A Parallelized Framework for Evolutionary Computation

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

1. Introduction

Evolutionary Computation is a branch of computational intelligence commonly used to solve problems with complex relationships between the parameters, multiple local optima, or no known approach to solving the problem. The term Evolutionary computation covers several different algorithmic approaches. These are evolutionary programming, genetic programming, genetic algorithms, and evolution strategies [3]. Although the approaches to each of these frameworks is slightly different, they all have the same general structure and themes; specifically, the adherence 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 [7, 1]:

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

The above can be abstracted easily into programming. To model evolution in a computer environment you need encodings for a set of individuals and operators to create variety within the individuals. Central to evolutionary computation is the *population*. This population is a set of *individuals*, which are each an encoding of a possible solution. The individuals can be analyzed on how well they solve the underlying problem. This analysis is called *fitness evaluation* and the score attributed to each individual is called its *fitness*.

Commonly these individuals are bit strings, however they can have more complex structures. The way in which the individual (solution) is encoded is often unique to the problem and specific implementation. The individual can be optimized to be space efficient, easily manipulated, or easily understood. Generally, the encoding will only be successful if it can be altered 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 certain subsets of evolutionary computation such as genetic algorithms. Both of these operators act on *alleles*, which are the smallest differentiable part of each individual.

Mutation is the alteration of an individual via a slight change in their genetic makeup. This change typically occurs through altering one of the individual's alleles or by switching two or more alleles. The resultant individual is slightly different from the original. The implementation determines how slight this difference is.

Crossover involves the creation of a new individual by combining two or more "parent" individuals. Typically, the parents are selected based on their genetic fitness. The fittest individuals are usually the ones allowed to procreate. When the parents are selected, the crossover operators usually use sections of each to create one or more new individuals.

The individuals in a population are operated on iteratively. Each iteration is called a *generation*, and typically involves both mutations and crossovers. The population at the end of each iteration is usually larger than the previous. The final stage of the iteration is to "kill" off certain individuals so that the population is the same size as it was at the beginning. This is often known as *selection*, and holds closest to the themes of natural selection. The goal of each iteration is to stochastically approach an optimal fitness value. Upon termination of the algorithm, either based on a set number of iterations or on some fitness metric, the top performing individual will be returned as the solution to the problem.

2. Background

In order to test the framework, I chose to implement the travelling salesperson algorithm. In this algorithm you are given an undirected fully connected graph. The edges are each weighted, the weights correspond to the cost of travelling from one node to the next. The goal is to find the least cost cycle that traverses every node at least once.

The travelling salesperson problem (TSP) is an NP-hard problem. As such, not only are we unable to compute the optimal solution in polynomial time, an optimal solution cannot even be verified in polynomial time. The TSP has many applications such as routing shipments and manufacturing goods (drilling holes in materials) [2]. Due to the importance of these applications, the TSP has become one of the most studied problems in computer science.

There are many different ways to approach TSP. There are exact methods such as simplex programming and linear programming methods. However, these suffer from prohibitively long execution times. More commonly, approaches to the TSP are often approximating algorithms. These give reasonable results within a reasonable execution time. These include programs such as optimization, nearest neighbor searching, heuristical approaches, and, of course, evolutionary approaches [6].

3. Implementation

All of the tests were conducted using data from TSPLib [8], specifically the symmetric datasets. All the tests were conducted using one of two machines:

1. Lenovo: 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.

2. Debondt: A McGill CIM server equipped with two Intel(R) Xeon(R) E5645 @ 2.40GHz processors. Each of these have 6 cores (12 with hyperthreading). This results in 12 actual and 24 virtual cores overall.

3.1. Primary version (v1.1)

The first iteration of the genetic algorithm framework was a simple mutation and fitness based selection algorithm. Essentially it consisted of two loops. The outer loop counts the generation number and is responsible for terminating the program after a specific number of iterations. The inner loop encapsulates all of the evolutionary methods. In this for loop, the individual is first mutated via a swap (two cities are selected at random and switched). The mutation returns a new individual, which is compared to the previous (nonmutated) parent. The fitter of the two is selected for the next generation. No crossover was implemented in this version. Since this was the case, this implementation was more of an evolutionary programming method.

