University of Queensland

COSC7502 Major Project - Milestone 2

Genetic Algorithm

Name	**Student Number**
Tristan Burns	42648493

November 6, 2021

Abstract

This report details serial and parallel implementation, verification and performance optimisation of solution to the OneMax problem using a Genetic Algorithm developed in C++.

Chapter 1 has been updated based on feedback from Milestone 1, with changes indicated by a grey change bar (see right).

Chapter 1

Serial Implementation

1.1 Summary

This chapter details the *serial* implementation, verification and performance optimisation of solution to the OneMax problem using a Genetic Algorithm (GA) implemented in C++.

1.2 Implementation

1.2.1 Algorithm

The GA is a type of metaheuristic optimisation algorithm which has biologically inspired selection of candidate solutions [1]. Metaheuristics are algorithms in the field of *stochastic optimisation*, which, broadly speaking, deals with optimisation problems where little is knowns about the search space, and where where high performance solutions are costly, difficult or impossible to determine a priori. Metaheuristics tend to make relatively few assumptions about the optimization problem making them broadly applicable [2].

GAs search a space of possible solutions by evolving a population through selective *crossover* and random *mutation* of individual solutions. Iteratively, pairs of solutions are selected as *parents*. Parts of the parent solutions are combined for form child solutions. The improvement of solutions arises by implementing *random* selection of the parent solutions, in some way *weighted* by their performance, referred to as its *fitness*. However, GAs are also able to avoid becoming stuck in local optima because of the possibility of selecting poor solutions and the random mutation permits the population to 'evolve' out of local optima [3].

More specifically, solutions are generally vectors, referred to here as *individuals*. Each element of the vector is a *gene*. Although there are implementations with vectors of continuous variables (i.e. floats or doubles), commonly genes are boolean, and the individual a bitwise vector. In this work, each individual has one *chromosome* of length n genes, equal to he length of the solution vector.

The GA starts with a *population* of individuals with randomly initialised genes. The algorithm implemented for this project initialises a $n \times m$ matrix, p, representing the population of individuals in the current generation.

An empty $n \times m$ matrix, q, is also initialised to store new *child solutions*. To generate new child solutions, pairs of individuals are selected as *parents* using *Tournament selection*. Tournament selection returns the fittest individual of some tournament size, t, individuals randomly selected, with replacement, from the population. In this work the tournament size is set to $t = 2$, stated to be a popular setting in [1].

These parents are then used to generate child solutions using single-point *crossover*. In the single-point crossover, the chromosomes of two parent solutions are swapped before and after a randomly selected point [3], producing two children with different segments of each parents chromosomes. Finally, in the *mutation* step, each gene in the child solutions is subject to a bit flip, with a probability of $1/n$.

Fitness evaluation, selection, crossover and mutation is repeated until the desired solution is reached or the maximum permitted number of iterations,*maxgenerations*, is exceeded. Pseudo-code for the algorithm implemented in this project is presented in Algorithm 1.

Algorithm 1 The Genetic Algorithm

```
$n \leftarrow$ Chromosome Size
$m \leftarrow$ Population Size
$p \leftarrow \{\}$                                          ▷ Population of $m$ individuals with chromosomes of length $n$
$q \leftarrow \{\}$                                          ▷ Children of $m$ individuals with chromosomes of length $n$
$maxgenerations \leftarrow$ Max generations.
$best \leftarrow$ null
$generation \leftarrow 0$
$fitness \leftarrow \{\}$
for $m$ times do
    $p \leftarrow$ random individual                         ▷ Define a random seed to permit deterministic behaviour
end for
while $best \neq n$ and $generations < maxgenerations$ do
    for each $i$ in $p$ do
        $fitness[i] \leftarrow IndividualFitness(i)$
    end for
    $best = \max(fitness)$
    for $m/2$ times do
        Parents $p_1, p_2 \leftarrow select(p)$              ▷ Use tournament selection in this implementation
        Children $c_1, c_2 \leftarrow crossover(p)$          ▷ Use single point crossover in this implementation
        $q \leftarrow \{mutate(c_1), mutate(c_1)\}$          ▷ Use bitflip mutation in this implementation
    end for
    $p \leftarrow q$
    $generation + +$
end while
return $best$
```

1.2.2 OneMax Problem

The OneMax problem is defined as the maximum number of value 1 bits in a bit string. Solution of the OneMax problem is effectively searching for a bitsting of exclusively ones.

For the GA this is encoded in the chromosome, with possible genes of 0 or 1. In this implementation, a GA is used to search through a population of m individuals with chromosomes of length n for a chromosome consisting of only ones. In this case a fitness function is defined to be the sum of all chromosomes, with optimal fitness being equal to the chromosome sum n.

1.2.3 Program Implementation

The GA was implemented in C++ and compiled to C++11 standard using g++. Optimisation flags are discussed in Section 4.3 below.

The program is structured to use of macros for user defined constants and static arrays with memory allocated at compile time. In order to maximise performance, verbose functions were duplicates of the non-verbose counterparts, to avoid passing verbose arguments with subsequent internal branching.

1.3 Verification

The implementation was verified through explicit inspection of verbose output results and plotting the (normalised fitness over time) for a number of trials. An advantage of the OneMax problem is that the optimal solution and associated fitness is the same as the chromosome size n, permitting easy verification. Because of the stochastic nature of GAs a random seed was defined as an input parameter to enable deterministic and repeatable results.

Verbose output allowed for explicit verification of the key steps of the GA, namely computing fitness, *crossover* and *mutation* verification. See Annex A for a single loop of verbose output.

