The University of Queensland

COSC3500: High-Performance Computing

Semester 2, 2016

Genetic Algorithm
Project Report

Compiled by

Jordan Bishop

42916893

Contents

Chapter 1

Serial Implementation

1.1 Introduction

This report chronicles a high-performance computing (HPC) project under the topic of a genetic algorithm (GA). In this chapter, background theory of GAs is introduced, and the purpose and applications of GAs is discussed. Following this, a serial implementation of a GA to solve the OneMax optimisation problem is presented. Analysis of this implementation is conducted in order to verify its correctness, and performance scaling results obtained through implementation testing are showcased.

1.2 Background Theory

1.2.1 Genetic Algorithms

GAs are an approach to solving optimisation problems that are inspired by nature. They follow an iterative process which resembles evolution, at each iteration storing a *population* of individual candidate solutions to a problem. The population at any particular iteration of the algorithm is called a *generation*. At each iteration, a new generation is established, which is a modified version of the previous generation. The idea is that after enough iterations, the population will evolve into an optimal state that solves the underlying optimisation problem. Typically, solutions are represented as bitstrings, with a length determined by the size of the problem being solved. An optimal solution is a solution with an optimal configuration of bits, as defined by the problem.

The evolutionary process involves three main components. These are *selection*, *crossover*, and *mutation*; with the latter two components being referred to as the *genetic operators* of the algorithm [1]. Selection mimics natural selection, in that only 'strong' solutions are chosen to participate in creating the next generation. The metric used to measure the 'strength' of a solution is its *fitness*, which is problem dependent.

After solutions with high fitness are chosen, they participate in crossover, which is akin to sexual reproduction. Typically a crossover operation is performed with two 'parent' solutions, and produces two 'child' solutions. The mechanics of this operation involve combining genetic material (which in the case of bitstrings is a bit or series of bits) from both parents to produce children that resemble elements of each parent.

One important theory regarding crossover is known as the *building block hypothesis* [2], which attempts to explain the effectiveness of GAs, given their complicated nature. It states that some individuals in the population possess a set of traits, called 'building blocks' that cause them to have high fitness. When these high fitness individuals are selected to participate in crossover, they spread their building blocks throughout the population and help to create new solutions with even greater fitness; with this process repeating cyclically, causing the average fitness of the population to increase as evolution progresses.

After children have been produced from crossover, they are mutated in order to maintain genetic diversity in the population. Mutation for bitstrings can be as simple as flipping each bit in the string with a certain probability.

GAs are classified as a *metaheuristic* technique, and are included under the broader field of *stochastic optimisation*; which involves the incorporation of randomness into optimisation algorithms with the intention of allowing such algorithms to effectively sample the solution space and to prevent them from becoming trapped in local optima. In addition to their random nature, GAs are also black-box; in that they operate with limited or no understanding of the problem which they are solving. These properties make GAs very powerful general-purpose optimisation techniques that can be applied to a wide variety of problems.

1.2.2 The OneMax Problem

The OneMax problem is a trivial optimisation problem that serves as a good basis with which to verify correctness and performance of optimisation algorithms. It involves trying to maximise the number of 1 bits present in a bitstring [3]. A GA being used to solve the OneMax problem encodes solutions as bitstrings of a length n, where n is the size of the problem. An optimal solution to the problem is a bitstring with all n bits set to a value of 1.

1.3 Implementation Details

1.3.1 Algorithm

Pseudocode for a GA (adapted from [1, p. 37]) is given in Algorithm 1. The GA implementation for this report used a fixed population size of 100 for profiling and performance

Algorithm 1 The Genetic Algorithm

```
1: $popsize \leftarrow$ population size
2: $P \leftarrow \{\}$
3: for $popsize$ times do
4:     $P \leftarrow P \cup$ new random individual
5: $bestIndiv \leftarrow$ null
6: $bestFitness \leftarrow 0$
7: repeat
8:     for each $I \in P$ do
9:         $fitness \leftarrow assessFitness(I)$
10:        if $bestIndiv =$ null or $fitness > bestFitness$ then
11:            $bestIndiv \leftarrow I$
12:            $bestFitness \leftarrow fitness$
13:    $Q \leftarrow \{\}$
14:    for $popsize/2$ times do
15:        parents $P_a, P_b \leftarrow select(P)$
16:        children $C_a, C_b \leftarrow crossover(P_a, P_b)$
17:        $Q \cup \{mutate(C_a), mutate(C_b)\}$
18:    $P \leftarrow Q$
19: until $bestIndiv$ is an optimal solution or time limit reached
20: return $bestIndiv$
```

testing. As the implementation was tasked with solving the OneMax problem, its fitness function (represented as *assessFitness* on line 9 of Algorithm 1), simply returned the number bits with value 1 in a solution bitstring. The algorithm terminated when a solution bitstring consisting of all 1s was found.

There are a variety of different algorithms that can be used to implement the *select*, *crossover*, and *mutate* functions seen in Algorithm 1 on lines 15, 16, and 17 respectively. In this implementation, the *select* function used tournament selection with a tournament size of 2.

Tournament selection operates by firstly selecting (with replacement) a random individual from the population, and marking them as the current best individual. Next, a 'tournament' is run, where at each step of the tournament, a new individual is selected (again, with replacement) from the population and compared with the best individual. If they are fitter, they become the new best individual. This comparison process repeats until the number of individuals (including the initial individual) that have been selected is equal to the specified tournament size [1, p. 45]. The actual selection function in the program source code is passed a copy of the population (in order to prevent the population from being modified in the main algorithm) and returns two distinct parent solutions from this population. The function operates by performing an initial round of tournament selection on the population copy in order to determine the first parent, P_a, then removes this parent from the population copy and performs a second round of tournament selection to determine the second parent P_b.

The *crossover* function was implemented as uniform crossover with a bit swap probability of 0.5. Uniform crossover produces child solutions by initially declaring the first child, C_a, to be a copy of the first parent P_a, and the second child C_b, to be a copy of the second parent P_b. Next, both child bitstrings are iterated over, and their bits at each index are swapped with the specified probability [1, p. 39].

The algorithm used for the *mutate* function was a simple bit flip mutation algorithm, where each element of the bitstring was flipped with a probability of $\frac{1}{n}$, with n being the length of the bitstring.

1.3.2 Program Design

The implementation was programmed in C++, specifically the C++11 standard, and makes liberal use of many C++ language features in order to better encapsulate certain parts of program functionality and to make future maintenance and extension easier. Most notably, classes are used to represent an abstract optimisation problem (the *OptProb* class), and a concrete definition of an optimisation problem, which in the case of this report is the *OneMax* class. This abstract representation of an optimisation problem will make it easy to use different concrete problems with the same GA code in the future. Standard containers are also used extensively, with the aim of reducing use of raw pointers and making memory management easier. One especially appreciated feature of the C++11 standard that has been used are the pseudorandom number generators and probability distributions defined in the *<random>* header. These are far more sophisticated than the C *rand()* function, and provide better functionality in a domain where quality randomness is essential.

In an effort to help reduce program runtime, reference parameters to functions have been used where possible in order to eliminate the time cost associated with passing function parameters by value. Because of the use of higher level language features and reliance on C++ standard library algorithms, the code is significantly less primitive than standard C code, which makes it difficult to analyse, as discussed further in § 1.3.3.

1.3.3 Profiling and Optimisation

In the the interest of runtime analysis and to identify any hotspots, the program was compiled and linked with *g++* using the *-pg* flag and without compiler optimisations (the *-O0* flag was explicitly set). It was then subsequently run with a population size of 100 on an instance of the OneMax problem of size 100 to create profiling information for use with *gprof*.

