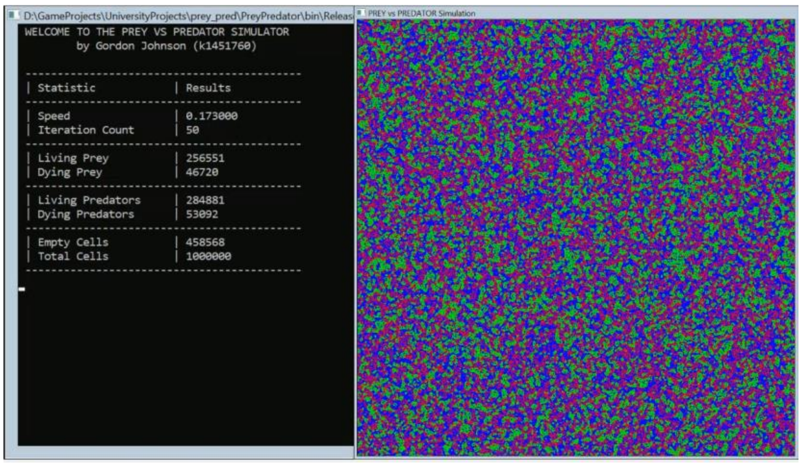
The final targeted loop, is only relevant in one situation of a user defined option, that is when the display mode is set to “ASCI Statistics Only. This section, demonstrates how global variables can be used and shared between threads, to provide an accumulated result.

```
	livePrey = 0, livePred = 0, empty = 0;
	UpdateSimulation();
#pragma omp parallel num_threads(numThreads) shared (livePrey, livePred, empty)
	{
#pragma omp for reduction (+: livePrey, livePred, empty)
		for (int x = 0; x < width; x++) {
			for (int y = 0; y < height; y++) {
				if (mainGrid[x][y].type > 0) {
					livePrey++;
				} else if (mainGrid[x][y].type < 0) {
					livePred++;
				} else {
					empty++;
				}
			}
		}
	}
```

Results

Screen shots, provided below, are taken at certain iteration counts, to demonstrate the evolving simulation.

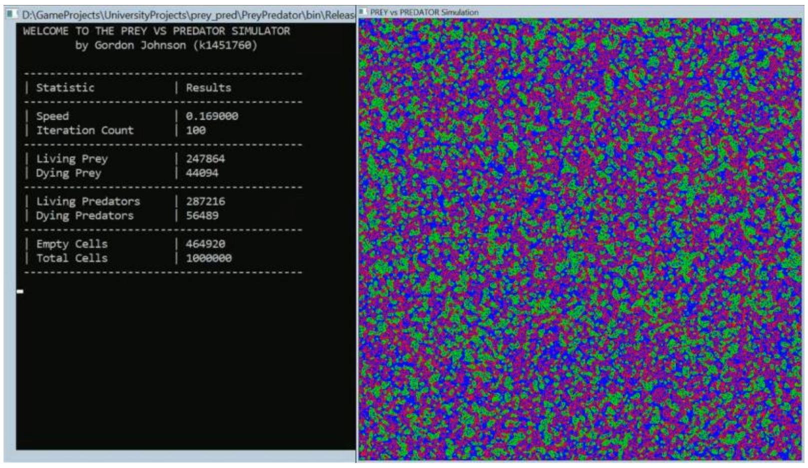
50 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.173000
| Iteration Count        | 50
---------------------------------------------
| Living Prey            | 256551
| Dying Prey             | 46720
---------------------------------------------
| Living Predators       | 284881
| Dying Predators        | 53092
---------------------------------------------
| Empty Cells            | 458568
| Total Cells            | 1000000
---------------------------------------------
```

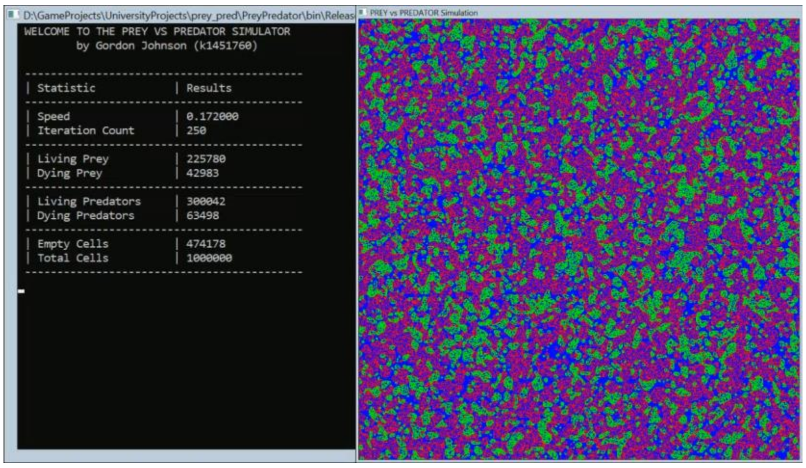
100 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.169000
| Iteration Count        | 100
---------------------------------------------
| Living Prey            | 247864
| Dying Prey             | 44094
---------------------------------------------
| Living Predators       | 287216
| Dying Predators        | 56489
---------------------------------------------
| Empty Cells            | 464920
| Total Cells            | 1000000
---------------------------------------------
```

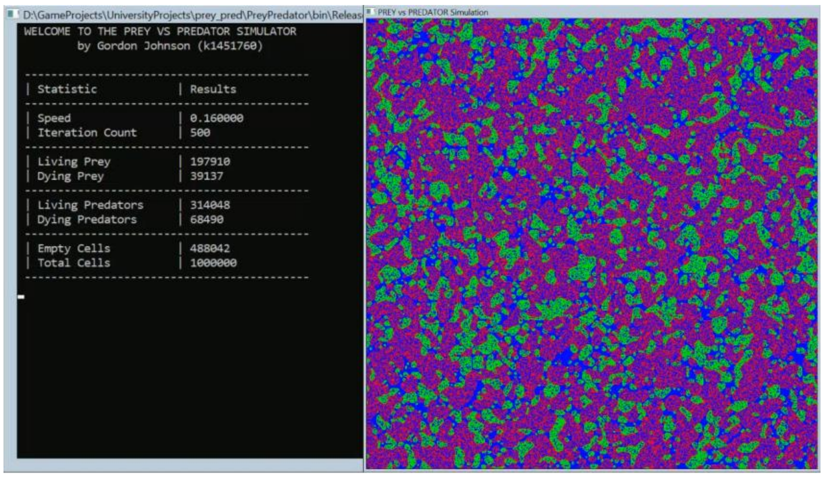
250 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.172000
| Iteration Count        | 250
---------------------------------------------
| Living Prey            | 225780
| Dying Prey             | 42983
---------------------------------------------
| Living Predators       | 300042
| Dying Predators        | 63498
---------------------------------------------
| Empty Cells            | 474178
| Total Cells            | 1000000
---------------------------------------------
```

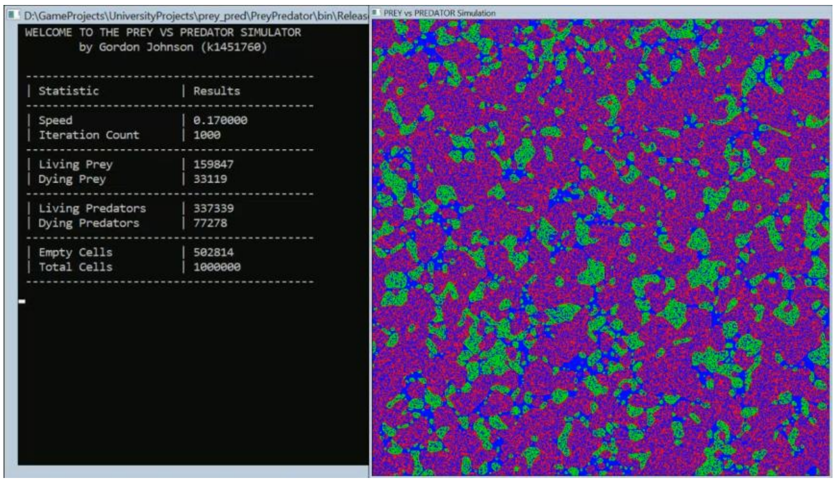
500 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

 ----------------------------------------
| Statistic            | Results
 ----------------------------------------
| Speed                | 0.160000
| Iteration Count      | 500
 ----------------------------------------
| Living Prey          | 197910
| Dying Prey           | 39137
 ----------------------------------------
| Living Predators     | 314048
| Dying Predators      | 68490
 ----------------------------------------
| Empty Cells          | 488042
| Total Cells          | 1000000
 ----------------------------------------
```

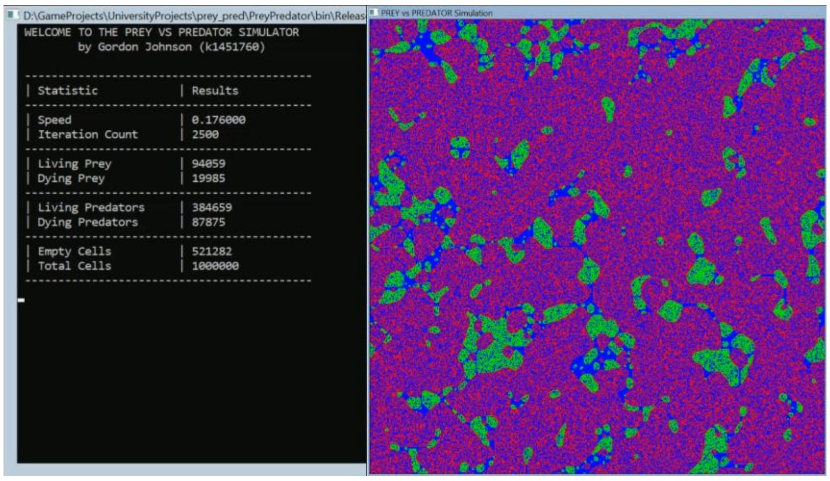
1000 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