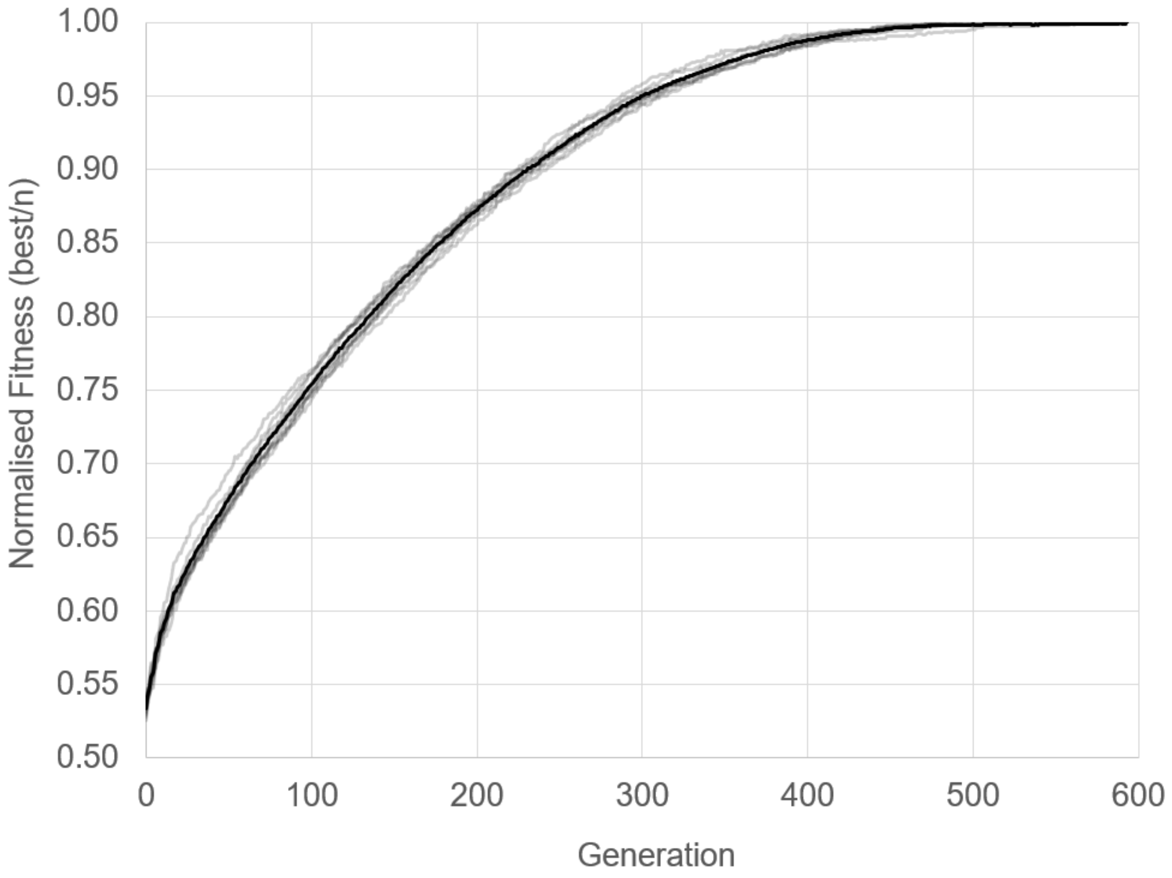


Figure 1.1: Normalised fitness vs number of generations, n, for a population size of $m = 100$ and a chromosome size of $n = 1024$. Different random seeds where used (light grey results), with the average plotted in black, demonstrating robust convergence for large problem size.

1.4 Performance Optimisation

1.4.1 Manual Optimisation

An attempt was made to manually optimise the code based on gprof profiling. The program was compiled with the *-pg -O0* flags set. Based on gprof profiling (see Annex B), the function *Mutate* (see below), was identifed as taking approximately 21 percent of the runtime.

```
void Mutate(int *q, int n, int m)
{
    int mu;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            mu = (std::rand() % n);
            if (mu == 0)
            {
                (q(i, j)) = 1 - (q(i, j));
            }
        }

    }
    return;
}
```

An attempt was made to remove the internal branching as follows.

```
void Mutate(int *q, int n, int m)
{
    int mu;
    for (int i = 0; i < m; i++)
    {

        for (int j = 0; j < n; j++)
        {
            mu = ((std::rand() % n)==0);
            q(i, j) = (q(i, j))*(1-mu) + (1 - (q(i, j)))*(mu); //remove branching
        }

    }
    return;
}
```

However, followup profiling revealed that this actually made performance worse with the function *Mutate* taking approximately 37 percent of the runtime. On reflection, the likely reason the manual optimisation reduced performance was the additional calls to memory required for matrix q. In the case that q is larger than the cache, the additional calls to memory to write values offset any advantage posed by reducing internal branching. Therefore, this implementation was reverted and optimisation was focused on compile flags, as discussed in section 4.2

1.4.2 Optimisation Flags

Because of the number of nested loops in the *Crossover*, *Assess Individual fitness* and *Mutate*, it was inferred that -funroll-loops would yield improvements.

What was unexpected, was the flags -flto and -fuse-linker-plugin in combination yielding a 10-20 percent improvement on -O3 optimisation alone. Although I do not understand this improvement from the documentation, I hypothesise that these flags improve the optimization by exposing more code to the link-time optimizer.