The flat profile produced by *gprof* was quite convoluted, with many calls to C++ standard library functions appearing at the top of the profile when the default sort order of descending self seconds was used. The call graph was no simpler, and further reinforced the complexity of the program. Due to the interconnectivity of these standard library calls, it was difficult to ascertain exactly which user-written parts of the program were responsible for what proportion of particular calls, meaning locating worthwhile areas to manually optimise was likely not a fruitful path to follow.

Because of this, compiler-based optimisations were utilised, as they were likely to provide at least some degree of performance impact for zero manual cost. The optimisation flags *-O0*, *-O1*, *-O2*, and *-O3* were used with *g++* to compile program source code, and testing of program performance was conducted with each of these optimisation levels, as detailed further in § 1.5.

1.4 Verification Procedure

In order to confirm that the implementation was operating correctly, a series of verification runs were conducted with small population and problem sizes. These runs involved both executing the program from the command line with verbose output enabled and stepping through the program in the *gdb* debugger while monitoring the values of variables.

Verbose program output displays, at each generation, the solution bitstrings that comprise the population and all selection, crossover and mutation operations that lead to the formation of the next generation. An example of a portion of this output for a population size of 4 and a OneMax problem size of 5 is.

```
                   Generation 1:
                Individual 1: 01000
                Individual 2: 11100
                Individual 3: 00110
                Individual 4: 01101
                  Best individual:
           Individual 2: 11100, Fitness = 3
                Genetic operations:
01101 × 11100 → (11101, 01100) ∼ (11011, 01011)
00110 × 01101 → (01100, 00111) ∼ (00100, 00110)

                   Generation 2:
                Individual 1: 11011
                Individual 2: 01011
                Individual 3: 00100
                Individual 4: 00110
                  Best individual:
           Individual 1: 11011, Fitness = 4
```

$$\text{Genetic operations:}$$
$$01011 \times 00110 \rightarrow (00010,\ 01111) \sim (00110,\ 11101)$$
$$01011 \times 00110 \rightarrow (00110,\ 01011) \sim (00100,\ 01011)$$

The genetic operations are read as P_a crossover with P_b produces (C_a, C_b) which mutate to alternate versions of (C_a, C_b). By following this output until the final generation when an optimal solution is present, it is possible to trace the evolutionary process of the algorithm and verify that each generation is correct.

1.5 Performance Results

To determine the performance scaling capabilities of the serial GA implementation, a series of runtime experiments were conducted. These experiments made use of functions in the <*chrono*> header of the C++11 standard library to measure the total running time of the program.

Using a fixed population size of 100, and for four different levels of compiler optimisation (*O0*, *O1*, *O2*, and *O3*) the running time of the program was recorded 20 times each for differing OneMax problem sizes, starting at 100 and increasing to 1000 in increments of 50. For each of these (optimisation level, problem size) pairs, the mean running time was calculated. Figure 1.1 shows the results of these experiments.

With no compiler optimisation applied (optimisation level *O0*), the mean running time increased at a steady pace, from 937 ms at a problem size of 100 to 25 522 ms at a problem size of 1000. Transitioning to optimisation level *O1*, a drastic improvement in running time was seen. Compared to optimisation level *O0*, the curve for level *O1* is much more linear and has a less steep gradient. At a problem size of 100, the mean running time was 107 ms, and at a problem size of 1000, it was 3600 ms. This is an impressive result; with the first level of compiler optimisation applied, the program runs on average more than 7 times faster compared to when no compiler optimisation is used.

Unfortunately, increasing the amount of optimisation to level *O2* and further to level *O3* did not yield any more significant performance gains, with the curves for these two levels closely following the curve for level *O1*. Overall, it can be said that a compiler optimised version of the serial GA implementation exhibits good performance scaling for the OneMax optimisation problem.

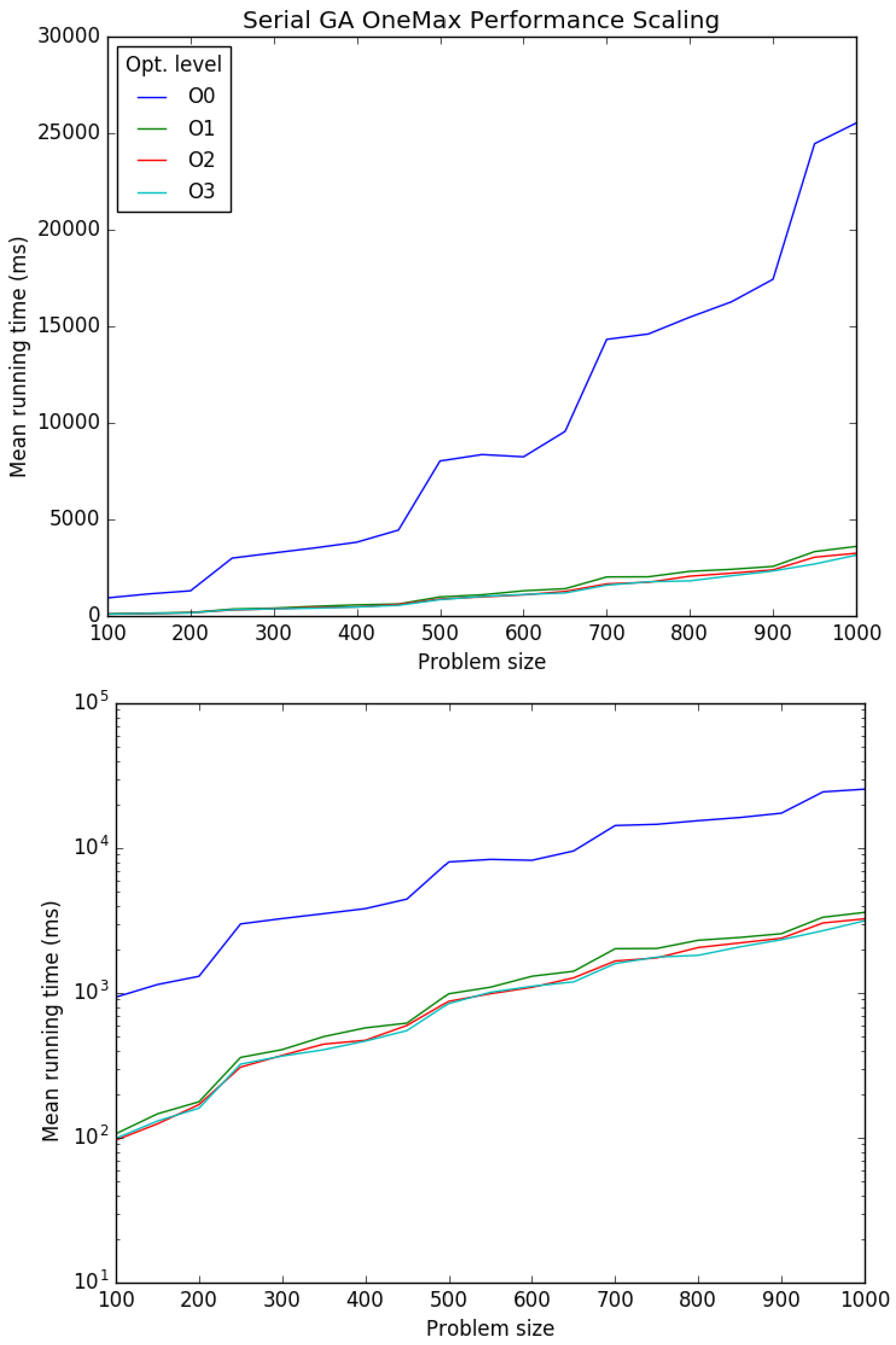

Figure 1.1: Performance scaling results of the serial GA implementation with population size 100 applied to the OneMax problem. Both plots show mean running time vs. problem size, with the bottom plot using a logarithmic scale for mean running time to better illustrate the relationship between the curves for optimisation levels *O1*, *O2*, and *O3*.