The parallelization for this implementation occurs over the population loop. It could not happen over the encapsulating iteration loop as the input of the loop depends on the output of the previous iteration. This linearity in generations is crucial to the evolutionary computation method. The population loop, on the other hand, was parallelizable because the individuals in the population were independent from one another.

The purpose of this version was to characterize the relationships between the various parameters. The parameters compared in this phase were the number of threads, the population size, and the number of iterations. These parameters were evaluated as per their impact on the fitness score of the fittest individual and execution time. The goal was to find relationships between these parameters. By finding relationships, further testing could be constrained by focusing on only the most important parameters.

3.1.1 Parameter Relationships

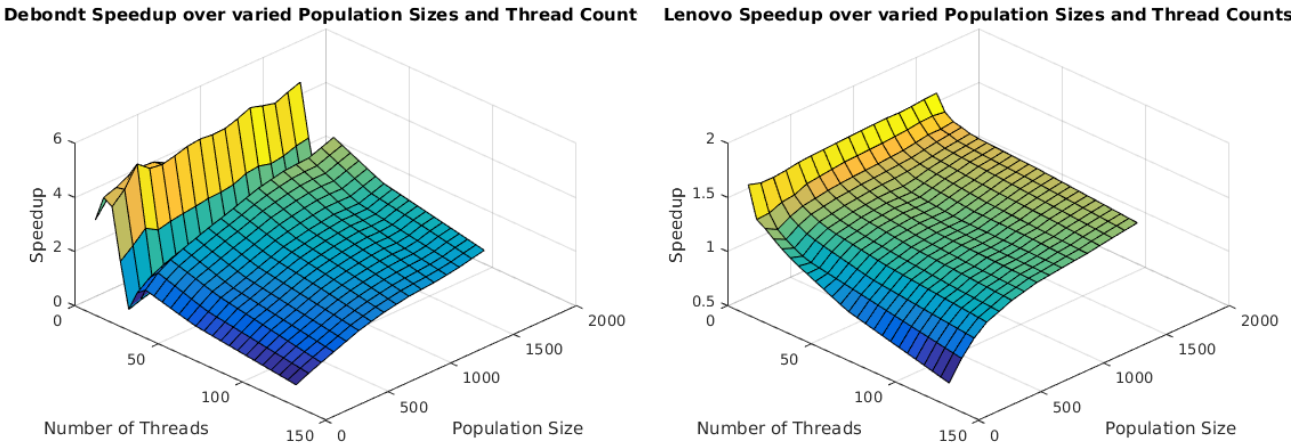
The first tests were focused on analyzing how fitness varied with the parameters. The first parameter focused on was the number of threads. Since the number of threads does not change the underlying computations, the fitness should be unaffected by an increase in the thread count. By manual inspection of the data, this was shown to be true. Since these two are uncorrelated, the fitness score of individuals can be largely left out when discussing the efficacy of the parallel implementation. It was also found that while an increase in population size does improve the fitness of the individual, there is a larger increase in fitness when the number of iterations is bolstered.

Since fitness proved to be rather irrelevant in discussion of the parallel performance, speedup and execution time were focused on next. An almost linear relationship was found between the parameters population size and number of iterations (taken individually) and execution time. Doubling either the maximum number of iterations or the population size almost doubles the execution time. Therefore, 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 impact on execution time, we can view how varying the thread count will impact both. This involved graphical analysis of the relative execution times where population size and number of iterations were varied independently over a range of thread counts. From the graph in Figure 1, it can be seen that the number of threads impact both the same. So, 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 on speedup.

This relationship only holds for small thread counts though. As the number of threads approaches the population size, the speedup decreases quickly. This is shown by the graphs of population size vs thread count (Figure **??**). Although this relationship exists between population size and speedup, there doesn't appear to be any relationship between number of iterations, number of threads, and speedup. This is shown by the graphs in Figure **??**. The reason why the population size has such a large effect is due to how the algorithm is parallelized. The OpenMP parallel construct was placed around a loop over the individuals in a population. When the number of threads increases, each thread is responsible for fewer and fewer individuals. When the number of threads is equal to the population size, each thread is responsible for one loop over

Figure 1: The effect of different thread counts and population/iteration sizes on runtime.