1.4.3 Performance Results

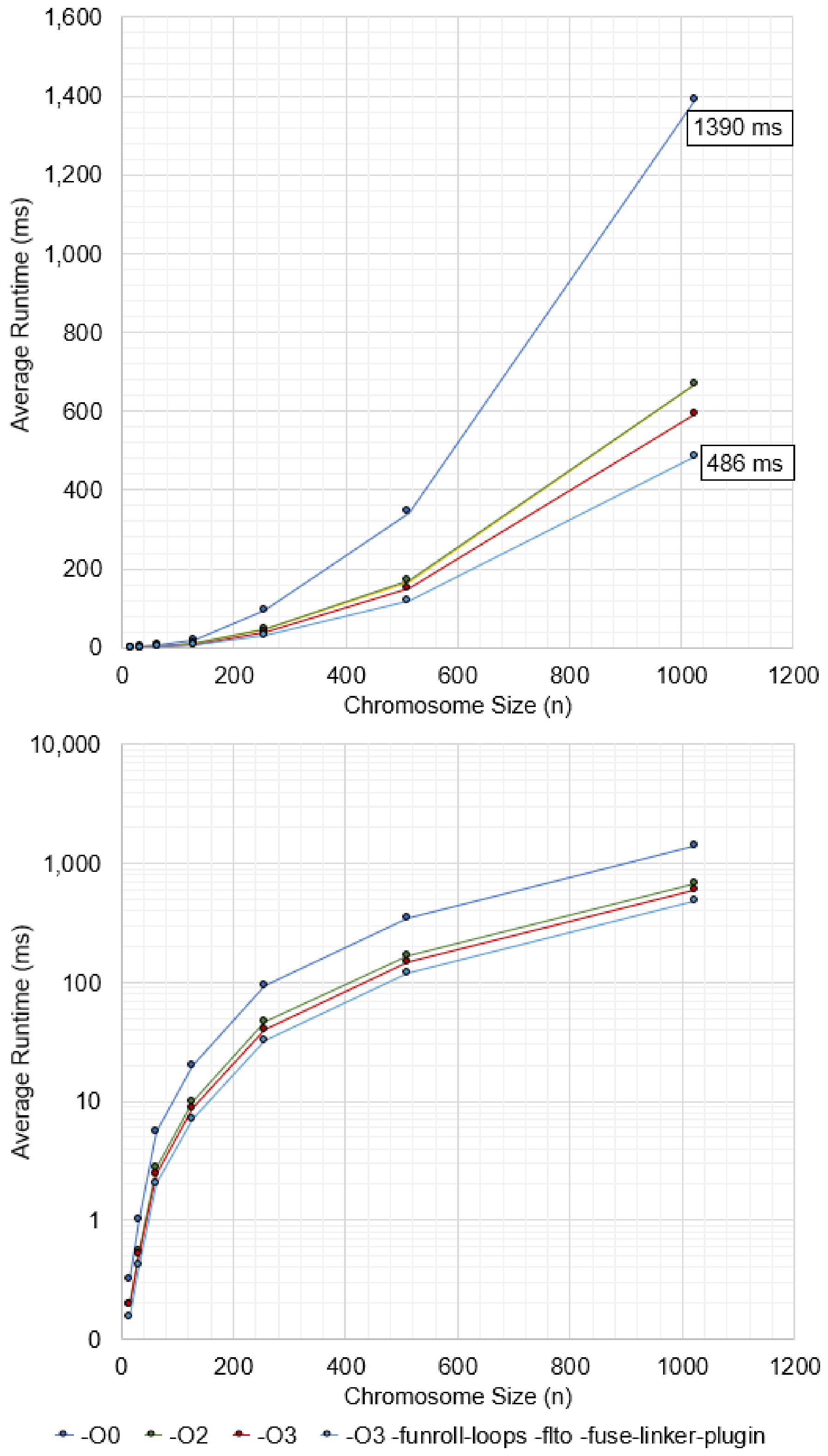


Figure 1.2: Average runtime (ten trials) vs chromosome size, n, for a population size of $m = 100$ individuals. Based on different optimisation flags a $\sim 2.8\times$ speedup is achieved. These results were computed using the same random seed, $seed = 42$.

Chapter 2

Parallel Implementation

2.1 Parallelization Approach

2.1.1 Approach

There are a large variety of ways to parallelize a GA, which are comprehensively discussed in [4]. To describe the parallel GA implementation in this work, the terminology in [4] has been adopted.

The approach to parallelising the GA was to develop a number of *static subpopulations with migration.* The population of individuals, m, is divided into *subpopulations* (also refereed to as *demes*) of size $m_{fraction} = m/k$, processed separately on k *islands* that are *geographically isolated.* In this model, a new operation, *migration*, is added to the GA, which periodically exchanges individuals between islands (see Section 2.1.2).

Reference [4] further categorises this parallelisation approach into *coarse grained algorithms* and *fine grained algorithms.* Coarse grained algorithms have a relatively small number of demes with many individuals, periodically exchanged through migration. Fine grained algorithms divide the population into a larger number of small demes. Inter-deme communication is realised *either* by using a migration operator, or by using overlapping demes.

For this work, MPI was used to parallellise the GA using a coarse grained approach. A coarse grained approach was selected because it allowed the majority of the original codebase developed in Chapter 1 to be reused[1]. Furthermore, it seemed to be the most directly comparable extension to the existing serial code. The parallel implementation

2.1.2 Migration

Migration depends on several parameters such as the topology of the connections between the subpopulations, the number of individuals that migrate (i.e the *migrants*), which individuals migrate and are replaced and the frequency of migration [4]. This work investigates the performance of two different migration topologies in addition to the number of *migrants* as a percentage of the total population. Individuals are randomly selected for migration, with the array representing the migrating individual being copied (i.e. replacing) to the same individual in the subpopulation array at the destination island (per the movement of the coloured arrays in Figure 2.1 (b) and (c)). To limit the number of parameters requiring tuning, the *migration interval* was set to one (i.e. migration is performed every generation). Given the communication overhead for parallelization, this selection is likely not optimal, especially for small problem sizes.

In addition to parallelization of the GA *without* migration, this investigation explored two different migration strategies, referred to here as *ring migration* and *random migration.* These approaches are described with reference

[1]This was an important consideration because the serial GA implementation was verified to be working. A fine grained model would have required extensive revalidation of the *crossover* and *mutation* functions.

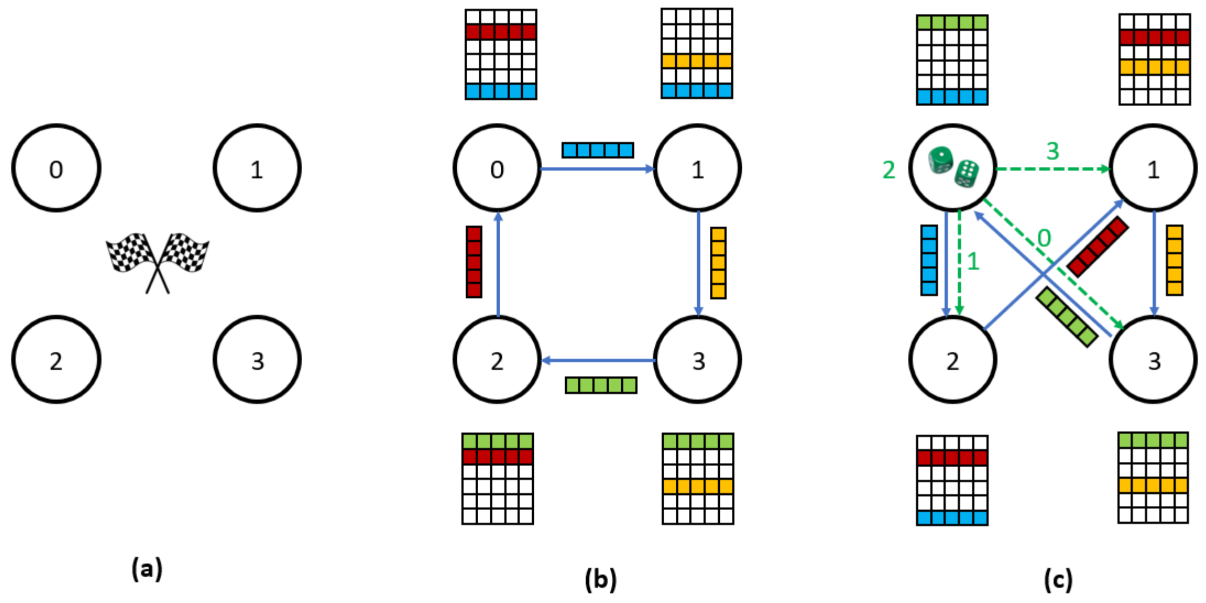


Figure 2.1: Depiction of migration approaches investigated for $k = 4$ islands. Subfigure **(a)** depicts parallelisation *without* migration, which is essentially a race to the maximum fitness individual between each of the subpopulations. Subfigures **(b)** and **(c)** depicts ring migration and random migration, respectively. The blue arrows in subfigures **(b)** and **(c)** represent the direction individuals migrate between islands in a generation. The green dashed arrows and numbers in subfigure **(c)** represents the random target islands (ranks) broadcast by island 0 to all islands, each generation.

to Figure 2.1. Without migration (Figure 2.1 (a)), the implementation is essentially a race between each of the islands to the fittest individual. An advantage of this approach is it's simplicity and low communication overhead, as the only inter-island communication is reporting the highest fitness achieved per generation.