Chapter 2

Parallel Implementation

This chapter details the parallelisation of the serial GA implementation from Chapter 1. Firstly, a brief summary of parallelisation strategies for GAs is given; including an explanation of the specific strategy that was employed for this project. Following this is a description of how the chosen strategy was implemented into the existing serial codebase using both OpenMP and MPI, as well as a discussion on the procedures used to verify this parallel implementation. Finally, a plan for experimental testing of this implementation is given, which will form the basis of the next chapter of this report.

2.1 Parallelisation Strategies for Genetic Algorithms

There are a variety of ways in which a GA can be parallelised, with the two dominant methods being an *island/coarse-grained model*, and a *cellular/fine-grained model*, the latter being the model that was chosen to be implemented in this project.

In an island model, the population is split into groups, or 'islands'. Each island undergoes its own local evolutionary process, with individuals being allowed to periodically travel between islands in an effort to sustain genetic diversity among the set of islands. Each island can develop its own 'culture' of individuals; with the main idea of this approach being to explore the search space of the optimisation problem in parallel starting with a scattering of initial seeds around the space (the initial population of the islands). In terms of implementing this on a computer, it is straightforward to see the that each island can be run on one or more CPUs, or even on an entire compute node if enough resources are available.

A cellular model takes the approach of considering each individual as its own 'island', and only allowing an individual to interact with its close neighbours throughout the evolutionary process. The population of individuals in this model can be thought of as residing in a certain spatial embedding, with connections between individuals determining the topology of this embedding [1]. A prominent topology is that of a 2D toroidal grid/mesh, illustrated in Figure 2.1.

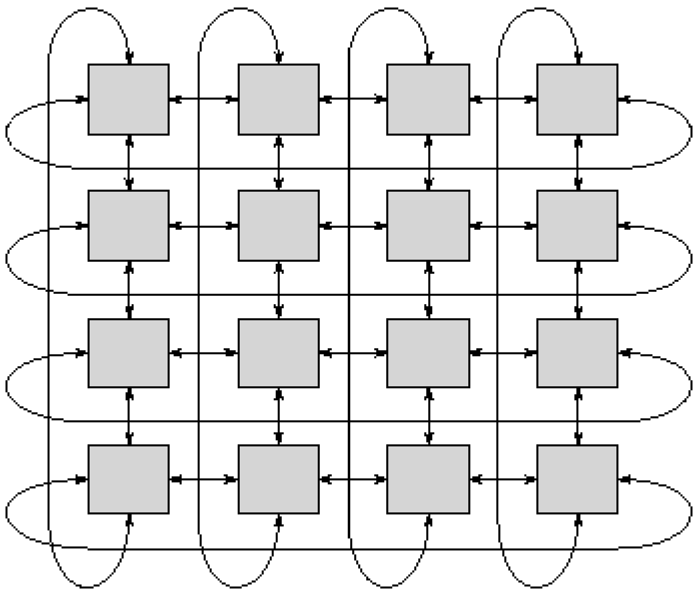

Figure 2.1: A toroidal mesh model as used in a cellular GA, from [4]. Each individual is connected to the four individuals to its north, south, east, and west, with wraparound connections for individuals on the edge of the mesh.

Each individual on the grid has a 'neighbourhood', which is a set of individuals close to it as defined by some combination of distance and shape. Examples are the L5 neighbourhood and the C9 neighbourhood, shown in Figure 2.2.

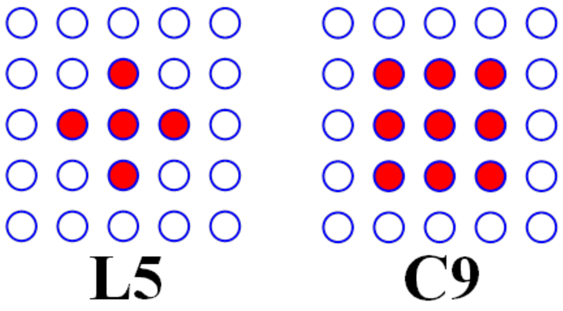

Figure 2.2: Two different types of neighbourhoods for cellular GAs. Image adapted from [5]. The L5 or 'Linear 5' neighbourhood encompasses individuals on both a horizontal and vertical axis from the individual whose neighbourhood is being determined (the 'center' individual), and has a total size of 5 individuals. The C9 or 'Compact 9' neighbourhood encompasses individuals in a 3×3 square surrounding the center individual, and has a total size of 9 individuals. Note that neighbourhood sizes are *inclusive* of the center individual.

Given a topology and a neighbourhood definition for the population, the typical GA process as per Algorithm 1 is applied, but with genetic operations for an individual being restricted to its neighbourhood. In a cellular model, clusters of similar individuals are formed, creating a niche in the grid; which acts similarly to an island in the island model.

Individuals of high fitness will continually be selected to reproduce in a niche, slowly propagating their genetic building blocks throughout the grid and raising the fitness of the overall population.

2.2 OpenMP and MPI Usage

2.2.1 Computational Representation

Both OpenMP and MPI were used to convert the serial GA implementation in Chapter 1 into a parallel, cellular model GA. The emphasis of a cellular GA is to consider *each* individual in the population as a separate entity, which naturally translates to the computational representation of one individual per CPU. However, using this representation was not feasible given the resources available to complete project work. On the Savanna HPC cluster, the *iceq* partition was made available for use by all COSC3500/7502 students. This partition consisted of 32 nodes, each with 8 CPUs, meaning that if the author was able to obtain access to *all* resources on this partition (meaning no other student could use the partition), then a maximum of 256 individuals could be simulated. This is obviously not a practical (or fair) scenario, and as such this representation was not considered.

Figure 2.3 shows the alternate representation that was used. In this representation, each MPI rank has access to multiple CPUs, meaning OpenMP shared memory parallelism can be employed intra-rank for tasks such as fitness assessment and genetic operations. As such, this approach makes more efficient use of the available computational resources compared to the individual-per-CPU approach; the computational load of the population is spread across a modest number of nodes via multi-process distributed memory parallelism, where each node can locally make use of multi-threading shared memory parallelism to speed up its operations. Thus, large population sizes can be simulated without using an entire cluster partition, as long as the population is appropriately spread across a reasonable number of ranks.

Figure 2.3 also shows how the notion of neighbourhood extends to this rank-based representation. The local population of each rank can be numbered from individual 1 to individual 25, starting at the top left hand corner and continuing in a left-to-right fashion until the bottom right hand corner. The green, red, and blue shaded groups respectively form the L5 neighbourhoods of individuals 1, 13, and 25 on rank 0. The entire neighbourhood of individual 13 resides on rank 0, while the neighbourhoods of individuals 1 and 25 are spread across multiple ranks. This means that each rank is dependent on the ranks to its north, south, east, and west; specifically being dependent on individuals along specific ‘edges’ of the local population grids of these adjacent ranks.