(a) Debondt

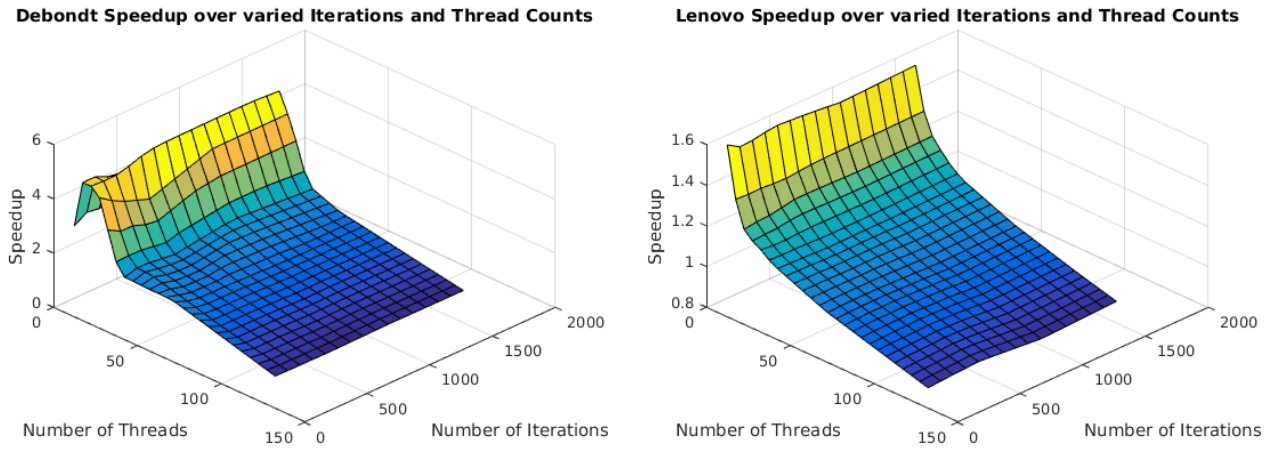
(b) Lenovo

Figure 2: ffect of population size and number of threads on speedup (EIL51).

one individual. The cost of spawning a thread is great when compared to the relatively small amount of work the thread is responsible for. An anomaly worth noting is that for 24 threads, the Debondt system has no speedup at all. This is further discussed in Section 5

So, based on the above, if the population size is sufficiently high, the population size and iteration count can essentially be left out of conversations on speedup. In order to accurately calculate speedup, an ideal number for population size and maximum number of iterations was ascertained. 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 the resultant graph (Figure 4), I found that a population size of 250 with an iteration number of 10000 was ideal for further testing on the Lenovo machine. On Debondt, the ideal population size seemed to be closer to 500 individuals. These parameter values maximized the speedup and minimized the overall execution time while still providing a good final fitness.

The aforementioned sizes were then run on Debondt and the Lenovo with variable thread counts to characterize the speedup. From these trials I got the graphs in Figure 5.



(a) Debondt

(b) Lenovo

Figure 3: Effect of number of iterations and number of threads on speedup (EIL51).

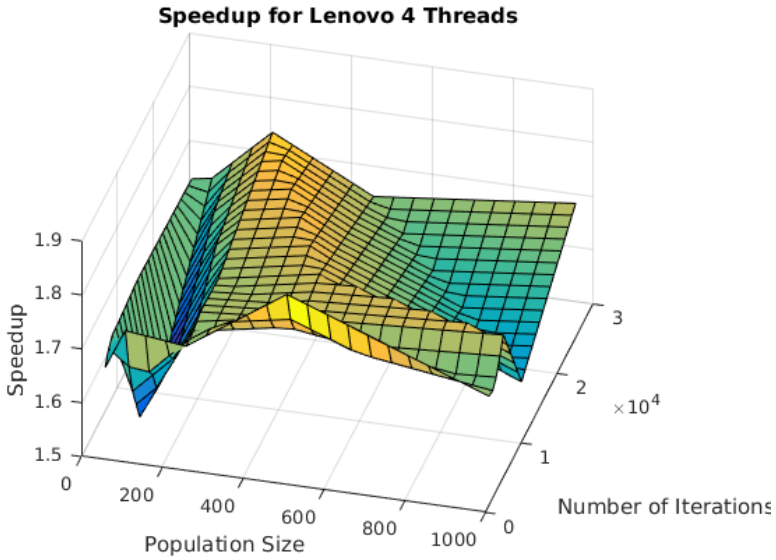
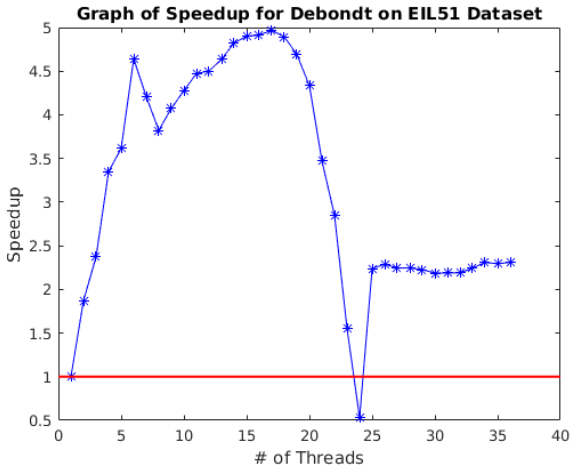