Ring migration (Figure 2.1 (b)) is one possible migration topology which is relatively simple to implement in MPI[2]. Ring migration involves individuals migrating from island rank k to island rank $k + 1$, with the last individual migrating to island rank 0. Although simple to implement, this topology is not biologically/geographically inspired and is somewhat contrived.

Random migration (Figure 2.1 (c)) involves migration from each island to a destination island of a random rank. MPI is based on all ranks having a deterministic set of communication patterns. As such, random communication patterns are somewhat more involved to implement in MPI. Each individual process uses a different random seed, to ensure the generation of distinct subpopulations. However, this means that if each individual process calls `std::rand()` independently to determine a random rank for migration, the processes will not know which island rank to receive communication (i.e. the island they receive migrants from). This implementation averts this difficulty by broadcasting an array of random integers, *rand_rank*, of length *worldsize* from island rank 0 to all islands. The random communication can then be achieved by looping through the k islands, with island k sending and island $\textit{rand\_rank}[k]$ receiving. The array $\textit{rand\_rank}[k]$ is updated and broadcast each generation.

[2]This approach was inspired by the lecture series.

2.2 Experimental Methodology and Plan

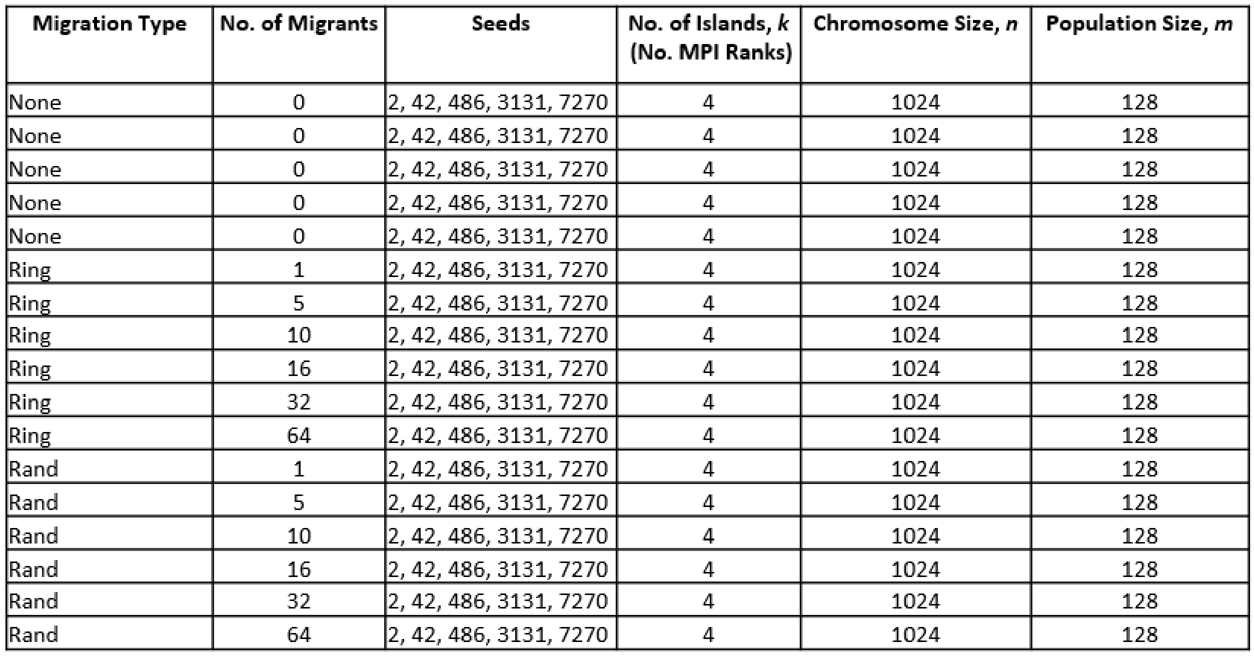
2.2.1 Overview

The experiments planned for this investigation are separated into two parts. Firstly, a study will be made to determine which migration strategy (see Section 2.1.2) is best for the parallel implementation. The results of the migration experiments will inform the migration strategy for the larger scale performance experiments. The performance experiments will compare the performance of the serial and parallel algorithms for a variety of island numbers and chromosome sizes. These experiments are detailed in Sections 2.2.2 and 2.2.3, respectively.

2.2.2 Migration Experiments

Rather than studying the optimality of migration for different problem scales, the objective of the migration investigation is to determine the (likely) best migration strategy to use for the performance experiments. The aim of the migration experiments is to determine 1) whether no migration, ring migration or rand migration offers the best performance and 2) to establish a number of migrants that provides robust performance, as a percentage of the total population.

To limit the scope of this investigation, the migration results are obtained for a fixed $n = 1024$, $m = 128$, and $k = 4$. For each experiment, five trials were performed with different random seeds (see also discussion at Section 2.2.3). The number of migrants was varied from 0 (no migration), 1, 5, 10, 16 and 32. The parameters for each of the migration experiments are specified in Table 2.1, below.



Migration Type	No. of Migrants	Seeds	No. of Islands, *k* (No. MPI Ranks)	Chromosome Size, *n*	Population Size, *m*
None	0	2, 42, 486, 3131, 7270	4	1024	128
None	0	2, 42, 486, 3131, 7270	4	1024	128
None	0	2, 42, 486, 3131, 7270	4	1024	128
None	0	2, 42, 486, 3131, 7270	4	1024	128
None	0	2, 42, 486, 3131, 7270	4	1024	128
Ring	1	2, 42, 486, 3131, 7270	4	1024	128
Ring	5	2, 42, 486, 3131, 7270	4	1024	128
Ring	10	2, 42, 486, 3131, 7270	4	1024	128
Ring	16	2, 42, 486, 3131, 7270	4	1024	128
Ring	32	2, 42, 486, 3131, 7270	4	1024	128
Ring	64	2, 42, 486, 3131, 7270	4	1024	128
Rand	1	2, 42, 486, 3131, 7270	4	1024	128
Rand	5	2, 42, 486, 3131, 7270	4	1024	128
Rand	10	2, 42, 486, 3131, 7270	4	1024	128
Rand	16	2, 42, 486, 3131, 7270	4	1024	128
Rand	32	2, 42, 486, 3131, 7270	4	1024	128
Rand	64	2, 42, 486, 3131, 7270	4	1024	128

Table 2.1: Migration experiment parameters.