 ----------------------------------------
| Statistic            | Results
 ----------------------------------------
| Speed                | 0.170000
| Iteration Count      | 1000
 ----------------------------------------
| Living Prey          | 159847
| Dying Prey           | 33119
 ----------------------------------------
| Living Predators     | 337339
| Dying Predators      | 77278
 ----------------------------------------
| Empty Cells          | 502814
| Total Cells          | 1000000
 ----------------------------------------
```

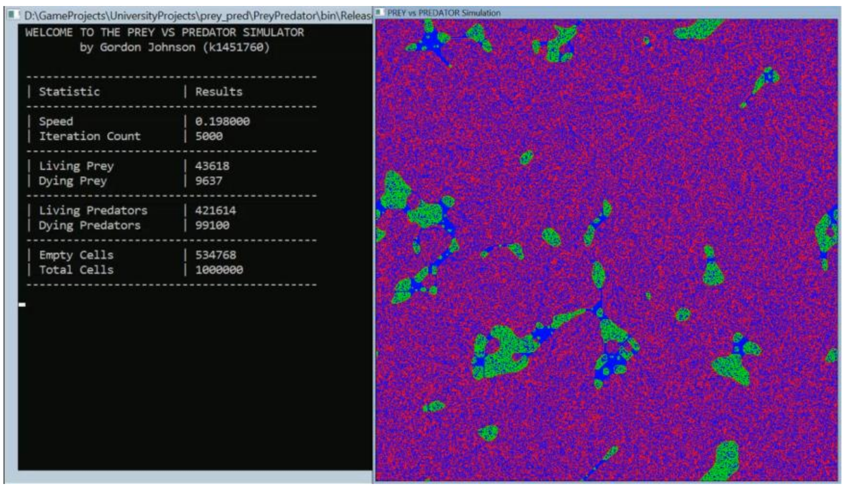
2500 iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

 ----------------------------------------
| Statistic            | Results
 ----------------------------------------
| Speed                | 0.176000
| Iteration Count      | 2500
 ----------------------------------------
| Living Prey          | 94059
| Dying Prey           | 19985
 ----------------------------------------
| Living Predators     | 384659
| Dying Predators      | 87875
 ----------------------------------------
| Empty Cells          | 521282
| Total Cells          | 1000000
 ----------------------------------------
```

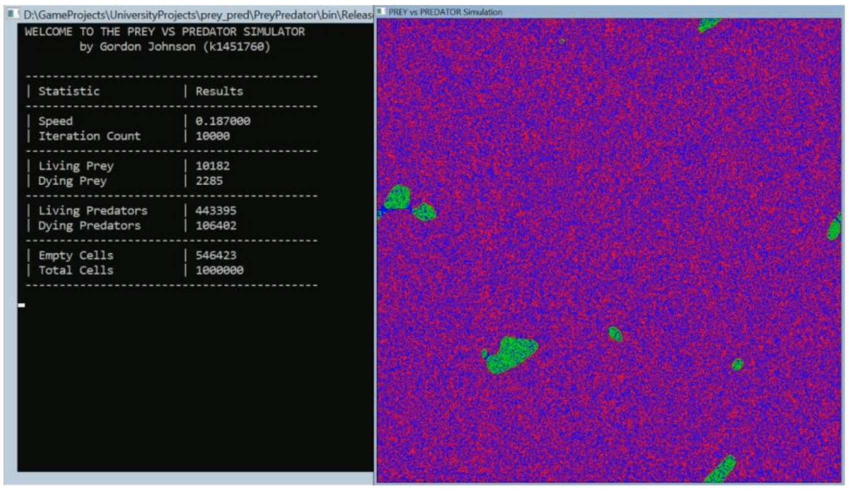
5000 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.198000
| Iteration Count        | 5000
---------------------------------------------
| Living Prey            | 43618
| Dying Prey             | 9637
---------------------------------------------
| Living Predators       | 421614
| Dying Predators        | 99100
---------------------------------------------
| Empty Cells            | 534768
| Total Cells            | 1000000
---------------------------------------------
```

10000 Iterations


D:\GameProjects\UniversityProjects\prey_pred\PreyPredator\bin\Releas
PREY vs PREDATOR Simulation

```
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

---------------------------------------------
| Statistic              | Results
---------------------------------------------
| Speed                  | 0.187000
| Iteration Count        | 10000
---------------------------------------------
| Living Prey            | 10182
| Dying Prey             | 2285
---------------------------------------------
| Living Predators       | 443395
| Dying Predators        | 106402
---------------------------------------------
| Empty Cells            | 546423
| Total Cells            | 1000000
---------------------------------------------
```

Performance

For the OpenMP implementation, the following tests were conducted, as presented in the introduction of this report. As a baseline comparison, the previous Serial Implementation results, are below.

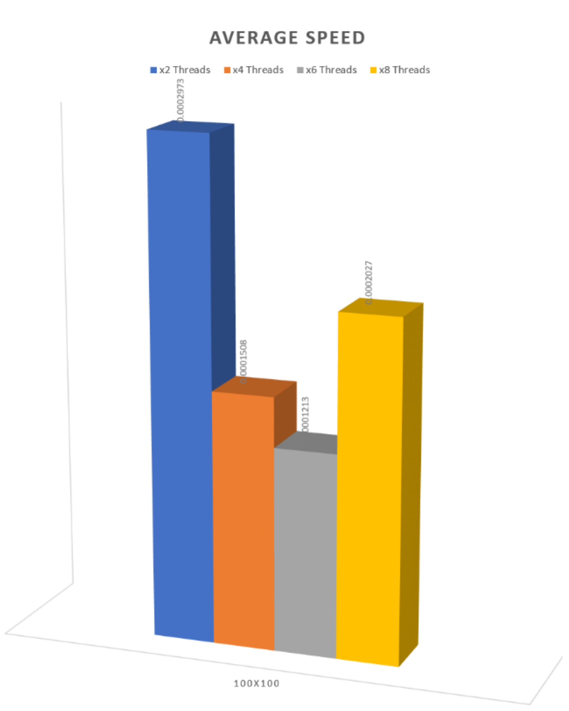
OpenMP Implementation

Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
2x Threads	0.0002973	0.0493534	4.547463
4x Threads	0.0001508	0.0295983	2.490935
6x Threads	0.0001213	0.0229262	2.008494
8x Threads	0.0002027	0.0187689	1.723995

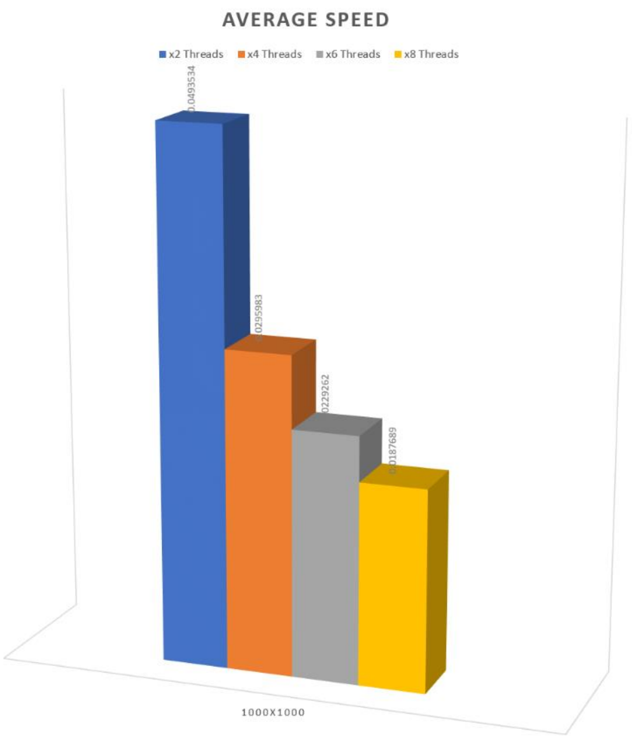
Serial Implementation Results (Baseline)

Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
Average Speed per Iteration	0.0005381	0.0951833	8.722674

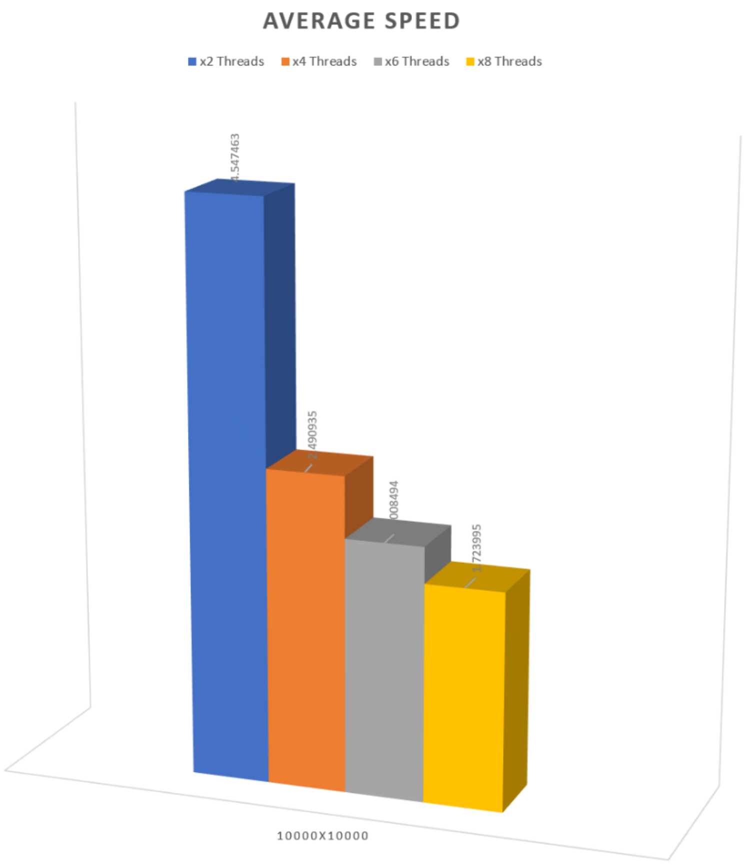
As can be seen by these results, using just 2x Threads within the OpenMP implementation, provides almost double the speed to the application. Providing the simulation with x4 threads, provides the largest increase in acceleration, compared to all other options. Providing x6 and x8, both provided minor improvements respectively, but still show better performance than the previous.



100x100 Grid Size (Average Speed)

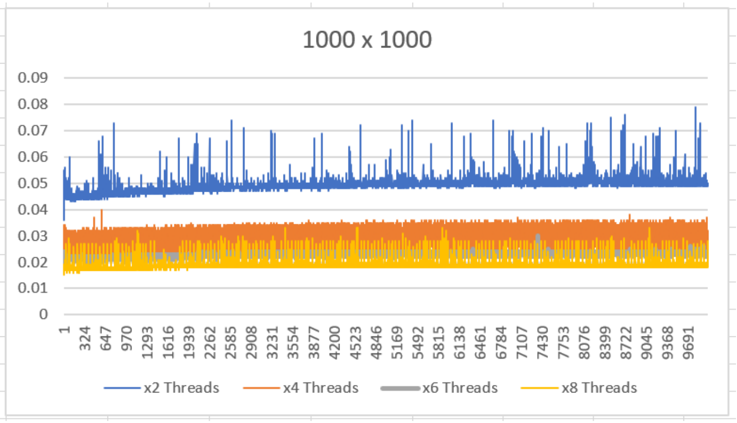


1000x1000 Grid Size (Average Speed)

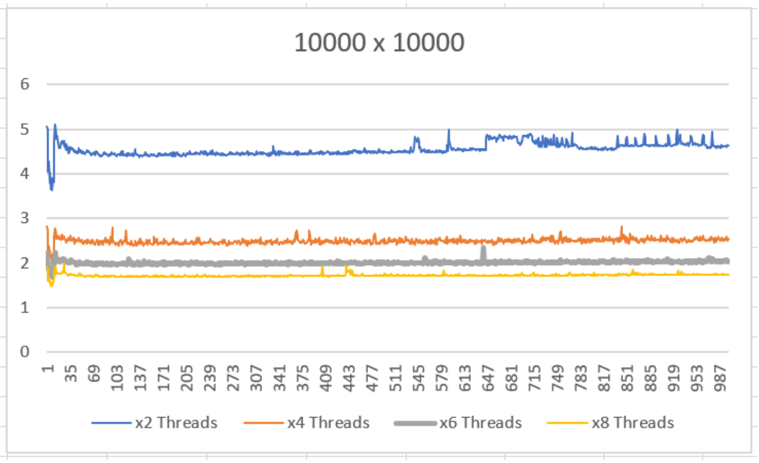


10000x10000 Grid Size (Average Speed)

The following graphs, show performance over time.



1000x1000 Grid Size (Performance per Iteration)



10000x10000 Grid Size (Performance per Iteration)

As can be seen from these charts, taken from the raw data, there is a gradual increase in seconds taken, indicative of a decline in performance over time

Also noted, that the 100x100 grid with x8 threads, shows a decrease in performance. This could have been an issue with processor usage at the time of testing and does not reflect in the other tests with this amount of threads.

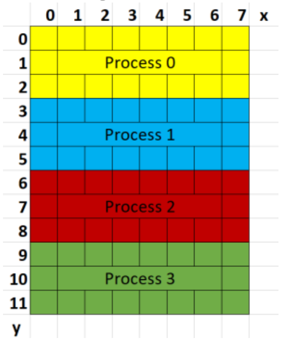
MPI Implementation

Algorithm / Strategy / Implementation

Due to the original design of implementing the project into a single application, as opposed to four, MPI needed to be implemented regardless, at the application point of entry. The main function of this, will need to be executed, regardless of whether MPI is being used.