Figure 4: Impact of population size and number of iterations on overall speedup (Lenovo - 4 threads - EIL51).

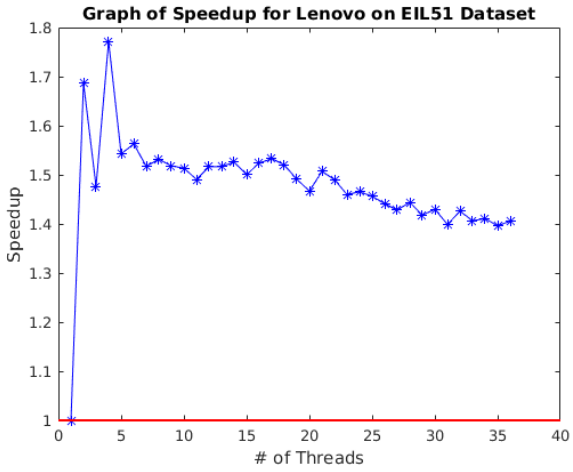
The final trials run on version v1.1 involved determining the effects of tour size on the speedup. Thankfully, I had a few different tours of varied length. These were run using Debondt and the outcome was graphed in Figure 6. As is shown, the larger the tour size, the larger the speedup.

3.2. Two Loop Version (v1.4)

This version was an optimized version of the initial. The first change was to move the `openmp parallel` construct to the outside of the first loop. This reduced overhead since threads did not have to be spawned and killed at every iteration of the outer loop. The second change was the parallelization of population instantiation. This section will take the cities from the tour and create individuals by scrambling the ordering. Previously this was omitted since the initialization was expected to take little time in comparison to the iterative updates on the population. The section was parallelized in this version using another pragma for loop. Since the work inside the for loop involved adding an object to a shared vector, a critical section had to be inserted. The critical section encapsulates the pushing of the individual to the population vector.



(a) Speedup of Debondt over the EIL51 dataset.



(b) Speedup of Lenovo over the EIL51 dataset.

Figure 5: Speedups of Debondt and Lenovo (EIL51 - v1.1

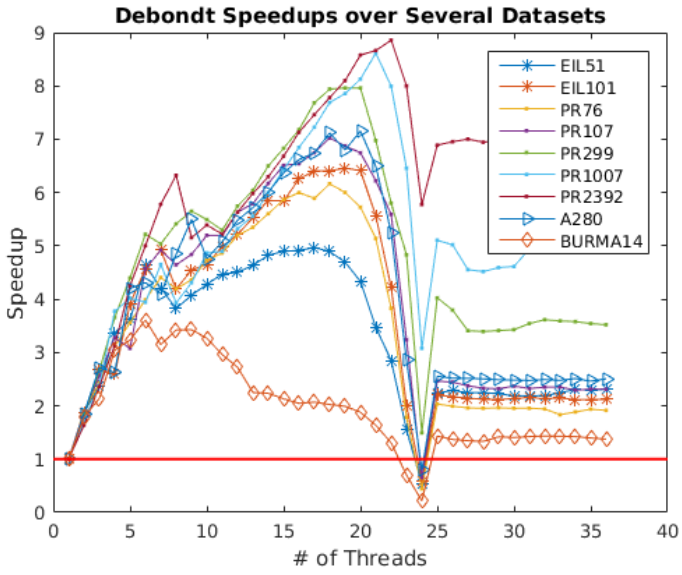


Figure 6: Impact of tour length on overall speedup (Debondt).