2.2.3 Performance Experiments

The aim of the performance experiments is to determine how the runtime, T, varies with chromosome size, n, for a number of *worldsizes* (i.e. no. islands, equivalent to MPI rank). The objective is to compare the scaling of the parallel implementation with the serial implementation, and to determine the respective empirical complexities.

Based on the results of the migration experiments, it was determined that random migration with a single migrant performed best for population size m (see migration results in Section 2.3.1). Informed by these results, the performance experiments are conducted using random migration for a fixed population size, $m = 128$, and a fixed number of migrants, $migrants = 1$, with the chromosome size varying from $n = 16$ to $n = 32768$ and the number of islands varying from $k = 1$ (serial) to $k = 16$.

Over the course of development it was noticed that the variation in runtime of individual instances with the same random seed tended to be less than the variation due to runs with different random seeds. Therefore, it was decided that varying the random seed better captured the variation in output results possible with the GA than the ten repetitions with the same random seed performed for the serial implementation (see Section 1.4.3). Therefore, for each performance experiment five trials are performed with the random seeds 2, 42, 486, 3131 and 7280.

The parameters for each of the performance experiments are specified in Table 2.2.

2.2.4 Use of Stream Editor (sed)

To automate the submission of jobs to the `getafix` cluster a bash script was developed to vary the input parameters. Because the program was structured to use macros for user defined constants (see Section 1.2.3), changing user parameters required `main.cpp` to be edited and the program recompiled. To enable this process to be automated using bash the stream editor (sed) was used in conjunction with regular expressions (regex). The stream editor can perform basic text transformations on an input stream, such as files. The following sed commands were used extensively to modify migration status, number of migrants, random seed and chromosome size.

```
sed -i '22 s/#define migration true/#define migration false/' main.cpp
sed -i "23 s/#define migrants [0-9]\+/#define migrants ${migrants}/" main.cpp
sed -i "17 s/#define seed [0-9]\+/#define seed ${seed}/" main.cpp
```

The stream editor was also used to modify the `goslurm.sh` slurm submission script for different numbers of islands, as follows

```
sed -i "5 s/#SBATCH --ntasks=[0-9]\+/#SBATCH --ntasks=${islands}/" goslurm.sh
sed -i "6 s/#SBATCH --ntasks-per-node=[0-9]\+/#SBATCH --ntasks-per-node=${islands}/" goslurm.sh
```

The use of sed allowed the large number of performance experiments to be automatically compiled and submitted, without requiring the GA implementation to be modified to parse additional command line arguments. This reduced the likelihood of human error, a significant possibility given upwards of 240 compilations were required for the performance experiments alone[3].

[3]It was also quite satisfying.

No. of Islands, k (No. MPI Ranks)	Chromosome Sizes, n	Seeds	Migration Type	No. of Migrants	Population Size, m
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
1 (Serial)	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
4	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
8	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128
16	16, 32, 64, 128, 256, 512, 2048, 4096, 8192, 16384, 32768	2, 42, 486, 3131, 7270	Rand*	1*	128

Table 2.2: Performance experiment parameters. The * denotes parameters which were selected based on the results of the migration experiments presented in Section 2.3.1.

2.3 Results

2.3.1 Migration Results

The migration results are plotted in Figure 2.2. Figure 2.2 (a) clearly demonstrates that ring migration offers no benefit over no migration (i.e. migrant percentage of zero)), with runtime increasing approximately linearly with migrant percentage. However, Figure 2.2 (b) clearly demonstrates that for low migrant percentages of approximately 1-5 percent, random migration offers approximately a 30-35 percent improvement in runtime.

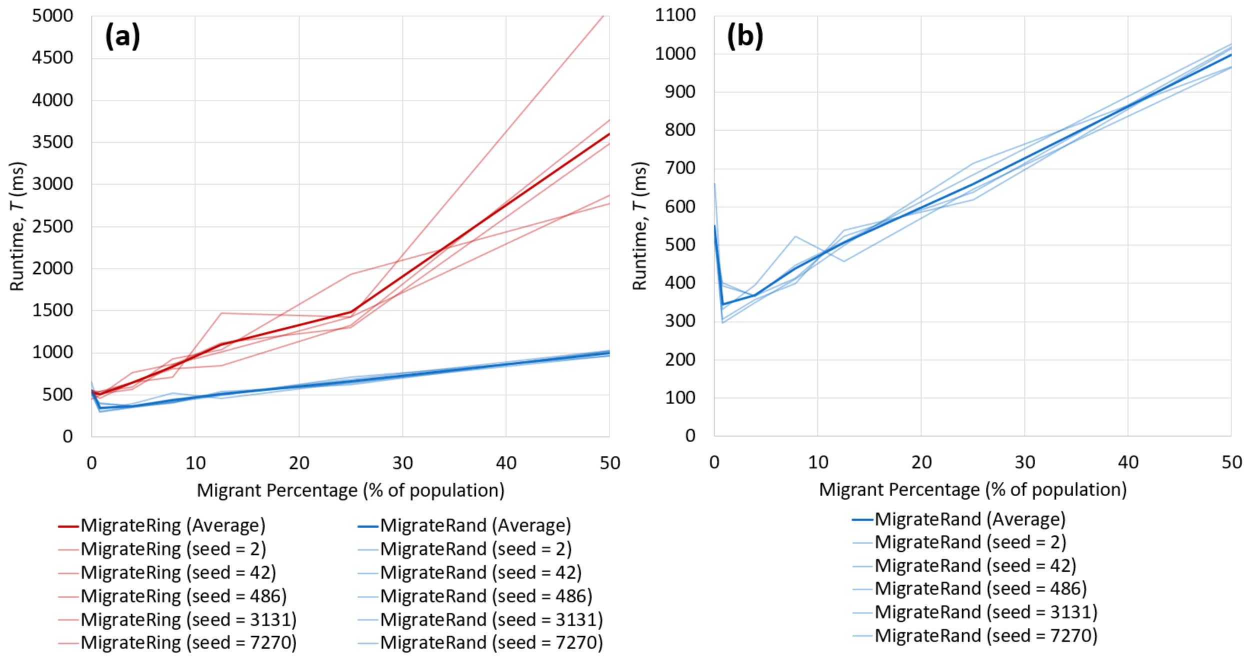


Figure 2.2: Migration results for both ring migration and random migration are presented in subfigure **(a)**. Subfigure **(b)** zooms in on the random migration results.