```
int main(int argc, char *argv[]) {
	MPI_Init(&argc, &argv);
	InfoMPI* info = new InfoMPI;
	MPI_Comm_size(MPI_COMM_WORLD, &info->numProcs);
	MPI_Comm_rank(MPI_COMM_WORLD, &info->rank);
	App* app = new App(*info);
	delete app;
	delete info;
	MPI_Finalize();
	system("PAUSE");
	return 0;
}
```

Extra consideration needed to be taken throughout the setup process, to ensure the master process took care of the user selection process. Whilst at the same time, ensuring the selection is passed to all other processes.



In the main simulation, it was important to decide early, how to best split the data being processed across two or more processes. The main grid, needed to be split either in the x or y direction. For this, I chose to split across the y (height) value, as seen in the diagram to the right. Process 0, will take on the role of the master process and will process everything outside of the main simulation, which includes managing the statistic and graphics to be displayed. An extra check, is needed before the simulation can start, this is to determine if the height value will evenly split between the selected number of processes. If this is not the case, then the value will be rounded down, to the next even split value.

The first method, to manage the split of the grid, is to populate the grid with its initial states. For this, the master process takes care of the initial declaration of states, depending on which process region y is currently at, it will send that information to the appropriate processors grid.

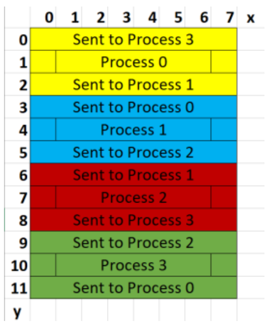
```
void MsMPI::PopulateGrid() {
	const int contributionY = abs(height / info.numProcs);
	int processorCounter = 1;
	for (int x = 0; x < width; x++) {
		for (int y = 0; y < height; y++) {
			// Only the master manages the initial declaration
			// of each cell state
			if (info.rank == 0) {
				// allocate random cell state
			}
			// determine if the y value has exceeded the current
			// processor split
			if (y >= contributionY * (processorCounter + 1)) {
				if (processorCounter != info.numProcs - 1) {
					processorCounter++;
				}
			}
			// If within the current processor copy the master cell
			// to the appropriate processor
			if (y >= contributionY * processorCounter &&
				y < contributionY * (processorCounter + 1)) {
				if (info.rank == 0) {
					MPI_Send(&mainGrid[x][y], 2, MPI_INT,
						processorCounter, y, MPI_COMM_WORLD);
				}
				if (info.rank == processorCounter) {
					MPI_Recv(&mainGrid[x][y], 2, MPI_INT,
						0, y, MPI_COMM_WORLD, &status);
				}
			}
		}
		// at the end of each y iteration reset processor to 1
		processorCounter = 1;
	}
```

Once complete, the master process will store all the initial states, but going forward will only mange its own starting section. For example, if the height of the grid was 100 and there are 4 active processes, then the management of the y value would be as follows

- Process 0 – (0 - 24)
- Process 1 – (25 - 49)
- Process 2 – (50 - 74)
- Process 3 – (75 - 99)

The next step, was to ensure that each process contains the latest neighbouring states. As each process is only managing a split of section, an addition loop is implemented, to copy the missing information to the relevant process (see image on right).

Including this, additional two rows of information in each process, will eliminate the need to apply any amendments, when it comes to checking the neighbouring states.



y \ x	0 1 2 3 4 5 6 7
0	Sent to Process 3
1	Process 0
2	Sent to Process 1
3	Sent to Process 0
4	Process 1
5	Sent to Process 2
6	Sent to Process 1
7	Process 2
8	Sent to Process 3
9	Sent to Process 2
10	Process 3
11	Sent to Process 0

```
for (int x = 0; x < width; x++) {
        if (info.rank == 0) {
                MPI_Send(&mainGrid[x][0], 2, MPI_INT,
                        info.numProcs - 1, x, MPI_COMM_WORLD);
                MPI_Recv(&mainGrid[x][height - 1], 2, MPI_INT,
                        info.numProcs - 1, x + width, MPI_COMM_WORLD, &status);
        }
        else if (info.rank == info.numProcs - 1) {
                MPI_Send(&mainGrid[x][height - 1], 2, MPI_INT,
                        0, x + width, MPI_COMM_WORLD);
                MPI_Recv(&mainGrid[x][0], 2, MPI_INT,
                        0, x, MPI_COMM_WORLD, &status);
        }
        if (info.rank != info.numProcs - 1) {
                MPI_Send(&mainGrid[x][(contributionY * (info.rank + 1)) - 1], 2,
                        MPI_INT, info.rank + 1, x, MPI_COMM_WORLD);
                MPI_Recv(&mainGrid[x][(contributionY * (info.rank + 1))], 2,
                        MPI_INT, info.rank + 1, x, MPI_COMM_WORLD, &status);
        }
        if (info.rank != 0) {
                MPI_Send(&mainGrid[x][contributionY * info.rank], 2,
                        MPI_INT, info.rank - 1, x, MPI_COMM_WORLD);
                MPI_Recv(&mainGrid[x][(contributionY * info.rank) - 1], 2,
                        MPI_INT, info.rank - 1, x, MPI_COMM_WORLD, &status);
        }
}
```

Before going into the main update loop, a barrier is applied to ensure all processes have sent their updated values, to the relevant other process. The main update loop itself, remains the same as the other implementations, with the exception that the for loop, only considers the y range related to the process that is going through.

```
MPI_Barrier(MPI_COMM_WORLD);
for (int x = 0; x < width; x++) {
	for (int y = contributionY * info.rank;
		y < contributionY * (info.rank + 1); y++) {
		// loop each neighbouring cell
			// check neighbouring cell states
		// Manage Prey
			// Check new state of cell
		// Manage Predator
			// Check new state of cell
		// Manage Empty
			// Check new state of cell
	}
}
```

The new states, are then copied to the main grid, taking into consideration each process has a unique y range.

```
for (int x = 0; x < width; x++) {
	for (int y = contributionY * info.rank;
	y < contributionY * (info.rank + 1); y++) {
		mainGrid[x][y] = copyGrid[x][y];
	}
}
```

An addition loop, was created and only used when the user option is set to display results, either ASCI or Graphical. All non-master processes, send their grid section to the master process, so the information can be drawn complete.

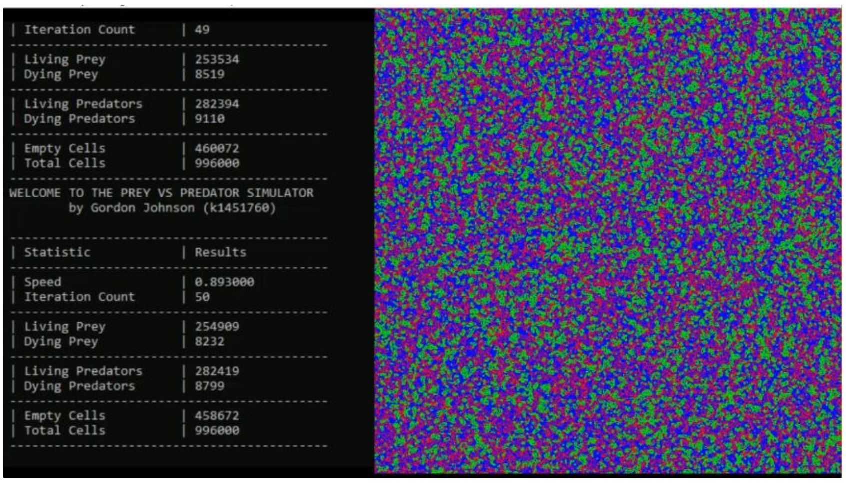
```
int processorCounter = info.numProcs - 1;
while (processorCounter != 0) {
       for (int x = 0; x < width; x++) {
              for (int y = contributionY * processorCounter; y < height; y++) {
                     if (info.rank == processorCounter) {
                            MPI_Rsend(&mainGrid[x][y], 2, MPI_INT,
                                   processorCounter - 1, y * (x + processorCounter),
                                   MPI_COMM_WORLD);
                     }
                     if (info.rank == processorCounter - 1) {
                            MPI_Recv(&mainGrid[x][y], 2, MPI_INT,
                                   processorCounter, y * (x + processorCounter),
                                   MPI_COMM_WORLD, &status);
                     }
              }
       }
       processorCounter--;
}
```

However, this loop contains a race condition. Although performance in this loop, has been improved considerably since, the condition persists. This does not affect the performance test results, as the tests are run with no draw attribute needed, so this loop is excluded.

Results

Screen shots provided below, are taken at certain iteration counts, to demonstrate the evolving simulation. Note: when using MPI, the system clear screen function no longer works, hence the statistic being displayed unusually.

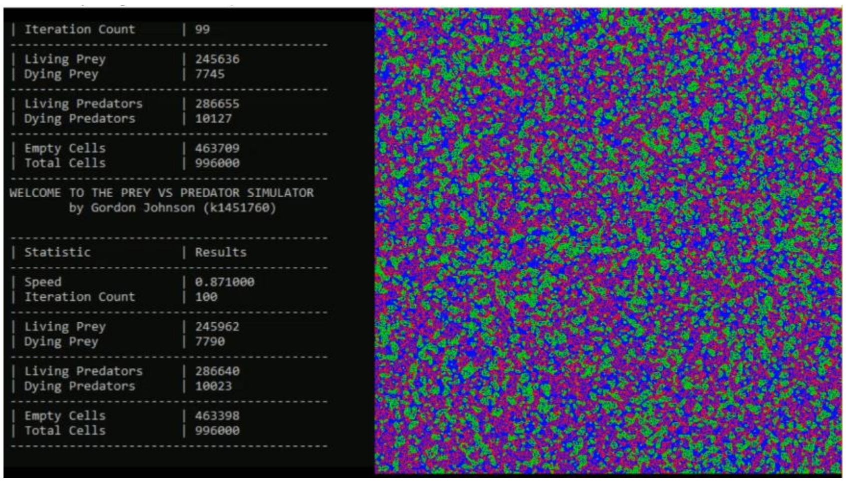
50 Iterations