It is also worth noting that there are a few constraints on population sizes and number of ranks that can be used with this representation. Firstly, the total population size must

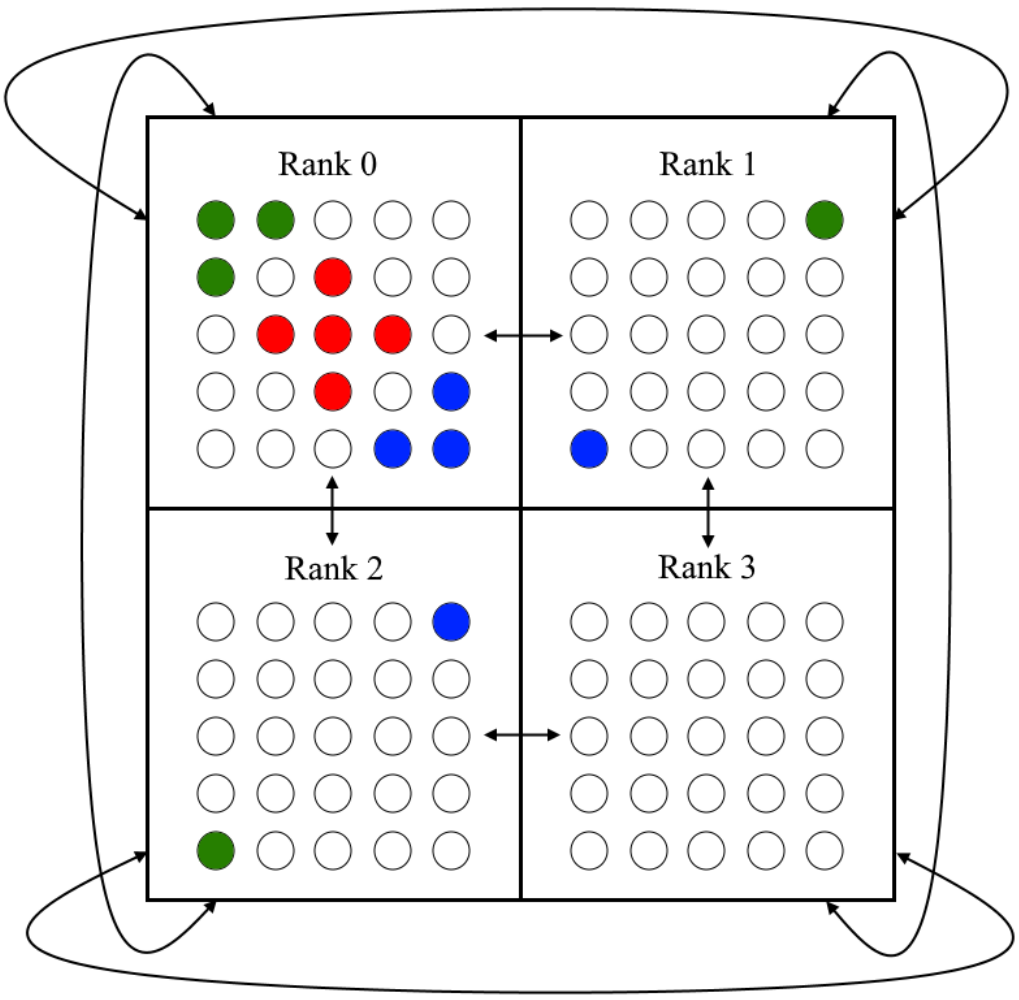

Figure 2.3: Computational representation of the population for the implemented cellular GA. MPI is used to distribute the population across a number of compute nodes, or *ranks*. Each rank holds an equally sized subset of the total population; its *local population*. In this example, there are 4 ranks numbered from 0 to 3, and the total population size is 100 (10×10 grid), giving each rank a local population size of 25 (5×5 grid). Similar to the toroidal mesh from Figure 2.1, each rank is fully connected to its neighbouring ranks in the north, south, east, and west directions, with wraparound connections being present; so that, for example, the 'north' neighbour of rank 0 is rank 2, and the 'west' neighbour is rank 1.

be a square number (so that the entire population grid is a square), and the 'side length' of this square grid must be able to be equally divided among the number of desired ranks. Thus, the local population grid of each rank is also square in shape, and has a side length of $\frac{global\ side\ length}{num\ ranks}$.

2.2.2 Application to Existing Serial Code

There were two major steps involved in applying the discussed parallel representation to the existing serial implementation. The first was redesigning the architecture of the codebase so that MPI parallelism could be introduced, and the second was the application of profiling to determine areas in which the introduction of OpenMP parallelism could see potential performance gains.

The first step was the most involved, with a majority of the programming effort being focussed on correctly implementing message passing between different ranks. The second step was much less time consuming, thanks to the fact that OpenMP parallelism can be added to a program through the simple introduction of compiler directives.

Algorithm 2 gives pseudocode for how the parallel representation in § 2.2.1 was integrated into the serial GA from Algorithm 1. It differs from Algorithm 1 in the following ways:

1. At the beginning of the algorithm, each rank initialises its own random local population (Lines 2 — 4).

2. At the start of the main generational loop, each rank determines its best local individual (Lines 10 — 14).

3. Each rank then sends their best local individual to the root rank (Line 15). This is implemented in code with an `MPI_Gather` call.

4. Once the best individuals from each rank have been received by the root, the best global individual is determined (Lines 16 — 21).

5. Next, each rank must send and receive dependent information (Lines 23 — 26). This is achieved in code with non-blocking `MPI_Isend` and `MPI_Irecv` calls. The function *getDependentRank*(*myRank*, *direction*) on Line 24 returns the rank that is adjacent to *myRank* in the specified *direction*. The function *opposite*(*direction*) on Line 26 simply returns the opposite direction from the one supplied; e.g. $opposite(North) = South$. This is required because the rank to the north of *myRank* needs to supply *myRank* with the individuals along its southern edge, while *myRank* supplies this northern rank with the individuals along its northern edge.

6. Genetic operations are then applied to the local population and the new local population is built (Lines 27 — 35). This immediately differs from the serial implementation in that each individual in the local population undergoes a parent selection process, then these parents are used to generate a single child solution. If this generated child has greater fitness than the original individual, the original individual is replaced by the child in the new local population. Parent selection is not done using the entire (local) population; instead each individual has parents selected from its neighbourhood (L5 neighbourhood was used in code). Like the serial implementation, crossover is then applied to these parents to produce two children. After this, a single child is randomly selected with probability 0.5, mutated, and then assessed to

Algorithm 2 Parallel Cellular Genetic Algorithm

```
1: Input: $localPopsize$, $myRank$, $rootRank$
2: $LP \leftarrow \{\}$                                          ▷ The local population
3: for $localPopsize$ times do                                  ▷ Initialise local population
4:     $LP \leftarrow LP \cup$ new random individual
5: $bestLocalIndiv \leftarrow$ null
6: $bestLocalFitness \leftarrow 0$
7: $bestGlobalIndiv \leftarrow$ null
8: $bestGlobalFitness \leftarrow 0$
9: repeat
10:     for each $I \in LP$ do                                  ▷ Find the best local individual
11:         $fitness \leftarrow assessFitness(I)$
12:         if $bestLocalIndiv =$ null or $fitness > bestLocalFitness$ then
13:             $bestLocalIndiv \leftarrow I$
14:             $bestLocalFitness \leftarrow fitness$
15:     Send $localBestIndiv$ to $rootRank$
16:     if $myRank = rootRank$ then                             ▷ Find the best global individual
17:         for each $I \in$ received individuals do
18:             $fitness \leftarrow assessFitness(I)$
19:             if $bestGlobalIndiv =$ null or $fitness > bestGlobalFitness$ then
20:                 $bestGlobalIndiv \leftarrow I$
21:                 $bestGlobalFitness \leftarrow fitness$
22:     $LQ \leftarrow \{\}$                                   ▷ Create the new local population
23:     for each $direction \in N, S, E, W$ do                  ▷ Send and receive dependencies
24:         $dependentRank \leftarrow getDependentRank(myRank, direction)$
25:         Send individuals along $direction$ edge of my local grid to $dependentRank$
26:         Receive individuals from $opposite(direction)$ edge of $dependentRank$
27:     for each $I \in LP$ do                                  ▷ Do local genetic operations
28:         parents $P_a, P_b \leftarrow selectFromNeighbours(I)$
29:         children $C_a, C_b \leftarrow crossover(P_a, P_b)$
30:         $keptChild \leftarrow selectRandomChild(C_a, C_b)$
31:         $mutate(keptChild)$
32:         if $assessFitness(keptChild) > assessFitness(I)$ then
33:             $LQ \cup keptChild$
34:         else
35:             $LQ \cup I$
36:     $LP \leftarrow LQ$
37: until $bestGlobalIndiv$ is an optimal solution or time limit reached
38: return $bestGlobalIndiv$
```

determine whether it should join the new local population. The selection, crossover, and mutation methods used in this parallel implementation are the same as those described in § 1.3.1.