2.3.2 Performance Results

The runtime as a function of chromosome size is plotted in Figure 2.3. Figure 2.3 clearly demonstrates performance improvements for increasing numbers of islands. Figure 2.3 also demonstrates that the variance in runtime associated with the different random seeds is significantly lower for the 16 island experiment, compared with the other experiments.

As discussed by Coffin and Saltzman in [6], if we parametrise the runtime, T, by the chromosome size, n, as

$$T(n) = Cn^k,$$

where C and k are regression parameters. By taking a regression of the log-transformed model, the gradient estimates the algorithm's order of the computational complexity, $O(n^k)$, the *empirical complexity*,

$$\ln T(n) = \ln C + k \ln n.$$

A log regression is performed for the serial results in Figure 2.4. The log regression gradient of approximately 2 indicates the empirical complexity of the serial GA is $O(n^2)$. The parallel GA tents towards the same gradient for increasing n, indicating the $O(n^2)$ complexity holds in the limit of large n.

Figure 2.4 also demonstrates that for small n the parallel GA performs significantly worse than the serial implementation, due to the communications overhead required to initialise all islands, and check global fitness. This is only apparent in the log scaling of Figure 2.4.

Figure 2.5 demonstrates the average runtime speedup of the parallel GA relative to the serial GA as a function of the number islands for different chromosome sizes n. As with Figure 2.4, we observe that the parallel GA reduces the performance for small problem sizes $n = 64, 256$. Conversely, for the large problem sizes $n = 8192, 32768$, The speedup approaches the limit of linear scaling with additional CPUs. For example, the maximum speedup achieved for

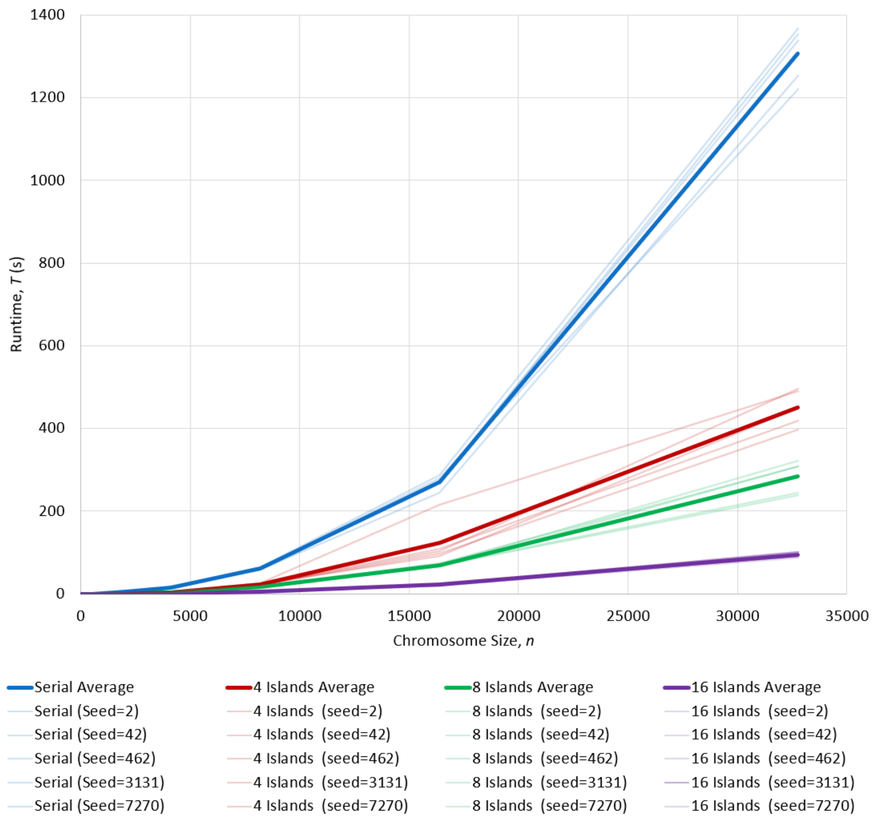


Figure 2.3: Runtime as a function of chromosome size for different numbers of islands (MPI ranks).

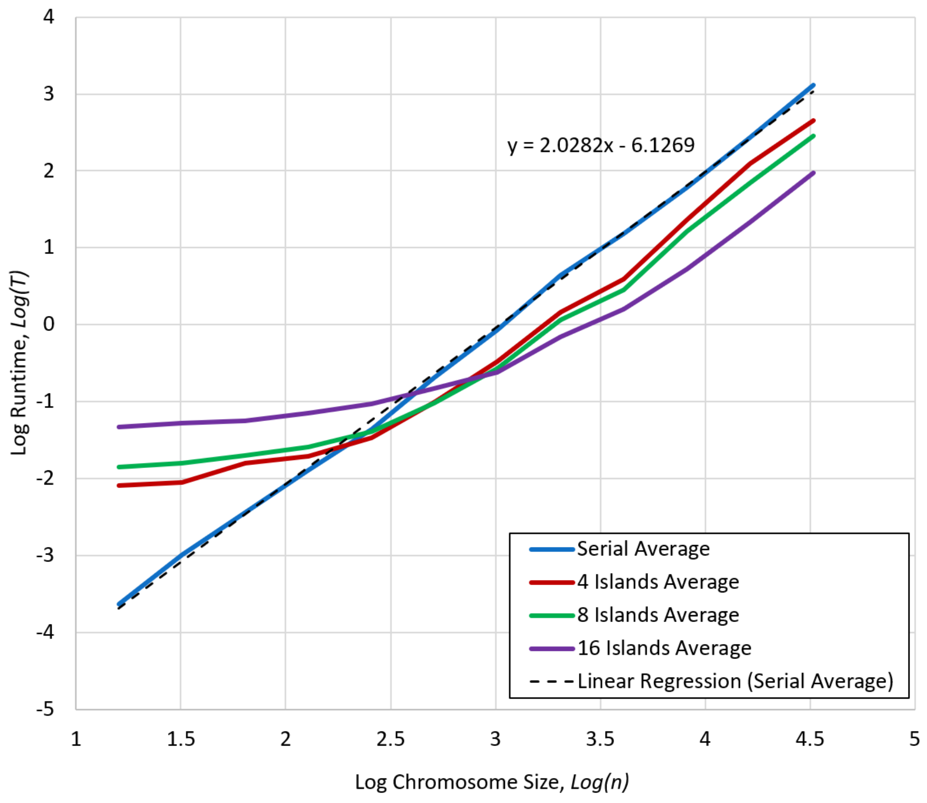


Figure 2.4: Log runtime as a function of log chromosome size for different numbers of islands (MPI ranks). A linear regression of the serial results is included with the equation depicted on the plot.