```
| Iteration Count      | 49
-------------------------------------------
| Living Prey          | 253534
| Dying Prey           | 8519
-------------------------------------------
| Living Predators     | 282394
| Dying Predators      | 9110
-------------------------------------------
| Empty Cells          | 460072
| Total Cells          | 996000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic            | Results
-------------------------------------------
| Speed                | 0.893000
| Iteration Count      | 50
-------------------------------------------
| Living Prey          | 254909
| Dying Prey           | 8232
-------------------------------------------
| Living Predators     | 282419
| Dying Predators      | 8799
-------------------------------------------
| Empty Cells          | 458672
| Total Cells          | 996000
-------------------------------------------
```

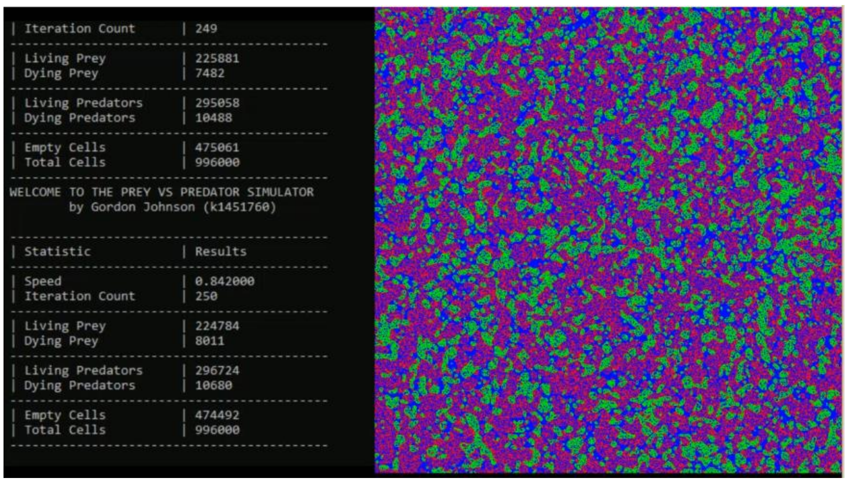
100 Iterations



```
| Iteration Count      | 99
-------------------------------------------
| Living Prey          | 245636
| Dying Prey           | 7745
-------------------------------------------
| Living Predators     | 286655
| Dying Predators      | 10127
-------------------------------------------
| Empty Cells          | 463709
| Total Cells          | 996000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic            | Results
-------------------------------------------
| Speed                | 0.871000
| Iteration Count      | 100
-------------------------------------------
| Living Prey          | 245962
| Dying Prey           | 7790
-------------------------------------------
| Living Predators     | 286640
| Dying Predators      | 10023
-------------------------------------------
| Empty Cells          | 463398
| Total Cells          | 996000
-------------------------------------------
```

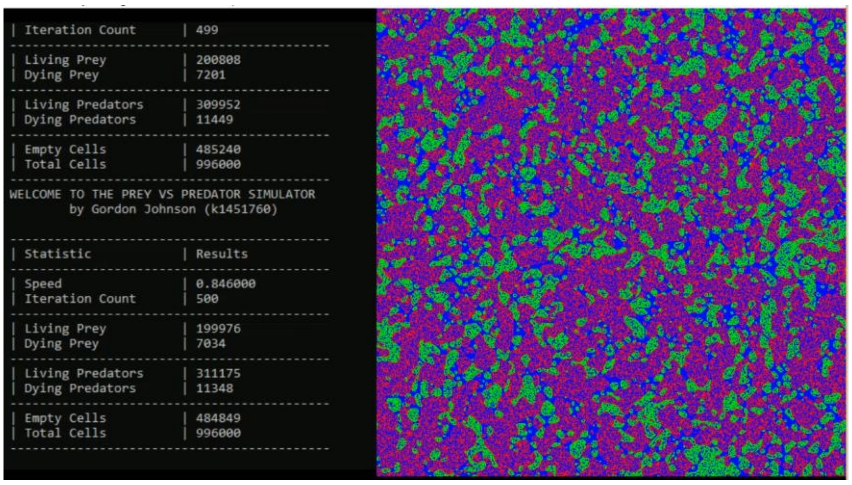
250 Iterations



```
| Iteration Count      | 249
-------------------------------------------
| Living Prey          | 225881
| Dying Prey           | 7482
-------------------------------------------
| Living Predators     | 295058
| Dying Predators      | 10488
-------------------------------------------
| Empty Cells          | 475061
| Total Cells          | 996000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic            | Results
-------------------------------------------
| Speed                | 0.842000
| Iteration Count      | 250
-------------------------------------------
| Living Prey          | 224784
| Dying Prey           | 8011
-------------------------------------------
| Living Predators     | 296724
| Dying Predators      | 10680
-------------------------------------------
| Empty Cells          | 474492
| Total Cells          | 996000
-------------------------------------------
```

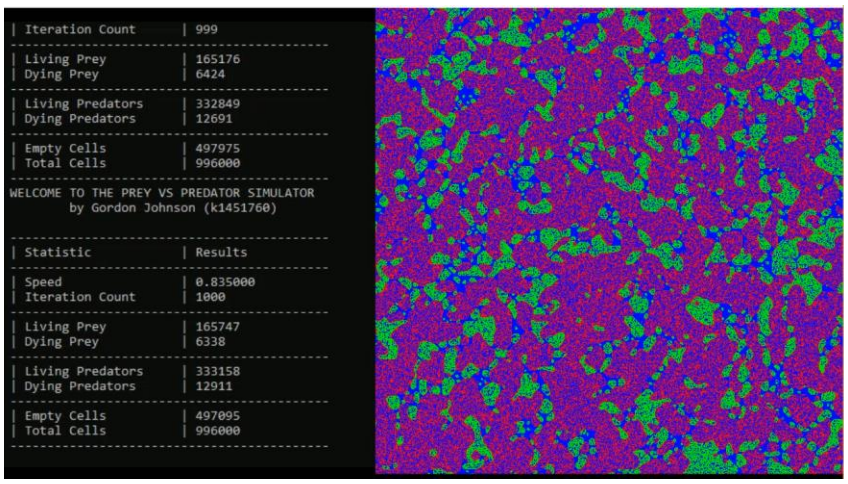
500 Iterations



```
| Iteration Count      | 499
--------------------------------------------
| Living Prey          | 200808
| Dying Prey           | 7201
--------------------------------------------
| Living Predators     | 309952
| Dying Predators      | 11449
--------------------------------------------
| Empty Cells          | 485240
| Total Cells          | 996000
--------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

--------------------------------------------
| Statistic            | Results
--------------------------------------------
| Speed                | 0.846000
| Iteration Count      | 500
--------------------------------------------
| Living Prey          | 199976
| Dying Prey           | 7034
--------------------------------------------
| Living Predators     | 311175
| Dying Predators      | 11348
--------------------------------------------
| Empty Cells          | 484849
| Total Cells          | 996000
--------------------------------------------
```

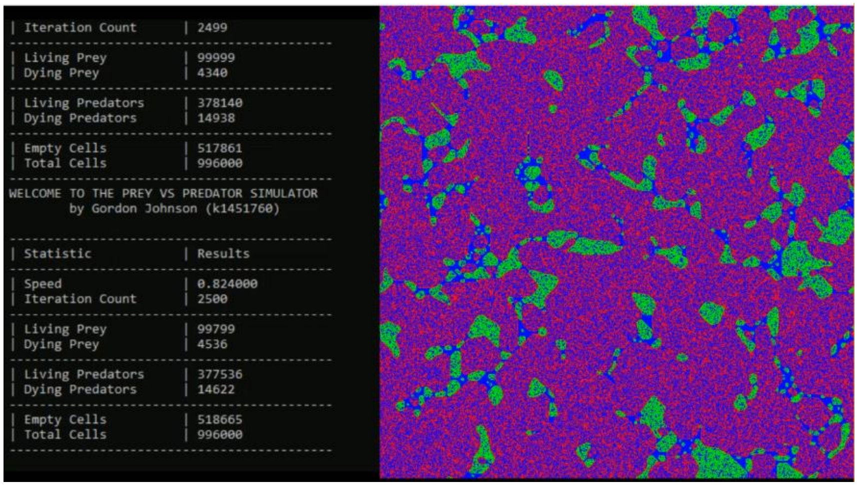
1000 Iterations



```
| Iteration Count      | 999
--------------------------------------------
| Living Prey          | 165176
| Dying Prey           | 6424
--------------------------------------------
| Living Predators     | 332849
| Dying Predators      | 12691
--------------------------------------------
| Empty Cells          | 497975
| Total Cells          | 996000
--------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

--------------------------------------------
| Statistic            | Results
--------------------------------------------
| Speed                | 0.835000
| Iteration Count      | 1000
--------------------------------------------
| Living Prey          | 165747
| Dying Prey           | 6338
--------------------------------------------
| Living Predators     | 333158
| Dying Predators      | 12911
--------------------------------------------
| Empty Cells          | 497095
| Total Cells          | 996000
--------------------------------------------
```

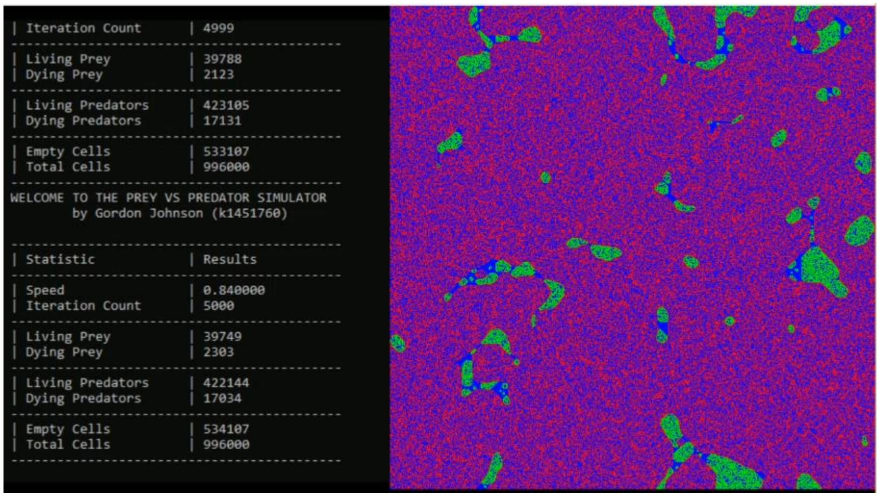
2500 iterations