Initially, this algorithm was implemented into the serial codebase using only MPI functionality. The program was then profiled using `gprof` to determine hotspot areas that would likely benefit from the introduction of OpenMP parallelism. Based on the flat profile produced by `gprof`, it was found that operations consuming large portions of program time were (in order):

1. Fitness assessment of individuals
2. Mutation
3. Crossover
4. Constructing `Solution` objects from data arrays received via MPI (these objects are used to represent individuals in the population).
5. Converting `Solution` objects to data arrays to be sent via MPI.

All of these operations involved looping over some kind of construct, and so were easily able to be parallelised with the use of `omp parallel for` pragmas.

2.3 Verification Procedure

Similar to the verification runs conducted for the serial GA implementation (as discussed in § 1.4), a set of small-scale experiments were run to verify the correctness of the parallel implementation when used to solve the OneMax optimisation problem. The parallel code was designed so that each rank would output its `stdout` and `stderr` to a separate file, making it easy to debug and determine exactly what each rank was doing. A command line parameter can be passed to the program to turn on verbose debugging output, which is printed to `stderr`. This output details all steps of the algorithm, including printing out for each generation:

- The local population
- The best local individual
- The best local individuals received over MPI (only on the root rank)
- Individuals received over MPI from dependent ranks
- Neighbours for each individual in the local population
- The genetic operations for each individual in the local population

One such verification experiment used a global population size of 16, OneMax problem size of 7, 4 MPI ranks, 4 OpenMP threads; and was run using 4 nodes, 1 task per node, and 4 CPUs per task. Each rank had a local population size of $\frac{16}{4} = 4$ individuals. Example output of this experiment for the first generation of the root rank (rank 0) is:

```
ROOT: Using a global popoulation size of 16
ROOT: Creating an instance of problem 'one_max' with size 7
ROOT: This problem has a solution size of 7
Hello from rank 0! The world has size 4
Using 4 threads per rank
Local population size = 4
Initialising the local population...
My dependent ranks are: 2 2 1 1
Side length = 2
Center indices in my local grid are:
Edge indices in my local grid are: 0 1 2 3
Creating random objects...
Entering main generational loop...

Generation 1:
DEBUG: Individual 1: 1010101
DEBUG: Individual 2: 1110010
DEBUG: Individual 3: 1100001
DEBUG: Individual 4: 0101101
Best individual:
Individual 1: 1010101, Fitness = 4
Sending my best local individual to the root...
ROOT: Determining if any rank has found an ideal individual...
ROOT: Individuals received:
ROOT: Rank 0: 1010101
ROOT: Rank 1: 1111100
ROOT: Rank 2: 1100111
ROOT: Rank 3: 1111011
DEBUG: Doing receives...
DEBUG: Filling send buffers...
DEBUG: Doing sends...
DEBUG: Breeding center...
DEBUG: Waiting on receives...
DEBUG: Converting received data...
DEBUG: Individuals from rank to the N:
1100111, 0010111
DEBUG: Individuals from rank to the S:
0000000, 0010010
DEBUG: Individuals from rank to the E:
```

```
1111100, 1011000
DEBUG: Individuals from rank to the W:
0010000, 0100110
DEBUG: Breeding edges...
DEBUG: Neighbours for individual 1: 1100111 1100001 1110010 0010000
DEBUG: Neighbours for individual 3: 1010101 0000000 0101101 0100110
DEBUG: Neighbours for individual 2: 0010111 0101101 1111100 1010101
DEBUG: Neighbours for individual 4: 1110010 0010010 1011000 1100001
DEBUG: 1100001 X 0010000 -> (1110001, 0000000) -> 0000000 ~ 1001100
DEBUG: 0100110 X 0000000 -> (0100110, 0000000) -> 0000000 ~ 1000100
DEBUG: 1111100 X 0010111 -> (1010100, 0111111) -> 0111111 ~ 0110111
DEBUG: 1100001 X 1110010 -> (1100011, 1110000) -> 1100011 ~ 1110011
DEBUG: Waiting on sends...
DEBUG: Updating population...
```

Within this output, genetic operation debug statements are read as:
`parent X parent -> (child a, child b) -> kept child -> mutated kept child`.

In this scenario, ranks 1, 2, and 3 also produced similar output, but without any `ROOT:` statements. By tracing the execution of each rank and drawing diagrams to depict the local population of each rank, it was possible to manually verify that the steps performed by the algorithm were correct for small-scale implementations.

2.4 Experimentation Plans

The verification experiments performed thus far have all been with no additional compiler optimisations (explicitly setting the `-O0` flag during compilation). As discussed in § 1.3.3, the heavy reliance on C++ standard library functionality in the serial implementation made it largely counter-productive and/or difficult to introduce any amount of noticeable manual optimisations into the program. Performance results for the serial implementation showed that any amount of compiler optimisation greatly improved performance.

The parallel implementation is still heavily reliant on the C++ standard library, and so the introduction of manual optimisations was not considered. This decision was again reinforced by inspection of the flat profile generated by `gprof` during the investigation of where to include OpenMP functionality; the flat profile was extremely complicated, with a large number of calls to library functions. It is likely that the introduction of compiler optimisations will, like in the serial case, significantly improve the performance of the parallel implementation.

These compiler optimisations, specifically flags `-O1`, `-O2`, and `-O3` are one element of the implementation that will need to be experimented with in the near future. Future

Table 2.1: Potential Parallel Implementation Experiments (a subset)

Number	**Compiler Optimisation Level**	**# MPI Ranks**	**# OpenMP Threads**	**Global Population Size**	**Problem Size**
1	-O0	4	1	100	1000
2	-O1	4	1	100	1000
3	-O2	4	1	100	1000
4	-O3	4	1	100	1000
5	TBD	4	1	100	2000
6	TBD	4	1	100	5000
7	TBD	4	1	100	10000
8	TBD	4	1	100	15000
9	TBD	4	1	100	20000
10	TBD	4	2	100	2000
11	TBD	4	2	100	5000
12	TBD	4	2	100	10000
13	TBD	4	2	100	15000
14	TBD	4	2	100	20000
15	TBD	4	4	100	2000
16	TBD	4	4	100	5000
17	TBD	4	4	100	10000
18	TBD	4	4	100	15000
19	TBD	4	4	100	20000
20	TBD	4	6	100	2000
21	TBD	4	6	100	5000
22	TBD	4	6	100	10000
23	TBD	4	6	100	15000
24	TBD	4	6	100	20000
25	TBD	4	8	100	2000
26	TBD	4	8	100	5000
27	TBD	4	8	100	10000
28	TBD	4	8	100	15000
29	TBD	4	8	100	20000

experiments will also be concerned with varying the number of MPI ranks used, number of OpenMP threads, the global population size, and the size of the optimisation problem being solved.

Tests thus far have all been run using the OneMax optimisation problem, however there is certainly potential to define a different optimisation problem, as long as its solutions can be easily encoded as bitstrings and there is a concrete method with which to measure the fitness of these bitstrings.

A set of potential experiments to evaluate the performance of the parallel implementation for the OneMax problem is given in Table 2.1.

Experiments 1 — 4 will be used to determine a suitable compiler optimisation level, which will then be used throughout the rest of experimentation. Experiments 5 — 9 will demonstrate how a small number of ranks, lack of local multithreading, and small population size scale to large problem sizes. The number of threads available to each rank will then be increased in experiments 10 — 14, and further increased in experiments 15 — 19, 20 — 24, and 25 — 29 in order to test the effect of OpenMP parallelism.