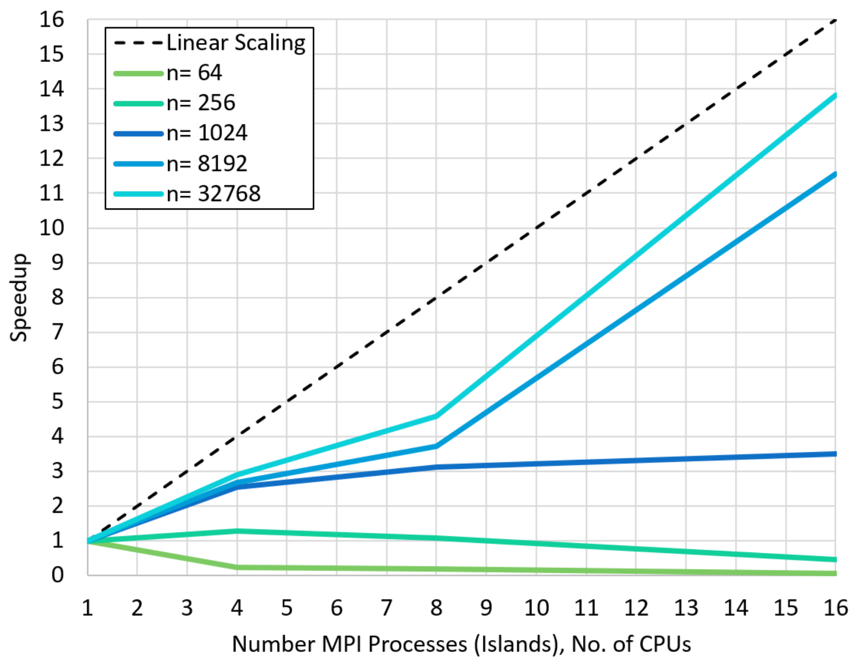


Figure 2.5: Average runtime speedup as a function of number of islands (MPI ranks) for chromosome sizes from $n = 64$ to $n = 32768$.

2.4 Discussion

2.4.1 Outcomes

One of the main findings of this investigation was the superior performance of random migration compared with ring migration. Superficially, this result is somewhat unexpected as the random migration involves and additional MPI broadcast call to distribute *rand_rank* in addition to the send and receive calls conveying the migrating individuals. However, the improvement in convergence rate was observed to more than make up for the additional broadcast call.

Somewhat imprecisely, this can be explained by considering the spread of successful individuals throughout the world. If there is a successful individual on an island through selection and crossover its genes become more prevalent in that population. Therefore, an individual containing at least some of these successful genes is more likely to be randomly selected for migration. In ring migration, these successful genes only travel to an adjacent island, where they must build up a majority to be likely to migrate to the next adjacent island. In random migration, successful genes are sent to an island of random rank, each generation, rather than incrementally populating their way through the loop. With this explanation in mind, it would be interesting to test the two migration strategies on a problem other than OneMax, with more local optima[4]. In this scenario, random migration may favour the propagation of local optima, reducing the diversity required to reach the global optima.

Another finding is that migration only yields an improvement for low migrant numbers, between 1 and 5 percent of the global population. This is understood in the context of the afore explanation, to be due to the penalty of communications overhead. Since more successful solutions tend to be more prevalent throughout a subpopulation at any time, repeated random sampling and propagating yields diminishing returns as similar genes are likely to be propagated, offering little convergence advantage with the same communications overhead.

The the performance experiments were successful in

2.4.2 Limitations

Only works for combinations of population size and number of islands that are evenly divisible.

SIngle point vs double point crossover

Other implementations incorporate a version of *elitism* for the migration step.[5] The migration step may occur less frequently than every iteration - this was not studied. More complex migration topologies, such as nearest neighbour, were not explored. The migration interval of one was not optimal, especially for small problem sizes, due to the communcations overhead. Parallel GAs have many control parameters - difficult to devise (efficient) experiments to optimise performance. Somewhat an art. Drawbacks of `std::rand()`

[4]Such as the Rastrigin, Schwefel or Griewank problems in [1]

2.5 Conclusion

This work successfully demonstrated the performance scaling of a parallel GA for the OneMax problem, using islands of subpopulations with migration. The final implementation was informed by a parameter and algorithm study, which determined that random migration of 1-5 percent of the population provided the best performance.

Empirical time complexity of the serial GA was estimated to be $O(n^2)$ using log regression. The parallel GA tents towards the same gradient for increasing n, indicating the $O(n^2)$ complexity holds in the limit of large n.

Bibliography

[1] S. Luke, *Essentials of Metaheuristics.* Lulu, second ed., 2013.

[2] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[3] S. Mirjalili, "Genetic algorithm," in *Evolutionary algorithms and neural networks*, pp. 43–55, Springer, 2019.

[4] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," in *1999 Third International Conference on Knowledge-Based Intelligent Information Engineering Systems. Proceedings (Cat. No. 99TH8410)*, pp. 88–92, Ieee, 1999.

[5] H. Mühlenbein, M. Schomisch, and J. Born, "The parallel genetic algorithm as function optimizer," *Parallel computing*, vol. 17, no. 6-7, pp. 619–632, 1991.

[6] M. Coffin and M. J. Saltzman, "Statistical analysis of computational tests of algorithms and heuristics," *INFORMS Journal on Computing*, vol. 12, no. 1, pp. 24–44, 2000.