```
| Iteration Count      | 2499
--------------------------------------------
| Living Prey          | 99999
| Dying Prey           | 4340
--------------------------------------------
| Living Predators     | 378140
| Dying Predators      | 14938
--------------------------------------------
| Empty Cells          | 517861
| Total Cells          | 996000
--------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

--------------------------------------------
| Statistic            | Results
--------------------------------------------
| Speed                | 0.824000
| Iteration Count      | 2500
--------------------------------------------
| Living Prey          | 99799
| Dying Prey           | 4536
--------------------------------------------
| Living Predators     | 377536
| Dying Predators      | 14622
--------------------------------------------
| Empty Cells          | 518665
| Total Cells          | 996000
--------------------------------------------
```

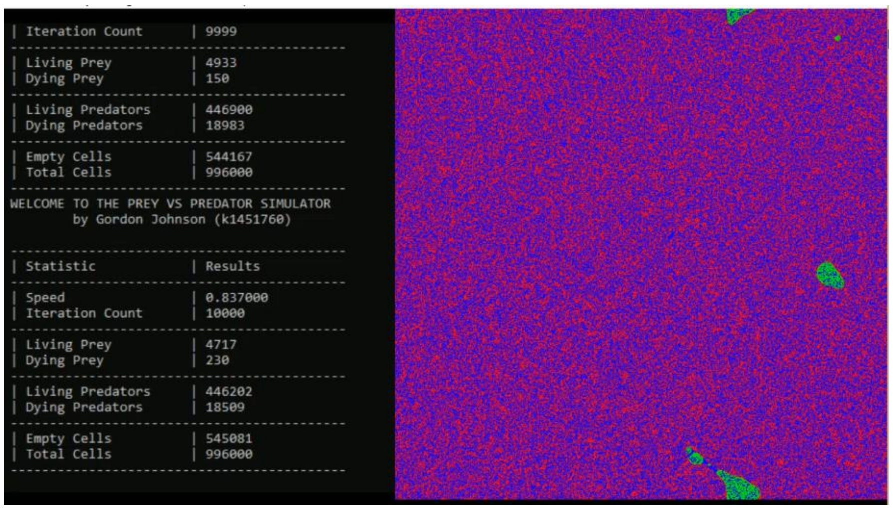
5000 Iterations



```
| Iteration Count      | 4999
-------------------------------------------
| Living Prey          | 39788
| Dying Prey           | 2123
-------------------------------------------
| Living Predators     | 423105
| Dying Predators      | 17131
-------------------------------------------
| Empty Cells          | 533107
| Total Cells          | 996000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic            | Results
-------------------------------------------
| Speed                | 0.840000
| Iteration Count      | 5000
-------------------------------------------
| Living Prey          | 39749
| Dying Prey           | 2303
-------------------------------------------
| Living Predators     | 422144
| Dying Predators      | 17034
-------------------------------------------
| Empty Cells          | 534107
| Total Cells          | 996000
-------------------------------------------
```

10000 Iterations



```
| Iteration Count      | 9999
-------------------------------------------
| Living Prey          | 4933
| Dying Prey           | 150
-------------------------------------------
| Living Predators     | 446900
| Dying Predators      | 18983
-------------------------------------------
| Empty Cells          | 544167
| Total Cells          | 996000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic            | Results
-------------------------------------------
| Speed                | 0.837000
| Iteration Count      | 10000
-------------------------------------------
| Living Prey          | 4717
| Dying Prey           | 230
-------------------------------------------
| Living Predators     | 446202
| Dying Predators      | 18509
-------------------------------------------
| Empty Cells          | 545081
| Total Cells          | 996000
-------------------------------------------
```

Performance

For the MPI implementation, the following tests were conducted, as presented in the introduction of this report. Below this, is included the previous Serial Implementation results, as a baseline comparison.

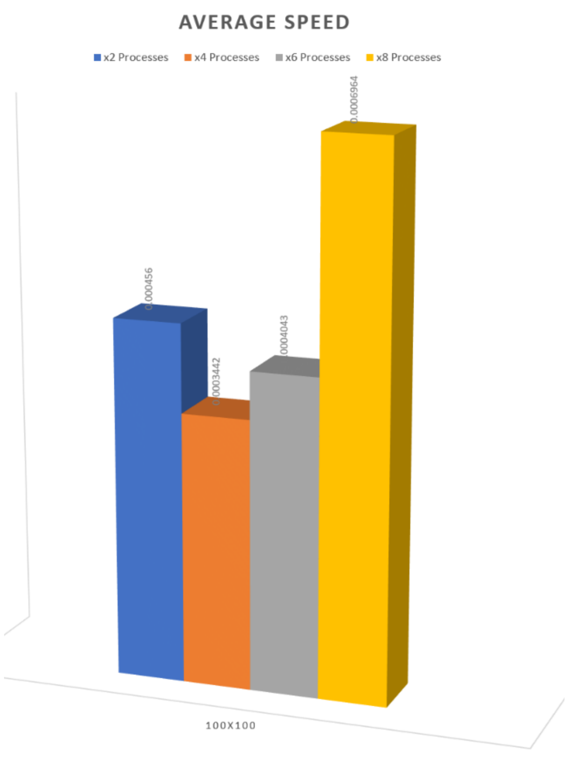
MPI Implementation

Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
2x Processes	0.0004560	0.0516448	4.746244
4x Processes	0.0003442	0.0336481	2.949087
6x Processes	0.0004043	0.0302868	2.706252
8x Processes	0.0006964	0.0322505	2.743532

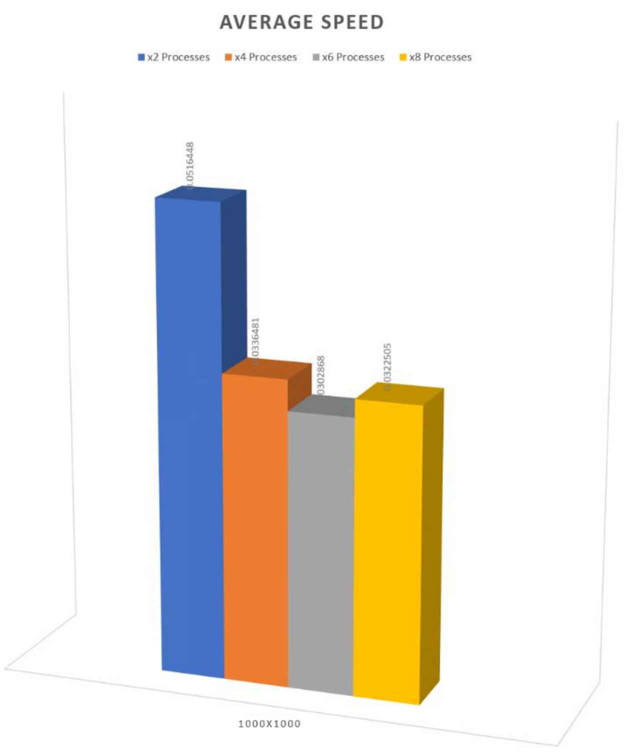
Serial Implementation Results (Baseline)

Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
Average Speed per Iteration	0.0005381	0.0951833	8.722674

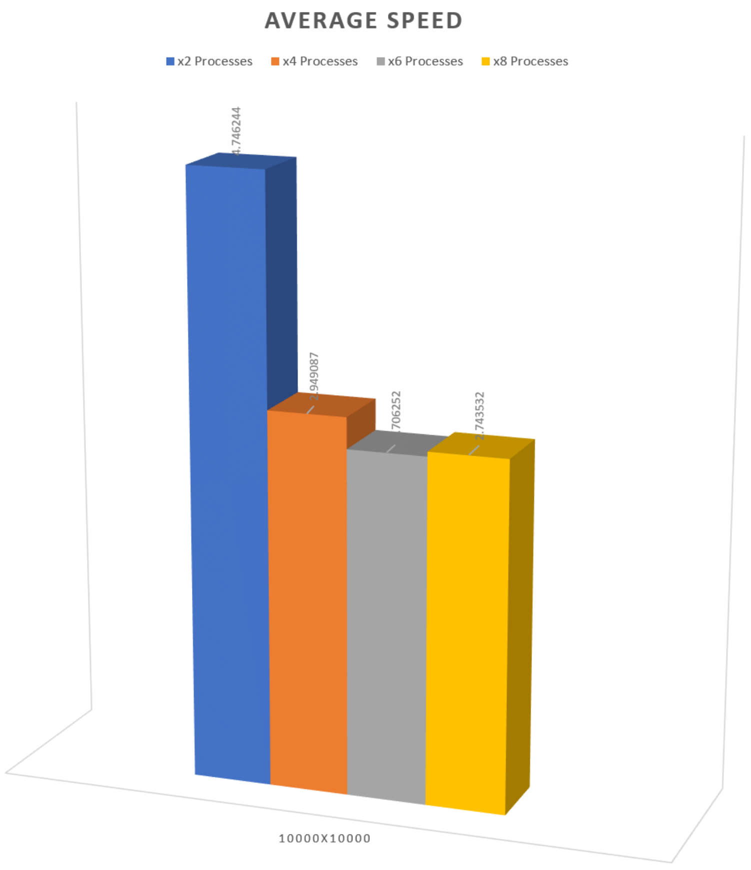
Note: Using 8x processes on the 100x100 grid, is considerably slower than its predecessor and falls below the baseline value. This could have been an issue at testing time and it does not drop below the baseline for any other tests.



100x100 Grid Size (Average Speed)

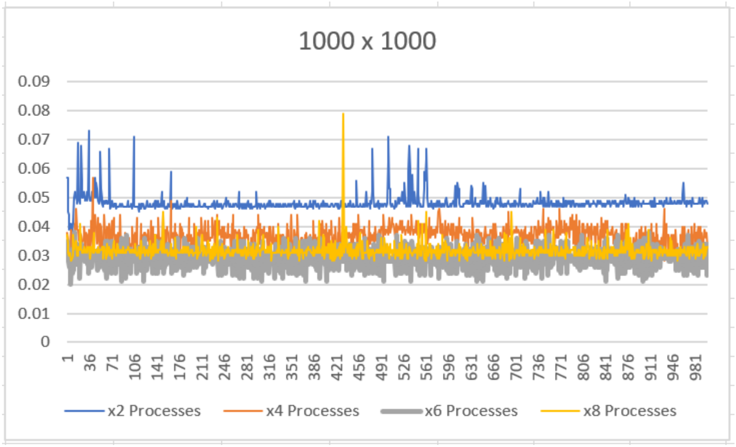


1000x1000 Grid Size (Average Speed)

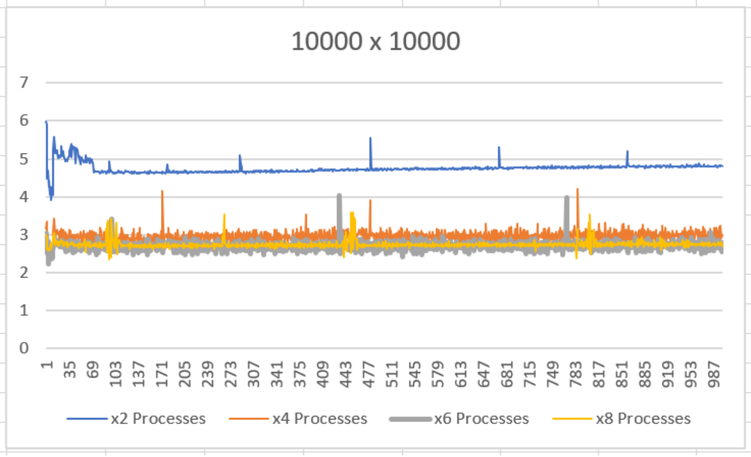


10000x10000 Grid Size (Average Speed)

Like the OpenMP implementation, this MPI version, provides almost double the speed to the application, compared to the serial version by using 2x processes. Providing the simulation with x4 processes, provides the largest increase in acceleration, compared to all other options. Providing x6 and x8, both displayed minor improvements respectively, but still show better performance than previously. The following graphs, show performance over time.



1000x1000 Grid Size (Performance per Iteration)



10000x10000 Grid Size (Performance per Iteration)

As can be seen from these charts, taken from the raw data, there is a gradual increase in seconds taken, indicative of a decline in performance over time

The 100x100 grid with x8 threads, also shows a decrease in performance. This could have been an issue with processor usage at the time of testing and does not reflect in other tests, with this amount of threads.

Hybrid Implementation

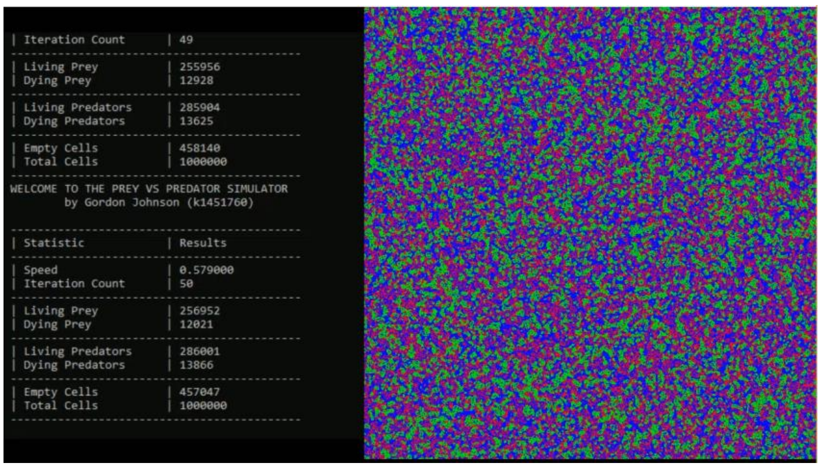
Algorithm / Strategy / Implementation

From the MPI implementation, there is no need to change any code for the Hybrid version, including OpenMP which was implemented into the previous targeted locations.

Results

Screen shots provided below are taken at certain iteration counts, to demonstrate the evolving simulation. Note: when using MPI / Hybrid, the system clear screen function no longer works, hence the statistic being displayed unusually.

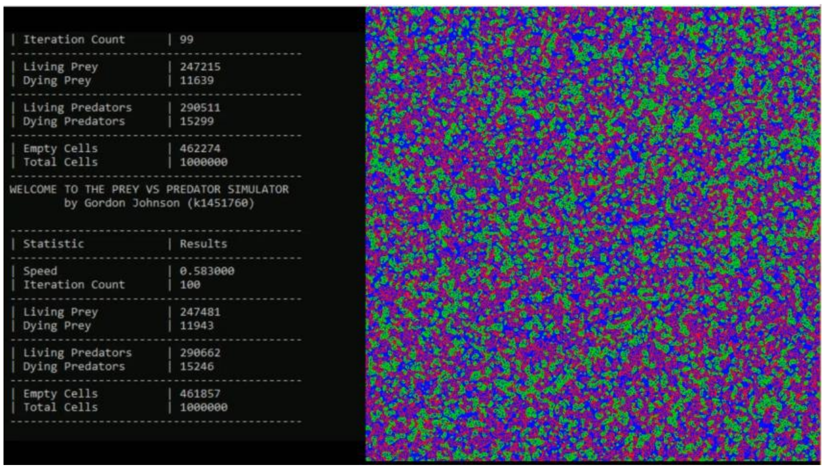
50 Iterations