In order to limit the size of Table 2.1, the remaining sets of experiments will be described based on those sets already in the table. Experiment sets 5 — 29 will attempted to be rerun with 9 MPI ranks, and all other parameters the same; and this will give some insight into how distributed parallelism affects performance.

Experiment sets 5 — 29 will also be rerun using larger global population sizes to show the effect of global population size on performance. If time and resources permit, a combination of larger global population sizes, increased number of MPI ranks, and increased number of OpenMP threads will also be investigated.

It is worth mentioning that, at this stage, it is not known whether the parallel implementation will be able to solve a OneMax problem of size 20,000 in a reasonable amount of time. As such, the proposed problem sizes are subject to change.

Chapter 3

Performance Analysis

This chapter gives a performance analysis for the parallel implementation developed in Chapter 2. Firstly, a description of the experimental process followed in order to conduct this analysis is detailed. Included as part of the analysis are scalability results, demonstrating how the runtime of the program behaves given varying computational resources and problem sizes. From these results, conclusions are drawn about the scientific findings of the project and the suitability of applying HPC techniques to solve the OneMax problem using a genetic algorithm.

3.1 Experimental Process

As part of performance analysis work, the preliminary experimentation plans outlined in § 2.4 were modified slightly.

Initially, as described in the preliminary plan, a small set of experiments were conducted to determine a suitable compiler optimisation level to use for the remaining experiments. These experiments were simply trying to verify that the parallel implementation did indeed run correctly when optimisations were applied. Using a global population size of 100, a OneMax problem size of 1000, and a fixed number of MPI ranks and OpenMP threads (both 1), four different compiler optimisation levels (`O0`, `O1`, `O2`, and `O3`) were trialled. Table 3.1 summarises the results.

Table 3.1: Compiler optimisation level evaluation for parallel implementation

Compiler optimisation level	**Mean running time over 10 trials (s)**
`O0`	24.74
`O1`	15.32
`O2`	12.03
`O3`	11.12

The output of these simple experiments was also verified, as per the process described in § 2.3, to ensure that the results being produced were correct, and that compiler optimisations had not interfered with the correctness of the program. From these results, it was clear that compiler optimisation level `O3` was the best choice. Thus, the remaining experiments conducted all used this optimisation level.

Table 3.2 gives the list of experiments that were conducted as part of the performance analysis.

Table 3.2: Performance Analysis Experiments

Experiment Set Label	**MPI Ranks**	**OpenMP Threads**	**Global population size**	**Problem Size**
OpenMP scaling single rank	1	1	1296	2000, 4000, 6000, 8000, 10000
	1	2	1296	2000, 4000, 6000, 8000, 10000
	1	3	1296	2000, 4000, 6000, 8000, 10000
	1	4	1296	2000, 4000, 6000, 8000, 10000
	1	5	1296	2000, 4000, 6000, 8000, 10000
	1	6	1296	2000, 4000, 6000, 8000, 10000
	1	7	1296	2000, 4000, 6000, 8000, 10000
MPI scaling one thread	4	1	1296	2000, 4000, 6000, 8000, 10000
	9	1	1296	2000, 4000, 6000, 8000, 10000
	16	1	1296	2000, 4000, 6000, 8000, 10000
Global pop size MPI + OpenMP scaling	4, 9, 16	1, 3, 5, 7	144	2000, 4000, 6000, 8000, 10000
	4, 9, 16	1, 3, 5, 7	576	2000, 4000, 6000, 8000, 10000
	4, 9, 16	1, 3, 5, 7	1296	2000, 4000, 6000, 8000, 10000

The first experiment set was designed to determine the scaling of OpenMP as the problem size increased, with the number of MPI ranks being fixed at 1, and the global population size being fixed at 1296. The second set of experiments was designed to evaluate MPI scaling as the problem size increased, with the number of OpenMP threads being fixed to 1. The final set was designed to evaluate the scaling of both MPI and OpenMP across varying population sizes and problem sizes.

To the uninitiated reader, the population sizes of 144, 576, and 1296 in the above table may seem strange, so here is an explanation of how they were chosen. Because of the way MPI parallelisation was implemented, the global population size was constrained to be a square number, and the local population size of any rank was also constrained to be a square number (for simplicity of coding). The number of ranks that were being used across all experiments were 4, 9, and 16. Thus, the global population size must be a square number that can be divided evenly by 4, 9, and 16 to produce another square number. 144 is the smallest such number for which this is possible, and the other numbers that also make this scenario possible are square multiples of 144; the next two such numbers being 576 and 1296.

These experiment sets were run on the Savanna cluster, with the largest amount of resources required for allocation at any given time being 16 nodes with 7 CPUs per node. Each individual experiment was executed 5 times, and the result recorded was the mean result of these 5 trials. See Figure 11 for an example Slurm submission script used to execute the last experiment set (global population size MPI + OpenMP scaling).

3.2 Scalability Results

This section details the results produced from conducting the experiment sets in Table 3.2.

Figure 3.1 shows the results for the 'OpenMP scaling single rank' experiment set. It can be seen that as the number of threads increases on a single rank, the mean running time decreases. The effect on reducing running time diminishes as more threads are added. This is reflective of Amdahl's Law; for a fixed problem size, the speedup attained by parallelisation is bottlenecked by the portion of the program code that cannot be parallelised.

Figure 3.3 illustrates this scenario, showing the speedup attained by OpenMP parallelisation for a fixed problem size of 10,000 and fixed population size of 1296. Note that the line in this plot is slightly convex; implying that if the number of threads was increased above 7, there would be a plateau point where speedup is maximised, and increasing the number of threads used past this point would not yield any performance benefit. Thus, as to be expected, parallelisation with OpenMP *does not* yield perfect strong scaling.

The results for the 'MPI scaling single thread' experiment set are shown in Figure 3.2. Like in the case of sole OpenMP parallelisation, the mean running time decreases when more ranks are used. Figure 3.4 demonstrates the speedup for MPI parallelisation; using the same scenario as in the case of OpenMP parallelisation (fixed problem size of 10,000 and fixed population size of 1296).

Because there are only three data points on this plot (as only three different rank counts were tested), it is not as easy to observe the convex nature of the speedup curve. However, *if* more rank counts were tested with, the behaviour seen in Figure 3.3 would again occur, implying lack of perfect strong scaling. In the case of MPI, the lack of strong scaling can additionally be attributed to the existence of increased communication overhead as the number of ranks increases.

Figures 3.5, 3.7, and 3.9 respectively show scaling results for population sizes of 144, 576, and 1296, with varying numbers of MPI ranks and OpenMP threads. These results correspond to the 'Global pop size MPI + OpenMP scaling' experiment set from Table 3.2. Each of these figures has a 'zoomed' counterpart beside it to make it easier to distinguish and separate the lines at the base of the plot. Within these plots, the experiments that

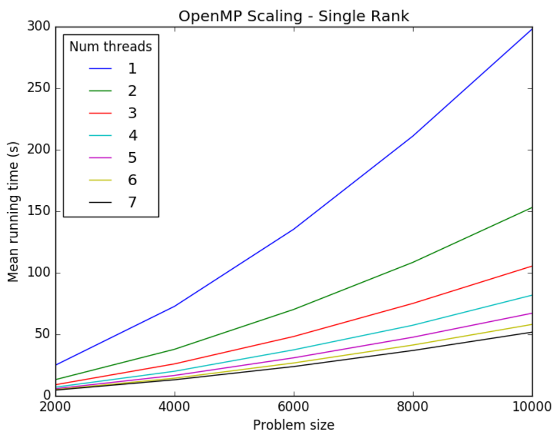

Figure 3.1: Scaling results for OpenMP running on a single MPI rank.

