A Parallelized Framework for Evolutionary Computation

Geoffrey Saxton Long (*260403840*)
McGill University, Quebec
Geoffrey.Long@mail.mcgill.ca

Abstract

Evolutionary algorithms are a common approach to problems with indeterminate strategies or lengthy computation times when exact results are not necessary. The goal of this project is to implement an extensible framework which allows for parallelization of an evolutionary algorithm. Although computation speed is a primary goal, I would also like to see the outcomes where "populations" of individuals are allowed to evolve in partial, or complete, isolation from one another. Each one of these populations would be implemented on a multithreaded Beowulf cluster, exposure to other populations would occur via MPI message passing. Within each cluster the populations would be evolved through different mutation, crossover, and fitness evaluation methods. This variance in operators would ensure that the populations diverge.

This framework will implement a genetic algorithm. Although the algorithm will be tested with the Travelling Salesperson Algorithm, it will be designed to work with a wide variety of problems. The overall performance of the framework will be evaluated on the results and speedup compared to the sequential version.

1. Introduction

Evolutionary Computation is a branch of computational intelligence commonly used to solve problems with multiple parameters or complex relationships between the parameters, multiple local optima, or no known approach to solving the problem. The term Evolutionary computation covers several different implementations. These are evolutionary programming, genetic programming, genetic algorithms, and evolution strategies. Although the approaches to each of these frameworks is slightly different, they all have the same general structure and themes. Central to all of them is the adherance to Darwinian principles.

The darwinian principles central to evolutionary computation are those of evolution by natural selection. This is commonly broken into four main themes:

1. More individuals are produced each generation than can survive
2. Variation exists amoung individuals, this variation is inhertiable
3. Those individuals with inherited traits better suited to the environment will survive
4. When reproductive isolation occurs a new species will form

survival of the fittest (natural selection), mutation, mating, As the name suggests, Evolutionary Computation methods follow the common themes of evolution as described by Darwin.

1.1. Terminology

In Evolutionary Computation you have a set of *individuals*, also known as a *population*. Each of these individuals is a possible solution to the problem. The way in which the solution is encoded is often unique to the problem and the implementation of the problem. At its core though, the solution must be formed in such a way that it can be changed by the evolutionary operators in the framework.

The evolutionary operators most commonly used are *mutation* and *crossover*. Although mutation is present in all evolutionary computation methods, crossover is often specific to genetic algorithms. Mutation is the alteration of an individual

via a slight change in their genetic makeup. This typically happens by changing a value of one of the individual's alleles or by switching two or more alleles. As a result, the resultant individual after mutation is slightly different than the original. Crossover involves the creation of a new individual by the combination of two "parent" individuals. Parents are often selected by their genetic fitness, though selection can be random. These crossover operators usually involve combining the two parents to create one or more new individuals.