```
| Iteration Count       | 49
-------------------------------------------
| Living Prey           | 255956
| Dying Prey            | 12928
-------------------------------------------
| Living Predators      | 285904
| Dying Predators       | 13625
-------------------------------------------
| Empty Cells           | 458140
| Total Cells           | 1000000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.579000
| Iteration Count       | 50
-------------------------------------------
| Living Prey           | 256952
| Dying Prey            | 12021
-------------------------------------------
| Living Predators      | 286001
| Dying Predators       | 13866
-------------------------------------------
| Empty Cells           | 457047
| Total Cells           | 1000000
-------------------------------------------
```

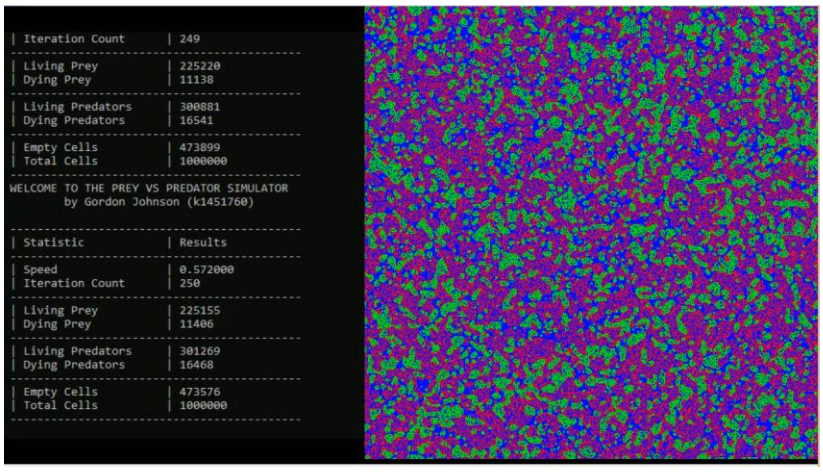
100 Iterations



```
| Iteration Count       | 99
-------------------------------------------
| Living Prey           | 247215
| Dying Prey            | 11639
-------------------------------------------
| Living Predators      | 290511
| Dying Predators       | 15299
-------------------------------------------
| Empty Cells           | 462274
| Total Cells           | 1000000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.583000
| Iteration Count       | 100
-------------------------------------------
| Living Prey           | 247481
| Dying Prey            | 11943
-------------------------------------------
| Living Predators      | 290662
| Dying Predators       | 15246
-------------------------------------------
| Empty Cells           | 461857
| Total Cells           | 1000000
-------------------------------------------
```

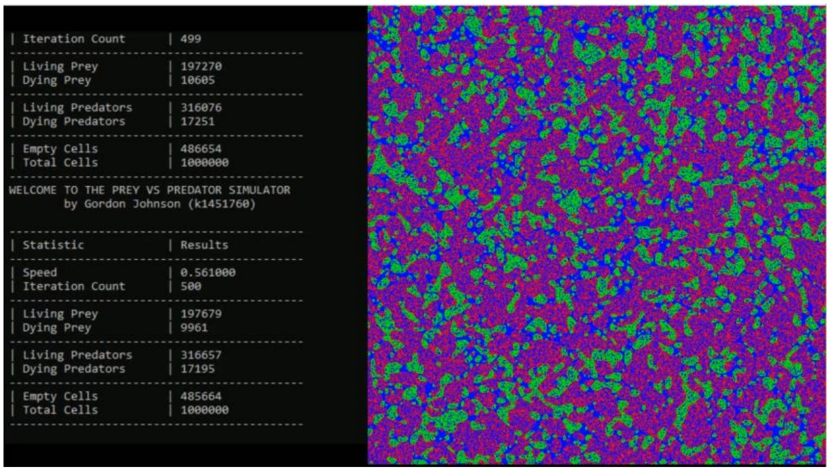
250 Iterations



```
| Iteration Count       | 249
-------------------------------------------
| Living Prey           | 225220
| Dying Prey            | 11138
-------------------------------------------
| Living Predators      | 300881
| Dying Predators       | 16541
-------------------------------------------
| Empty Cells           | 473899
| Total Cells           | 1000000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.572000
| Iteration Count       | 250
-------------------------------------------
| Living Prey           | 225155
| Dying Prey            | 11406
-------------------------------------------
| Living Predators      | 301269
| Dying Predators       | 16468
-------------------------------------------
| Empty Cells           | 473576
| Total Cells           | 1000000
-------------------------------------------
```

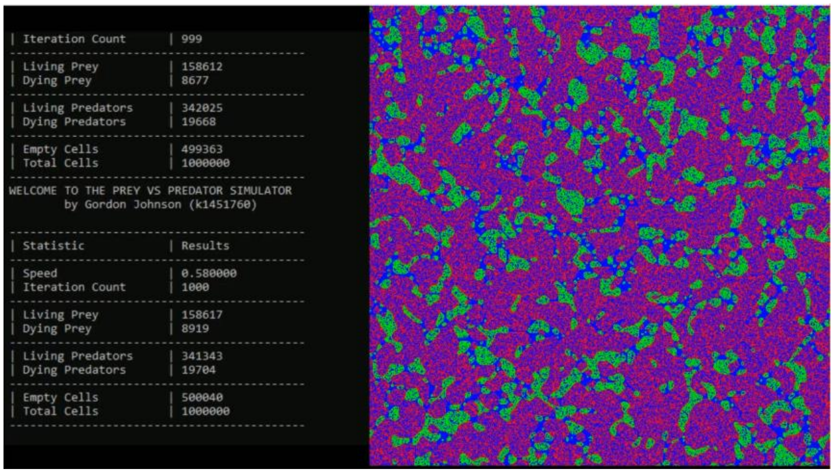
500 Iterations