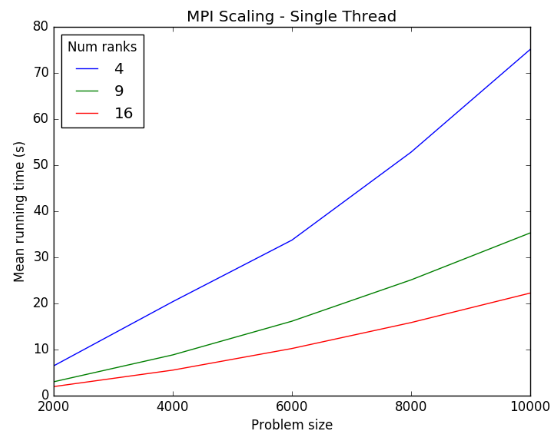

Figure 3.2: Scaling results for multiple MPI ranks using a single OpenMP thread

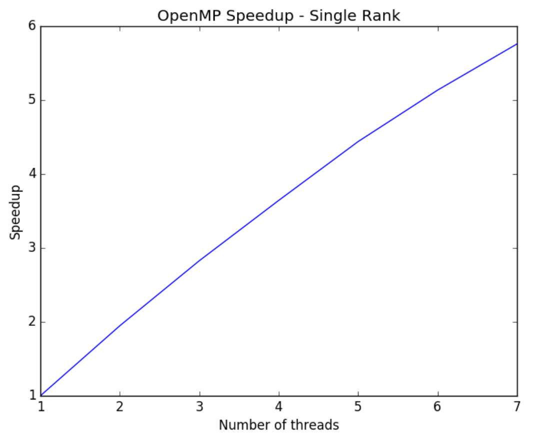

Figure 3.3: Speedup results for OpenMP running on a single MPI rank, with a fixed problem size of 10,000 and a fixed population size of 1296.

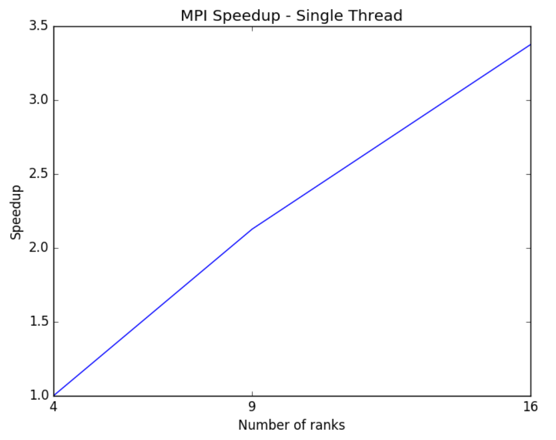

Figure 3.4: Speedup results for multiple MPI ranks using a single OpenMP thread, with a fixed problem size of 10,000 and a fixed population size of 1296.

used 4 MPI ranks are coloured in blue, 9 MPI ranks are coloured in green, and 16 MPI ranks are coloured in red. Different shades of these colours indicate different numbers of OpenMP threads; the lightest shade indicates 1 thread and the darkest shade indicates 7 threads, with the shades in between indicating 3 and 5 threads.

For a population size of 144 (Figures 3.5 and 3.6), it can be seen that overall, when using 4 and 9 ranks, increasing the number of OpenMP threads does improve the mean running time (the darker coloured blue and green lines are mostly below lighter coloured ones). When 16 ranks are used, a speedup is seen when the number of threads is increased from 1 to 3 and from to 5. However, when an increase from 5 to 7 threads is made, the mean running time actually increases for all problem sizes.

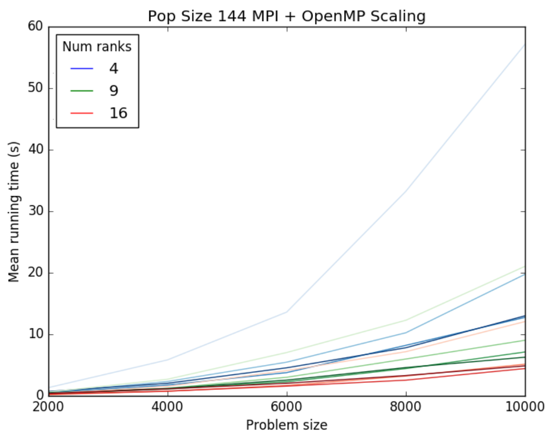

Figure 3.5: Scaling results for a global population size of 144, and varying numbers of MPI ranks and OpenMP threads.

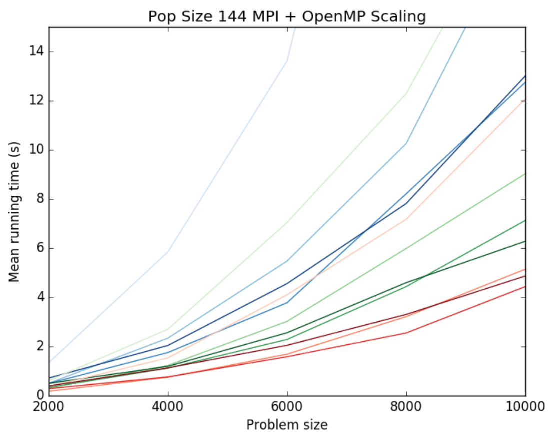

Figure 3.6: A zoomed version of Figure 3.5

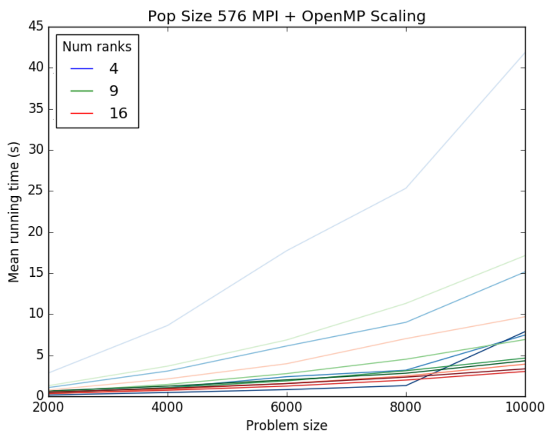

Figure 3.7: Scaling results for a global population size of 576, and varying numbers of MPI ranks and OpenMP threads.

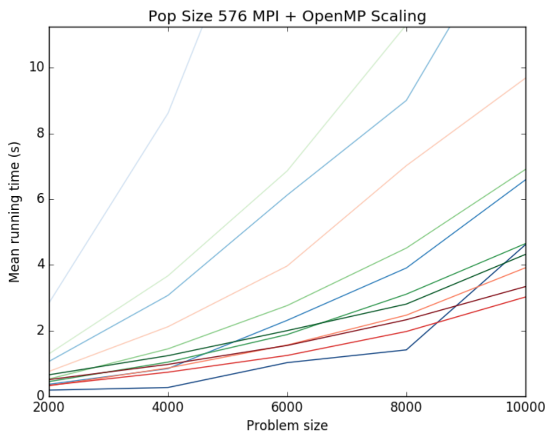

Figure 3.8: A zoomed version of Figure 3.7

At 16 ranks with a global population size of 144, each rank has a local population size of only 9. Because OpenMP parallelisation is implemented in a loop that performs genetic operations for each individual in the local population, 7 threads would be used to distribute work for only 9 individuals. In this situation, what is likely happening is that the overhead associated with dividing the work between 7 threads is actually outweighing the benefit of parallelising such a small amount of work. This is precisely why larger population sizes were also included in this experiment set.