Annexes

Annex A - Serial Verbose Example

Using the following inputs, the verbose output for a single loop is displayed as an example.

```
#define n 16                  // Chromosome size (number of genes per individual).
#define m 8                   // Population size (number of individuals).
#define seed 42               // Psuedorandom number generator seed (std:srand(seed)).
#define maxgenerations 2000   // Maximum number of generations (while loop limit).
#define t 2                   // Tournament size (parents competing for selection).
#define verbose true          // Verbose output (cout) for verification.
#define printfitness false    // Best fitness per generation output (cout).
```

Single loop (generation) verbose output (cout):

```
Running MaxFitness (Verbose)
0011001011011000 -> Individual 0, Fitness: 7
0001101111101011 -> Individual 1, Fitness: 10
0000011011101111 -> Individual 2, Fitness: 9
0101010111101110 -> Individual 3, Fitness: 10
1100010101010110 -> Individual 4, Fitness: 8
0110111001000101 -> Individual 5, Fitness: 8
0010010000011111 -> Individual 6, Fitness: 7
0101011101100100 -> Individual 7, Fitness: 8
 ------------ Generation: 0 Best Fitness: 10 --------------
Running Crossover (Verbose)
Crossover at chromosome 7:
0001101|111101011 -> Parent 1 (Individual 1)
0110111|001000101 -> Parent 2 (Individual 5)
0001101|001000101 -> Child 0
0110111|111101011 -> Child 1
Crossover at chromosome 7:
0110111|001000101 -> Parent 1 (Individual 5)
1100010|101010110 -> Parent 2 (Individual 4)
0110111|101010110 -> Child 2
1100010|101000101 -> Child 3
Crossover at chromosome 15:
000110111110101|1 -> Parent 1 (Individual 1)
010101011110111|0 -> Parent 2 (Individual 3)
000110111110101|0 -> Child 4
010101011110111|1 -> Child 5
Crossover at chromosome 0:
|0001101111101011 -> Parent 1 (Individual 1)
|0001101111101011 -> Parent 2 (Individual 1)
|0001101111101011 -> Child 6
|0001101111101011 -> Child 7
Running Mutate (Verbose)
 0  0  0  1  1  0  1 *1* 0  1  0  0  0 *0* 0  1 -> Child 0 Mutation at *_*
 0  1  1  0  1  1  1  1  1  1  1  0  1  0  1  1
 0  1  1  0  1  1  1  1 *1* 1  0  1  0  1  1  0 -> Child 2 Mutation at *_*
 1 *0* 0  0  0  1  0  0  0  1 *1* 0  0  1  0  1 -> Child 3 Mutation at *_*
 0  0  0  1  1  0 *0* 1  1  1  1  0  1  0  1  0 -> Child 4 Mutation at *_*
 0  1  0  1  0  1  0  1  1  1  1  0  1  1  1  1
 0  0  0  1  1  0  1  1  1  1  1  0  1  0  1  1
 0  0  0  1  1  0  1  1  1  1  1  0  1  0  1  1
Assigning next generation (Verbose)
0001101101000001 -> Child 0 to Individual 0, Fitness: 6
0110111111101011 -> Child 1 to Individual 1, Fitness: 12
0110111111010110 -> Child 2 to Individual 2, Fitness: 11
1000010001100101 -> Child 3 to Individual 3, Fitness: 6
0001100111101010 -> Child 4 to Individual 4, Fitness: 8
0101010111101111 -> Child 5 to Individual 5, Fitness: 11
0001101111101011 -> Child 6 to Individual 6, Fitness: 10
0001101111101011 -> Child 7 to Individual 7, Fitness: 10
```

Annex B - Serial Profiling Results

Initial gprof results, prior to (attempted) manual optimisation of the function *Mutate*.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.19      0.45     0.45   144600     0.00     0.00  IndividualFitness(int, int*, int, int)
 21.21      0.70     0.25      482     0.52     0.52  Mutate(int*, int, int)
 17.82      0.91     0.21      482     0.44     0.44  NextGeneration(int*, int*, int, int)
 11.88      1.05     0.14      483     0.29     0.29  MaxFitness(int*, int, int)
 10.18      1.17     0.12      482     0.25     1.20  Crossover(int, int*, int*, int, int)
  0.85      1.18     0.01    48200     0.00     0.01  TournamentSelection(int, int*, int, int)
  0.00      1.18     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14ParseArgumentsiPPc
  0.00      1.18     0.00        1     0.00     0.00  _GLOBAL__sub_I_useseed
  0.00      1.18     0.00        1     0.00     0.00  PrintFitness(int*, int)
  0.00      1.18     0.00        1     0.00     0.00  RandomPopulation(int*, int, int)
  0.00      1.18     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(
    int, int)
  0.00      1.18     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(
    int, int)
```

gprof results, after to (attempted) manual optimisation of the function *Mutate*. Subsequently, the initial implmentation of mutate was used.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.98      0.55     0.55      482     1.14     1.14  Mutate(int*, int, int)
 27.62      0.95     0.40   144600     0.00     0.00  IndividualFitness(int, int*, int, int)
 12.43      1.13     0.18      482     0.37     1.20  Crossover(int, int*, int*, int, int)
 10.36      1.28     0.15      483     0.31     0.31  MaxFitness(int*, int, int)
  8.98      1.41     0.13      482     0.27     0.27  NextGeneration(int*, int*, int, int)
  0.00      1.41     0.00    48200     0.00     0.01  TournamentSelection(int, int*, int, int)
  0.00      1.41     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14ParseArgumentsiPPc
  0.00      1.41     0.00        1     0.00     0.00  _GLOBAL__sub_I_useseed
  0.00      1.41     0.00        1     0.00     0.00  PrintFitness(int*, int)
  0.00      1.41     0.00        1     0.00     0.00  RandomPopulation(int*, int, int)
  0.00      1.41     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(
    int, int)
  0.00      1.41     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(
    int, int)
```

Annex C - Shell Scripts to Automate slurm Submission

```bash
islands=$1
migrants=1

sed -i "5 s/#SBATCH --ntasks=[0-9]\+/#SBATCH --ntasks=${islands}/" goslurm.sh
sed -i "6 s/#SBATCH --ntasks-per-node=[0-9]\+/#SBATCH --ntasks-per-node=${islands}/" goslurm.sh

#2048 4036 8192 16384 32768
#16 32 64 128 256 512 1024

for seed in 42 3131 462 2 7270
do
    for sizen in 16 32 64 128 256 512 1024
    do
        outputname="Parallel_${migrants}_${sizen}_${seed}_${islands}"
        echo ${outputname}
        sed -i "3 s/#SBATCH --job-name=\(.*\)/#SBATCH --job-name=${outputname}/" goslurm.sh
        sed -i "17 s/#define seed [0-9]\+/#define seed ${seed}/" main.cpp
        sed -i "15 s/#define n [0-9]\+/#define n ${sizen}/" main.cpp
        if [ $migrants -eq 0 ]
        then
            sed -i '22 s/#define migration true/#define migration false/' main.cpp
            sed -i "23 s/#define migrants [0-9]\+/#define migrants ${migrants}/" main.cpp
        else
            sed -i '22 s/#define migration false/#define migration true/' main.cpp
            sed -i "23 s/#define migrants [0-9]\+/#define migrants ${migrants}/" main.cpp
        fi
       mpic++ -std=c++11 -g -O3 -funroll-loops -flto -fuse-linker-plugin functions.cpp main.cpp -
    o ${outputname}
       sleep 6
       sbatch goslurm.sh ${islands}
       sleep 2
    done
done
```