```
| Iteration Count       | 499
-------------------------------------------
| Living Prey           | 197270
| Dying Prey            | 10605
-------------------------------------------
| Living Predators      | 316076
| Dying Predators       | 17251
-------------------------------------------
| Empty Cells           | 486654
| Total Cells           | 1000000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.561000
| Iteration Count       | 500
-------------------------------------------
| Living Prey           | 197679
| Dying Prey            | 9961
-------------------------------------------
| Living Predators      | 316657
| Dying Predators       | 17195
-------------------------------------------
| Empty Cells           | 485664
| Total Cells           | 1000000
-------------------------------------------
```

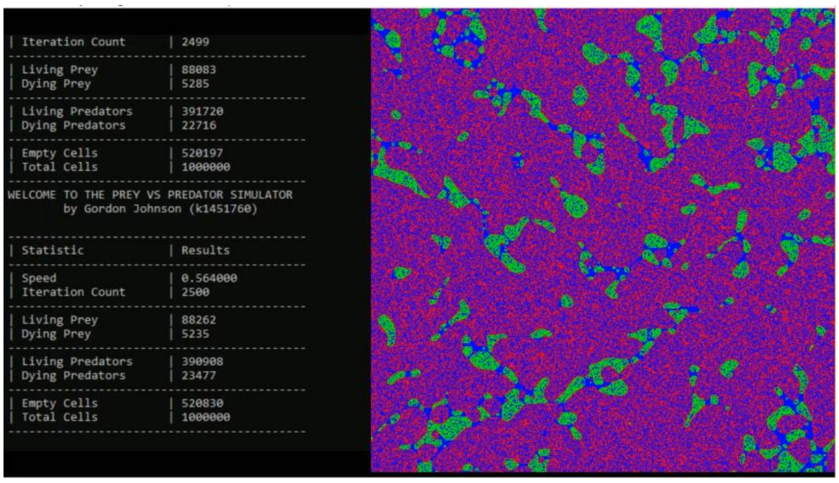
1000 Iterations



```
| Iteration Count       | 999
-------------------------------------------
| Living Prey           | 158612
| Dying Prey            | 8677
-------------------------------------------
| Living Predators      | 342025
| Dying Predators       | 19668
-------------------------------------------
| Empty Cells           | 499363
| Total Cells           | 1000000
-------------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-------------------------------------------
| Statistic             | Results
-------------------------------------------
| Speed                 | 0.580000
| Iteration Count       | 1000
-------------------------------------------
| Living Prey           | 158617
| Dying Prey            | 8919
-------------------------------------------
| Living Predators      | 341343
| Dying Predators       | 19704
-------------------------------------------
| Empty Cells           | 500040
| Total Cells           | 1000000
-------------------------------------------
```

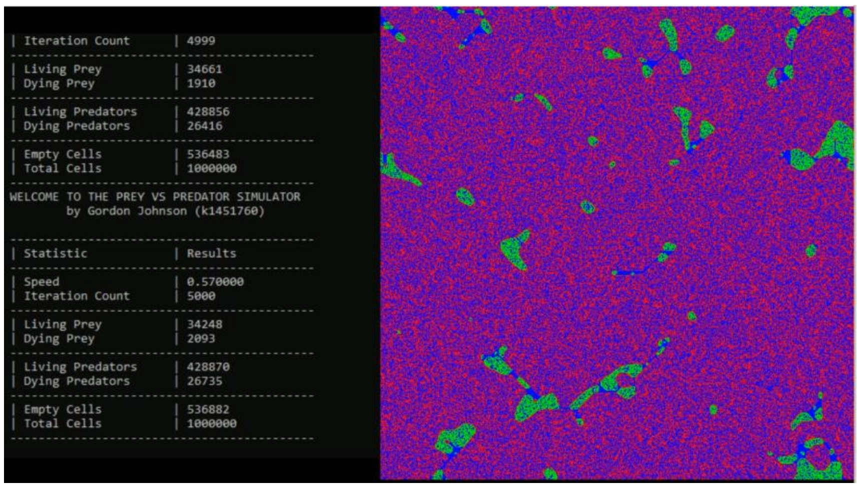
2500 iterations



```
| Iteration Count          | 2499
-----------------------------------------
| Living Prey              | 88083
| Dying Prey               | 5285
-----------------------------------------
| Living Predators         | 391720
| Dying Predators          | 22716
-----------------------------------------
| Empty Cells              | 520197
| Total Cells              | 1000000
-----------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-----------------------------------------
| Statistic                | Results
-----------------------------------------
| Speed                    | 0.564000
| Iteration Count          | 2500
-----------------------------------------
| Living Prey              | 88262
| Dying Prey               | 5235
-----------------------------------------
| Living Predators         | 390908
| Dying Predators          | 23477
-----------------------------------------
| Empty Cells              | 520830
| Total Cells              | 1000000
-----------------------------------------
```

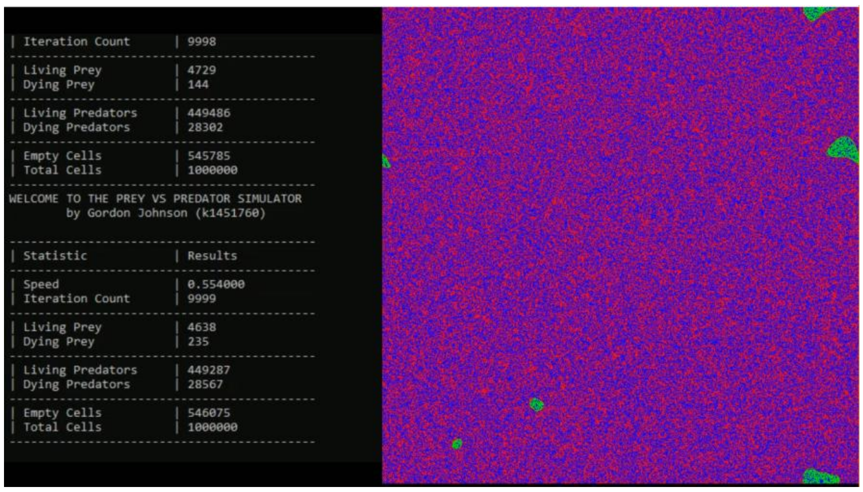
5000 Iterations



```
| Iteration Count          | 4999
-----------------------------------------
| Living Prey              | 34661
| Dying Prey               | 1910
-----------------------------------------
| Living Predators         | 428856
| Dying Predators          | 26416
-----------------------------------------
| Empty Cells              | 536483
| Total Cells              | 1000000
-----------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-----------------------------------------
| Statistic                | Results
-----------------------------------------
| Speed                    | 0.570000
| Iteration Count          | 5000
-----------------------------------------
| Living Prey              | 34248
| Dying Prey               | 2093
-----------------------------------------
| Living Predators         | 428870
| Dying Predators          | 26735
-----------------------------------------
| Empty Cells              | 536882
| Total Cells              | 1000000
-----------------------------------------
```

10000 Iterations



```
| Iteration Count          | 9998
-----------------------------------------
| Living Prey              | 4729
| Dying Prey               | 144
-----------------------------------------
| Living Predators         | 449486
| Dying Predators          | 28302
-----------------------------------------
| Empty Cells              | 545785
| Total Cells              | 1000000
-----------------------------------------
WELCOME TO THE PREY VS PREDATOR SIMULATOR
        by Gordon Johnson (k1451760)