For a population size of 576 (Figures 3.7 and 3.8), similar behaviour to a population size of 144 is observed; with the speedup obtained by using 16 ranks and 7 threads still smaller than using 16 ranks and 5 threads. When the population size is increased to 1296 (Figures 3.9 and 3.10), the situation becomes better, with roughly equal speedup being

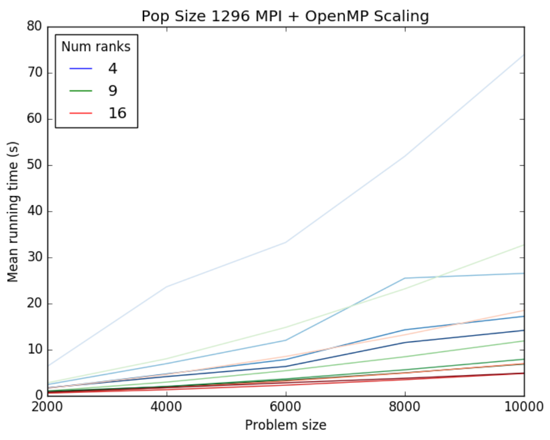

Figure 3.9: Scaling results for a global population size of 1296, and varying numbers of MPI ranks and OpenMP threads.

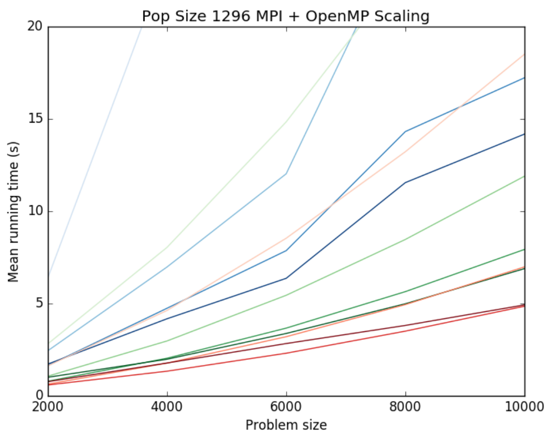

Figure 3.10: A zoomed version of Figure 3.9

seen for the 16 rank case with 5 and 7 threads, for the largest problem size of 10000. If larger problem sizes were tested and the plot was extended to include these, the speedup for the 7 threads case would likely be greater than for the 5 threads case.

These results reflect the idea of Gustafson's Law; namely that as the amount of available computing resources increases, the size of problems that can be solved in an equivalent amount of time also increases, i.e. the workload is adjusted to suit the resources.

3.3 Conclusion

Overall, a genetic algorithm is an algorithm that is well suited to parallelisation. The problem used for experimentation throughout this project, the OneMax problem, is a simple and effective problem to use for the purposes of testing the performance of a GA. The specific type of GA implemented for the project (a fine-grained/cellular GA) was quite intuitive to hybridise with both MPI and OpenMP parallelism. MPI was used to split the population of individuals across multiple ranks, and OpenMP was used to parallelise genetic operations intra-rank.

The scalability results produced show that (as expected) the algorithm does not exhibit perfect strong scaling when the problem size is kept fixed and compute resources are increased. This is because the work performed by the algorithm is composed of a serial section and a parallel section, and the serial section 'bottlenecks' the parallel section when more computational resources are given to solve the problem.

One notable serial section of the algorithm that was difficult to parallelise was fitness evaluation of the local population of a rank at the beginning of each generation. This is difficult to parallelise properly because of the fact that each individual must be assessed, then compared to the current best individual to determine if they are the best. Then, the best individual is updated if required. Using OpenMP to parallelise this was likely a counter-productive strategy; the best individual is a shared object that must have access to it controlled via locks. Threads should not be able to write to (modify) the object while other threads are reading its value, forcing this task to be executed in serial.

The produced results also consider weak scaling when the problem size is varied to better suit the amount of available compute resources. Three different population sizes (144, 576, and 1296) were permuted with problem sizes ranging from 2000 to 10000, MPI ranks in the set {4, 9, 16} and OpenMP threads in the set {1, 3, 5, 7} to demonstrate this.

For greater amounts of compute resources, larger workloads (read, larger population and problem sizes) had the lowest runtimes, which is in line with the idea of weak scaling put forth by Gustafson's Law. Ideally, the results produced would have been extended with even larger workloads and an increased amount of computer resources, but due to time constraints and a fair few technical difficulties, this was unfortunately not possible.

To close, a few memorable lessons about parallel programming using MPI and OpenMP were learnt throughout the course of the project:

1. MPI is quite difficult and time consuming to actually implement correctly, and requires significant refactoring of serial code.

2. Conversely, adding OpenMP functionality to a program *seems* easy because it really just involves adding compiler directives to existing sections of code, *however*, it can be quite difficult to do this correctly, and requires a good understanding of the overheads associated with multithreading. One lesson that was learnt in the early stages of attempting to add OpenMP functionality to the GA program was that it is a bad idea to introduce parallelism into functions that are called a large number of times, *especially* if the work they do in one call is negligible. This is because the overhead of setting up and synchronising threads heavily outweighs the parallel speedup that is gained. Rather, it is a better idea to try and 'push' parallelism as close to the base of the call stack of the program as possible, and to try and encapsulate as much work in each parallel section as possible.

3. Writing to the same file from multiple threads in a parallel section of code is an *extremely* bad idea! After observing large parallel *slowdown* on a large number of experiments (where the runtime actually decreased worse than linearly when the number of threads was increased), the effects of doing this were made painfully clear. Writing to a file on disk involves acquiring access to a write lock for that file; something which multiple threads will 'fight' over, introducing a large amount of pointless overhead to runtime.

Appendices

Figure 11: Slurm submission script for global pop size MPI + OpenMP scaling experiment set

```
#!/bin/bash -login
#SBATCH --job-name=gen_alg_para
#SBATCH --partition=iceq
#SBATCH --nodes=16
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=7
#SBATCH --time=12:00:00

module load intel
module load intel-mpi

POP_SIZES=(2304 3600)
PROB_NAME="one_max"
TRIALS_PER_PROB_SIZE=5
PROB_SIZES=(12000 14000 16000 18000 20000)
RANK_NUMS=(4 9 16)
THREAD_NUMS=(1 3 5 7)
DEBUG=0
QUIET=1
RESULTS_DIR="mpi_omp_prob_size_scaling"
USE_SHELL_REDIR=0

for pop_size in ${POP_SIZES[@]}; do
    for num_ranks in ${RANK_NUMS[@]}; do
        for num_threads in ${THREAD_NUMS[@]}; do
            export OMP_NUM_THREADS=${num_threads}
            for prob_size in ${PROB_SIZES[@]}; do
                for ((j=0; j < ${TRIALS_PER_PROB_SIZE}; j++)); do
                    TAG="${RESULTS_DIR}-mpi_${num_ranks}-omp_${OMP_NUM_THREADS}-pop_${
                        pop_size}-${PROB_NAME}_${prob_size}-trial_$(($j+1))"
                    echo "mpirun -n ${num_ranks} ../gen_alg.out ${pop_size} ${PROB_NAME}
                        ${prob_size} ${USE_SHELL_REDIR} ${RESULTS_DIR} ${TAG} ${DEBUG} ${
                        QUIET}"
                    mpirun -n ${num_ranks} ../gen_alg.out ${pop_size} ${PROB_NAME} ${
                        prob_size} ${USE_SHELL_REDIR} ${RESULTS_DIR} ${TAG} ${DEBUG} ${
                        QUIET}
                done
            done
        done
    done
done
```

Bibliography

[1] S. Luke, *Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes.* 2013, OCLC: 900622585, ISBN: 978-1-300-54962-8.

[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989, ISBN: 0201157675.

[3] J. Schaffer and L. Eshelman, "On crossover as an evolutionary viable strategy," in *Proceedings of the 4th International Conference on Genetic Algorithms*, R. Belew and L. Booker, Eds., Morgan Kaufmann, 1991, pp. 61–68.

[4] I. Foster. (1995). Designing and Building Parallel Programs: 3.7 - A Refined Communication Cost Model, [Online]. Available: `https://www.mcs.anl.gov/~itf/dbpp/text/node33.html`.

[5] *Cellular evolutionary algorithm*, in *Wikipedia, the Free Encyclopedia*, Page Version ID: 673162067, Jul. 26, 2015. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Cellular_evolutionary_algorithm&oldid=673162067`.