The bulk of the parallelization still occurs at the inner population loop as before. Several different loop schedules were attempted for the inner loop, however, the default seemed to work the best for the Lenovo system with four cores. The schedule, static looping over large chunks, works best because each individual should take roughly the same amount of work. It also reduces overhead over the more dynamic scheduling systems.

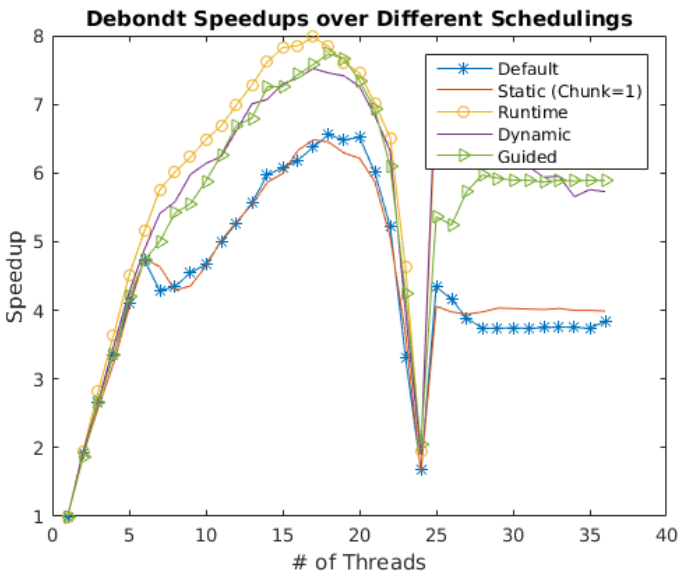


Figure 7: Speedup comparison of different loop scheduling constructs (Debondt - EIL51 - v1.5 Critical).

3.3. Three Loop Version (v1.5)

This version saw an added parallelization for the fittest individual calculations. This loop is performed at the end of the iterative process and will find the fittest individual in the population. Similar to the initialization loop, this calculation was viewed as trivial in comparison to the main iterative loop. Although the increase in speedup was expected to be rather small, the optimization was still performed. Since this loop involves storing the top fitness value in a public object, a critical section needed to be added. Although an atomic operation was deemed to have less overhead [5], it would not work properly for the encapsulated assignment. The critical section was eventually replaced by a reduction clause, since a reduction seemed to have even less overhead than an atomic in addition to being more dynamic [9]. So the parallelization ended up being a parallelized for loop. Each thread will get its own copy of the best fitness value. At the end of the loop openmp will reduce this array of fitness values to find the optimal value. In addition to the added parallelization, the population initialization was optimized. The critical section in the population parallelization was viewed as unnecessary synchronization. The first option was to remove the critical section by directly accessing the indices on writes. I decided not to proceed with this change because it would involve changing the vector datatype and it could introduce false sharing and other unfavorable conditions. Luckily, I found a better alternative. This was to declare an openmp reduction function for merging the vectors created by each individual thread. The resultant vectors from each thread are merged out of order, however, this is actually preferred since the individual creation is supposed to be randomized. Optimizing this loop required an update to GCC 4.9 (OpenMP 4.0), but was otherwise rather simple.

Although this worked on the Lenovo, Debondt did not have GCC 4.9. So, ”v1.5 critical” was created to test this version on Debondt. Unfortunately, this uses the aforementioned critical section, so there is a decent amount of synchronization occurring. With this version, different schedulings were tested. Contrary to what was believed earlier, static is not the best scheduler for this program. Based on Figure 7, runtime is actually the fastest. Although static works best for the Lenovo, Debondt works best with runtime. Static was actually the poorest performing. It is worth noting that this could be because of the critical synchronization.

Also tested on v1.5 Critical were the different mutation operators.