2. Background

3. Implementation

3.1. v1.1

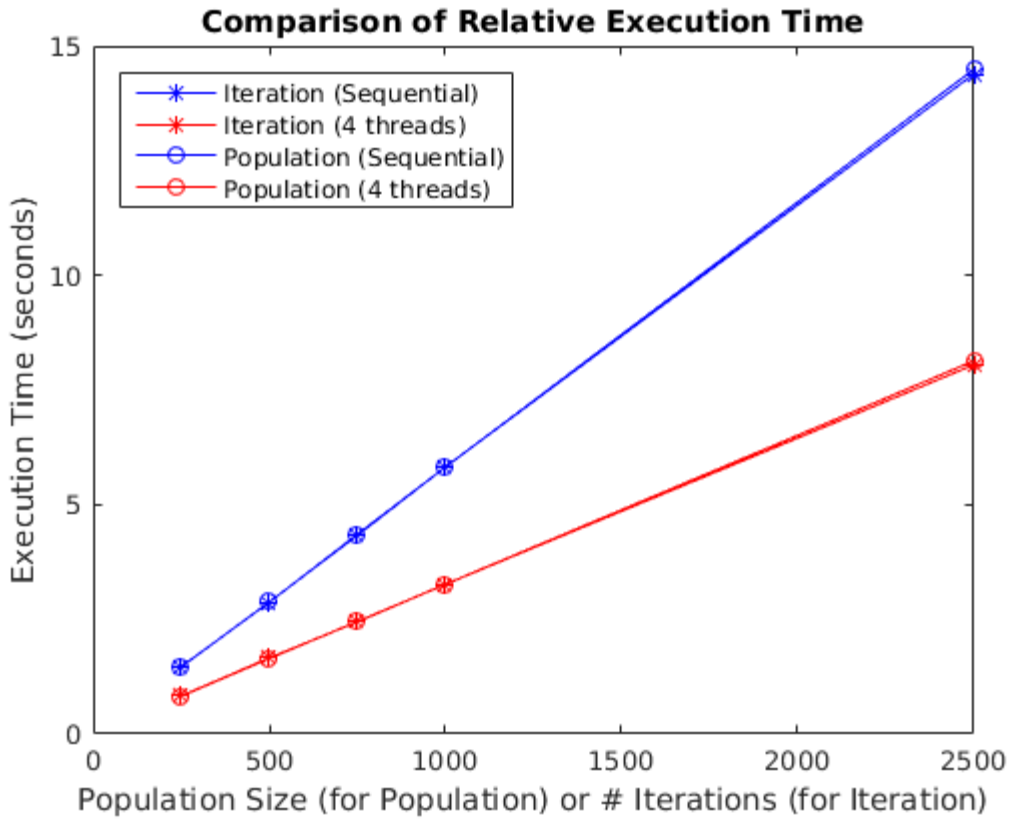
The first iteration of the genetic algorithm framework was more of an evolutionary programming method. In this, there is no crossover. Each parent generates its own offspring. The parent is then compared to it's offspring, and the fitter of the two is allowed to survive.

The purpose of this version was to do some easy characterization of the relationships between the number of threads, the population size, the number of iterations, and the fitness of the fittest individual. By finding relationships between these parameters, futher testing could be constrained so that only the most important behaviour is focused on.

By manual inpsection of the test data, we can see that the fitness level of the individual is uncorrelated with the number of threads being run. Therefore, when discussing the efficacy of the parallel implementation, the fitness score can be largely left out. We can also see that as population size increases, the fitness score becomes better as well. The correlation here is not as dramatic as the correlation between number of iterations and fitness though.

The relationship between population size, maximum number of iterations, and execution time seems to be linear. The result of doubling either the maximum number of iterations or the population size seems to be a doubling in time. So population size and number of iterations seem to be directly proportional in their effect on execution time.

Since we can take the population size and number of iterations to be roughly equal in their implact on execution time, we can easily see which has a larger impact on execution time. Based on graphical analysis of the relative speedup when varying population or the number of iterations independently, it can be seen that the number of threads impact both the same.



When the number of iterations and the population size are both much greater than the number of threads, a change in the population size or the number of iterations will have generally the same effect. However, as the number of threads approaches the population size, the speedup decreases.

If the number of iterations and the population size is sufficiently high, I get an average speedup of 1.68 on a Lenovo ideapad with an Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz. This processor has two cores with two threads per core. With hyperthreading this creates four virtual cores. The peak speedup occurs at 2 threads, and again at 4 threads. Three threads sees a bit of a lull. This may be because of OpenMP's thread migration. Two threads is really all the computer can fully utilize though, since hyperthreading doesn't really help for cases such as this where there is a lot of floating point computation.

Since only having two cores was severly limiting, I got access to a cim server called Debondt. This server has

Armed with Debondt, I was able to then do some testing in parallel, which sped up my results. From here, I found the ideal number for population size and maximum number of iterations. This was found by running the program on the Lenovo with a set number of threads and a varying population size and iteration number. From this, I found that a population size of 500 with an iteration number of 10000 was ideal for futher testing. These number maximized the speedup and minimized the overall execution time while still providing a good final fitness. These sizes were then run on Debondt and the Lenovo with variable thread counts to characterize the speedup. From these trials I got the following graphs.