-----------------------------------------
| Statistic                | Results
-----------------------------------------
| Speed                    | 0.554000
| Iteration Count          | 9999
-----------------------------------------
| Living Prey              | 4638
| Dying Prey               | 235
-----------------------------------------
| Living Predators         | 449287
| Dying Predators          | 28567
-----------------------------------------
| Empty Cells              | 546075
| Total Cells              | 1000000
-----------------------------------------
```

Performance

For the Hybrid implementation, the following tests were conducted, as presented in the introduction of this report. The previous Serial Implementation results, are included below, as a baseline comparison

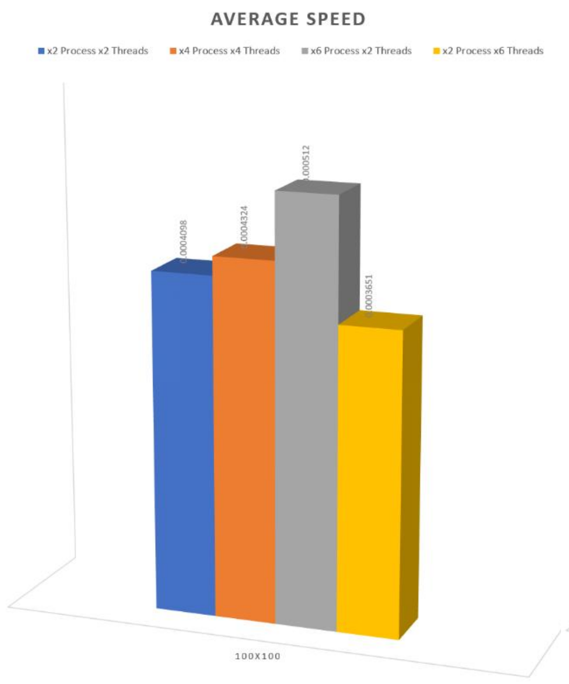
Hybrid Implementation

Grid Size	**100 x 100**	**1,000 x 1,000**	**10,000 x 10,000**
2x Processes 2x Threads	0.0004098	0.0332343	2.849304
4x Processes 4x Threads	0.0004324	0.0253125	2.234793
6x Processes 2x Threads	0.0005120	0.0311172	2.532860
2x Processes 6x Threads	0.0003651	0.0265826	1.977537

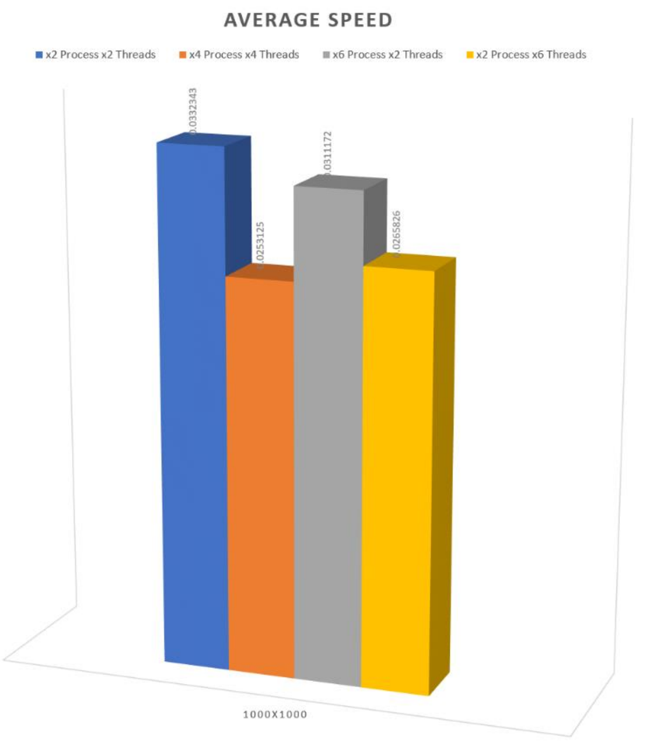
Serial Implementation Results (Baseline)

Grid Size	**100 x 100**	**1,000 x 1,000**	**10,000 x 10,000**
Average Speed per Iteration	0.0005381	0.0951833	8.722674

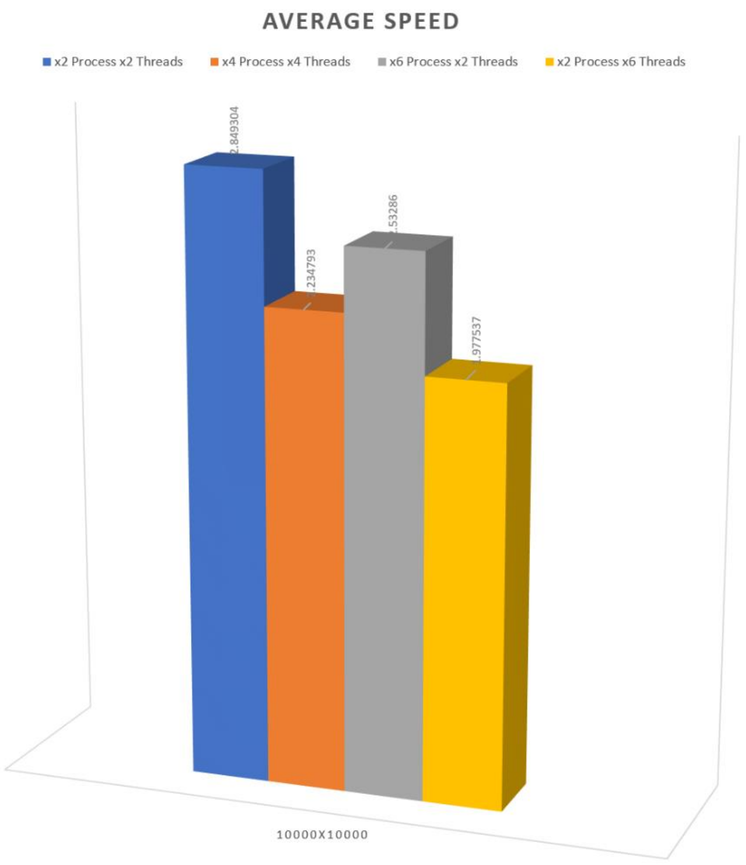
Note: Using 8x processes on the 100x100 grid, is considerably slower than its predecessor and falls below the baseline value. This could have been an issue with processor usage at the time of testing, it does not drop below the baseline for any other tests.



100x100 Grid Size (Average Speed)

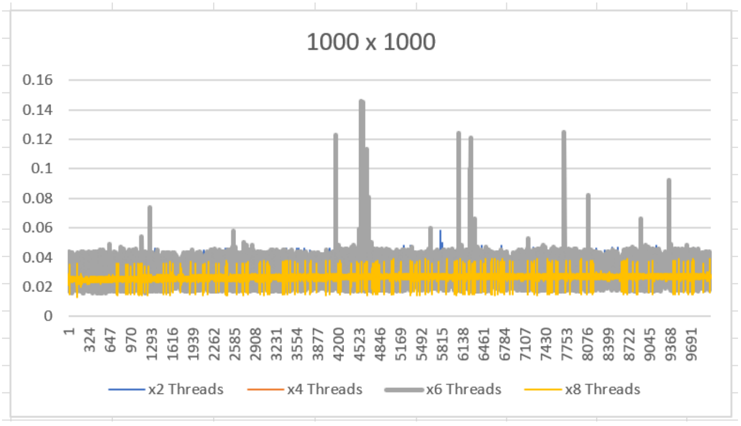


1000x1000 Grid Size (Average Speed)

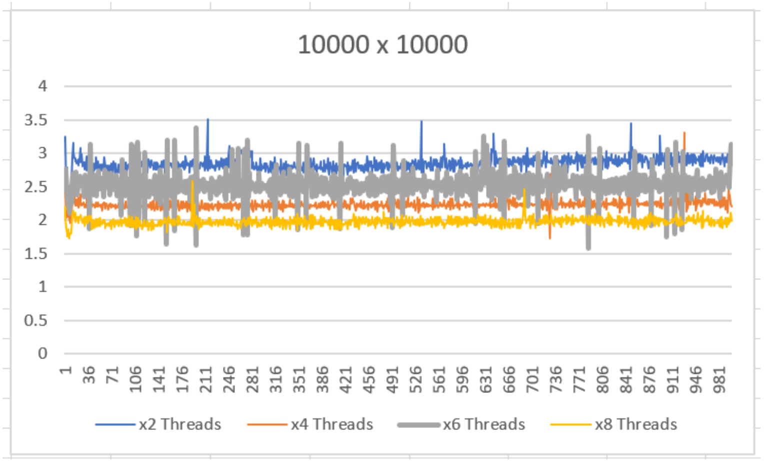


10000x10000 Grid Size (Average Speed)

Like the OpenMP implementation, this MPI version provides almost double the speed to the application, compared to the serial version by using 2x processes. Providing the simulation with x4 processes, provides the largest increase in acceleration compared to all other options. Providing x6 and x8, displayed minor improvements respectively, but still show better performance than the previous. The following graphs, show performance over time.



1000x1000 Grid Size (Performance per Iteration)



10000x10000 Grid Size (Performance per Iteration)

As can be seen from these charts, taken from the raw data, there is a gradual increase in seconds taken, indicative of a decline in performance over time.

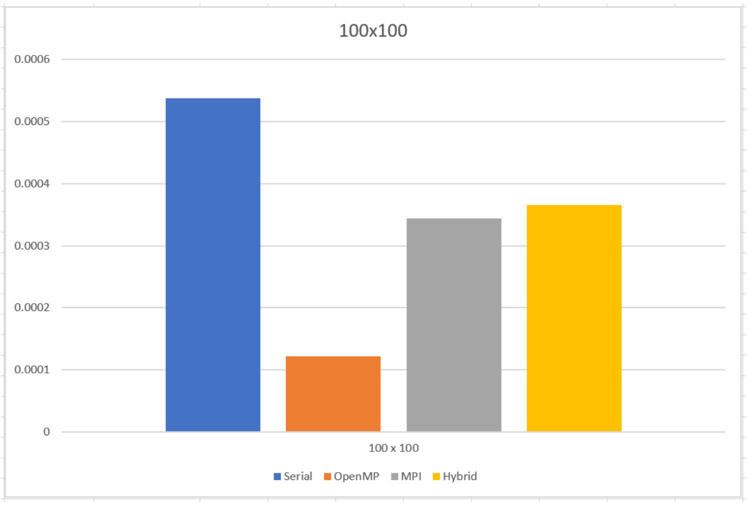
Also noted that the 100x100 grid with x8 threads, shows a decrease in performance. This could have been an issue with processor usage at the time of testing and does not reflect in the other tests with this amount of threads.

Evaluation

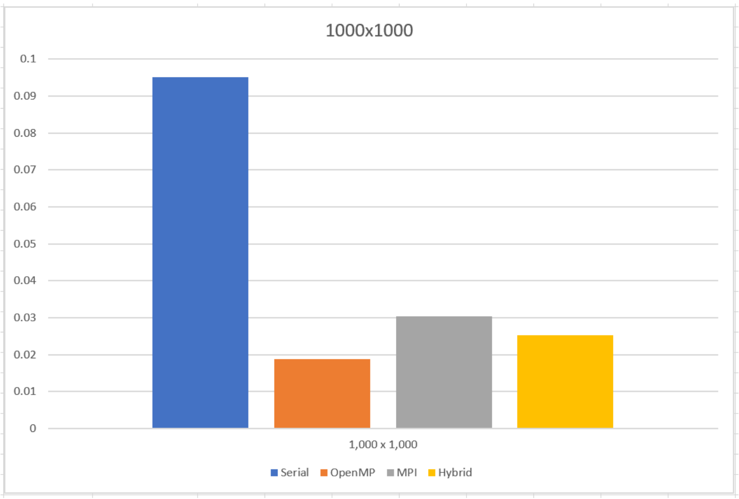
In conclusion, using Multi-Threaded or Multi-Processed technologies, adds considerable value to the overall performance of an application. The table highlights the best performing configurations, for each category of implementation, within each grid size.

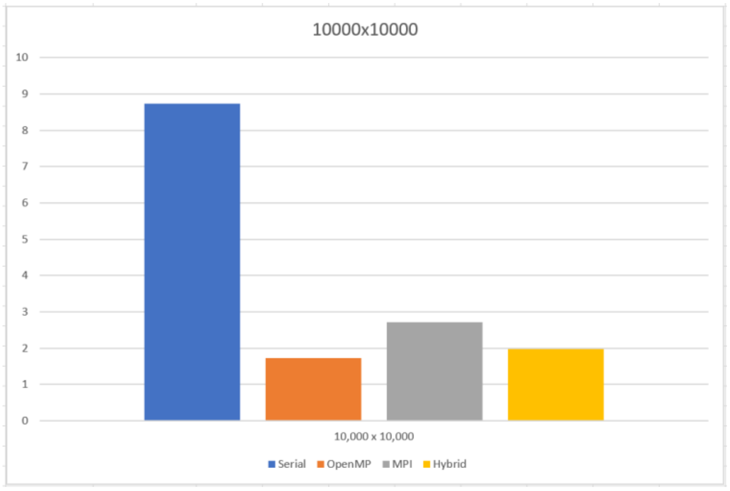
Grid Size	100 x 100	1,000 x 1,000	10,000 x 10,000
Serial	0.0005381	0.0951833	8.722674
OpenMP	0.0001213 x6 Threads	0.0187689 x8 Threads	1.723995 x8 Threads
MPI	0.0003442 x4 Processes	0.0302868 x6 Processes	2.706252 x6 Processes
Hybrid	0.0003651 x2 Processes x6 Threads	0.0253125 x4 Processes x4 Threads	1.977537 x2 Processes x6 Threads

As can be seen from the table above, OpenMP, performed the best across every scenario. The graphics below, make these results more visual. The 100 x 100 grid results where more varied, this highlights that these methods are more suited to a higher demanding application.



The 1,000 x 1,000 and 10,000 x 10,000 grid sizes, show a more consistent result. In both cases, OpenMP performed 5.07 and 5.06 times faster, than the baseline serial implementation. MPI, while performing slower than the OpenMP method, still provided extremely fast performance at 3.14 and 3.22 time faster than the baseline. The Hybrid implementation, improved the performance of the vanilla MPI version, by making the 1,000 x 1,000 test 3.76 times faster than baseline, however the 10,000 x 10,000 test, saw the largest performance boost of 4.41 times faster.





OpenMP, was considerably easier to implement into a serial model, by targeting specific loops where the application was demanding. Making this a cost-effective implementation, for overall performance increase.

The MPI implementation, required much more consideration and amendments to the serial version. To achieve message passing, additional loops where implemented, to ensure appropriate information was gathered, prior to evaluation of the states. These additional requirements provide a larger overhead, which could impede the performance of the MPI and Hybrid version. However, on larger scale infrastructure / applications, I would assume MPI would be more in its element, especially with multiple infrastructures across networked processing units. Mixing this with a Hybrid implementation, has also proved advantages when performance is key.

References

Manekar. A, Poundekar. M, Gupta. H, Nagle. M (2012) ‘A Pragmatic Study and Analysis of Load Balancing Techniques in Parallel Computing’, *International Journal of Engineering Research and Applications*, 2(4), pp.1914-1918.

MSDN (2018) *Microsoft MPI.* Available at: https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx (Accessed: 01 April 2018)

OpenMP (2018) *The OpenMP API specification for parallel programming.* Available at: http://www.openmp.org/ (Accessed: 29 March 2018)