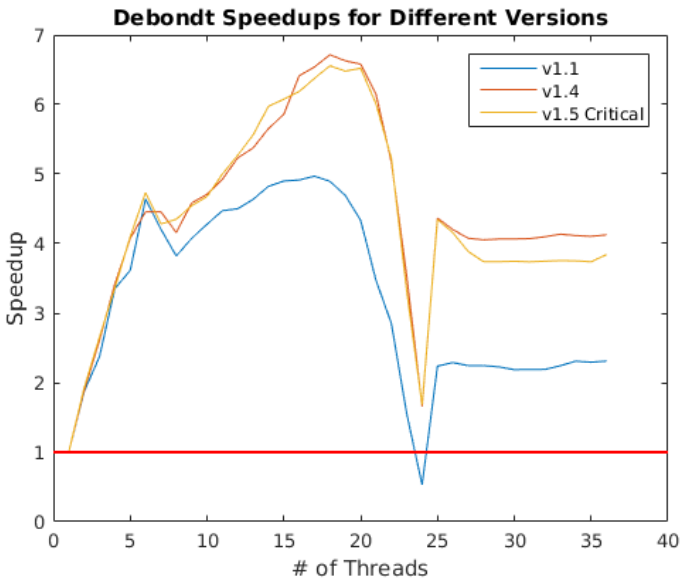


Figure 8: Speedup comparison of versions v1.1, v1.4, and v1.5 critical (Debondt - EIL51).

3.4. Final Version (v1.6)

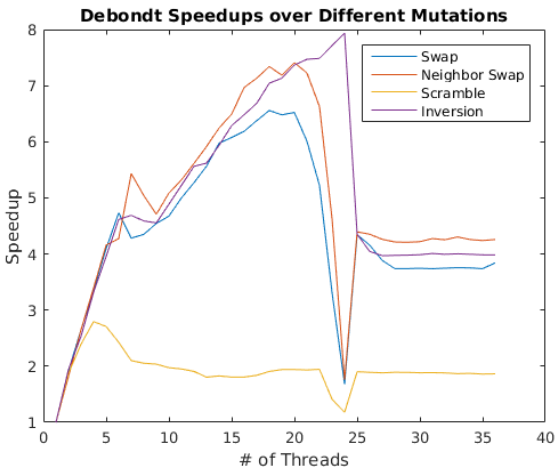
The final optimizations mostly had to do with changing the scheduling for the loops. As was shown before, the schedule that works best is the ”runtime” schedule. The schedule was added to only the population loop. This decision was mostly based on the fact that the population loop is the most dynamic due to the mutation methods. Also, the mutation operator was switched as well. Previously most tests were conducted using the swap mutation, however, inversion was shown to have both the best fitness results and the greatest speedup. The final alteration was preallocating the vectors.

4. Results

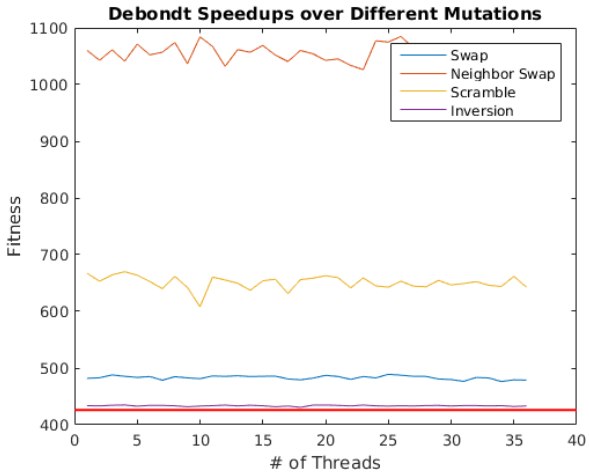
The three iterations were tested with the EIL51 dataset with a population size of 500 and an iteration count of 10000. Unfortunately, version v1.5 could not be tested on Debondt due to compiler issues, so v1.5 critical was used instead. The resulting speedups are shown in Figure 8. As can be seen, the speedup of v1.4 was faster than v1.5 critical. This was likely due to the synchronization overhead of the critical section in v1.5 critical.