4. Results

5. Conclusions

6. Pertinent Results

6.1. Simple Sequential

6.1.1 Population Based

PopulationSize=100 maxNumIterations=100 Fitness=1006.61 Time=0.0594495 PopulationSize=200 maxNumIterations=100 Fitness=971.531 Time=0.116636 PopulationSize=300 maxNumIterations=100 Fitness=983.57 Time=0.179451 PopulationSize=400 maxNumIterations=100 Fitness=940.753 Time=0.23334 PopulationSize=500 maxNumIterations=100 Fitness=985.453 Time=0.292069 PopulationSize=600 maxNumIterations=100 Fitness=995.133 Time=0.353345 PopulationSize=700 maxNumIterations=100 Fitness=976.031 Time=0.438044 PopulationSize=800 maxNumIterations=100 Fitness=959.766 Time=0.466137 PopulationSize=900 maxNumIterations=100 Fitness=969.391 Time=0.522619 PopulationSize=1000 maxNumIterations=100 Fitness=972.207 Time=0.585324 PopulationSize=2000 maxNumIterations=100 Fitness=942.047 Time=1.17845 PopulationSize=3000 maxNumIterations=100 Fitness=938.467 Time=1.88575 PopulationSize=4000 maxNumIterations=100 Fitness=940.066 Time=2.37096 PopulationSize=5000 maxNumIterations=100 Fitness=927.209 Time=3.02321 PopulationSize=6000 maxNumIterations=100 Fitness=920.407 Time=3.73201 PopulationSize=7000 maxNumIterations=100 Fitness=919.812 Time=4.54565 PopulationSize=8000 maxNumIterations=100 Fitness=922.916 Time=5.10203 PopulationSize=9000 maxNumIterations=100 Fitness=929.764 Time=5.55095 PopulationSize=10000 maxNumIterations=100 Fitness=925.476 Time=6.07093

6.1.2 Iteration Based

PopulationSize=100 maxNumIterations=100 Fitness=1005.18 Time=0.0612377 PopulationSize=100 maxNumIterations=200 Fitness=879.523 Time=0.117883 PopulationSize=100 maxNumIterations=300 Fitness=800.886 Time=0.174866 PopulationSize=100 maxNumIterations=400 Fitness=791.031 Time=0.237152 PopulationSize=100 maxNumIterations=500 Fitness=716.329 Time=0.292422 PopulationSize=100 maxNumIterations=600 Fitness=700.984 Time=0.351435 PopulationSize=100 maxNumIterations=700 Fitness=698.819 Time=0.41049 PopulationSize=100 maxNumIterations=800 Fitness=685.63 Time=0.476722 PopulationSize=100 maxNumIterations=900 Fitness=667.82 Time=0.530665 PopulationSize=100 maxNumIterations=1000 Fitness=650.368 Time=0.59416 PopulationSize=100 maxNumIterations=2000 Fitness=573.286 Time=1.23812 PopulationSize=100 maxNumIterations=3000 Fitness=534.997 Time=1.79002 PopulationSize=100 maxNumIterations=4000 Fitness=521.264 Time=2.36564 PopulationSize=100 maxNumIterations=5000 Fitness=510.855 Time=2.95029 PopulationSize=100 maxNumIterations=6000 Fitness=503.941 Time=3.781 PopulationSize=100 maxNumIterations=7000 Fitness=505.155 Time=4.3372 PopulationSize=100 maxNumIterations=8000 Fitness=498.558 Time=4.94316 PopulationSize=100 maxNumIterations=9000 Fitness=502.808 Time=5.59283 PopulationSize=100 maxNumIterations=10000 Fitness=497.015 Time=6.0404

6.1.3 Thread Based

Threads=1 PopulationSize=500 maxNumIterations=10000 Fitness=480.492 Time=28.9301 Threads=2 PopulationSize=500 maxNumIterations=10000 Fitness=481.042 Time=17.08 Threads=3 PopulationSize=500 maxNumIterations=10000 Fitness=486.164 Time=19.4246 Threads=4 PopulationSize=500 maxNumIterations=10000 Fitness=481.419 Time=16.1616 Threads=6 PopulationSize=500 maxNumIterations=10000 Fitness=478.596 Time=18.4624 Threads=8 PopulationSize=500 maxNumIterations=10000 Fitness=484.597 Time=18.7737 Threads=16 PopulationSize=500 maxNumIterations=10000 Fitness=484.133 Time=19.2235 Threads=32 PopulationSize=500 maxNumIterations=10000 Fitness=485.378 Time=20.5241 Threads=64 PopulationSize=500 maxNumIterations=10000 Fitness=480.417 Time=20.9859

7. Sources