The peak speedup for the Debondt server was found to be 8.85 at 22 threads on the pr2392 dataset with a population of 500 and 5000 iterations (version 1.1). On the eil51 dataset with a population of 500 and 1000 iterations the peak speedup was 6.71 at 18 threads (version 1.4). On the eil51 dataset with a dynamic scheduling on the population loop, 500 individuals, and 10000 iterations, the speedup at 12 threads (number of cores) was 6.99, at the peak (17 threads) it was 7.98, and with two threads it was 1.947. A peak speedup of 11 was found at 18 threads on Debondt for v1.5 critical with a dynamic runtime, preallocation of the vectors, a population of 500 and an iteration count of 2500. Although these were the found speedups, there could be a configuration of dataset, population size, and iteration count that yields a higher speedup. As can be seen, a good amount of speedup is achieved, especially proportional to the number of threads with a lower thread count.

The speedup for v1.5 critical run on Debondt with dynamic scheduling on the population, a population size of 500, and an iteration count of 10000 was fit with a few equations. The linear fit for the first 6 threads was found to be $y = 0.8368+0.2528$. The logarithmic fit for the first 17 threads (until the maximum speedup) was found to be $y = 2.6918 * \log(x) + 0.3317$. This is graphed in Figure 10. The logarithmic fit gives a rather good characterization of the speedup for the Debondt system.



(a) Speedup



(b) Fitness (Optimal=426)

Figure 9: Effect of different mutations on Speedup and Fitness (EIL51 - Debondt - v1.5Critical

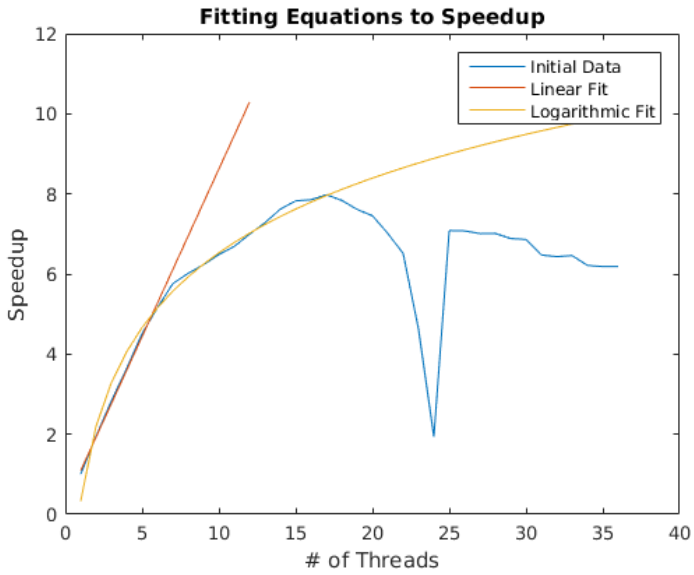


Figure 10: Fitting equations to speedup (Debondt - EIL51 - v1-5crit - runtime scheduling).

5. Discussion

Many of the characterizations of the evolutionary strategy can be seen in Section 3. To summarize the section, the speedup was mostly dependent on the population size and the resultant fitness was mostly dependent on the number of iterations. A suitable relationship between optimal runtime, fitness value, and speedup was found to be a population of 500 individuals and a number of iterations around 10000. Population size had a large effect on speedup since the main parallelization is in a loop over the individuals in a population. As the number of threads and the population size converge, the speedup decreases. Dramatic decreases can be seen when the number of threads reaches roughly half of the population size. At this point, a significant number of threads are left hanging as the others perform work. In addition, as was discussed before, the thread creation overhead increases as the useful work per processor decreases. As a result, the speedup decreases.

Tour size was another parameter that greatly affected speedup. As was shown Figure 6, As the tour size increased, the speedup did as well. This behaviour seemed to be logarithmic with the number of cities in the tour. The speedup dropped off below about 75 cities, and seemed to reach a peak around 1000 cities. This is because the most computationally rigorous section of the code is the fitness evaluation. The fitness evaluation calculates the overall distance travelled in a tour via the euclidean distance formula. Since any tour traverses each city, the fitness evaluation will depend on the number of cities. During the fitness evaluation, each thread is doing useful work. If the tour size is small, then the threads may spend more time looping over the shared index variables. So the proportion of useful work to overhead is greater when the city size is greater.

As can be seen in many of the tests, the speedup on Debondt dropped off rapidly at around 24 threads. As was discussed previously, the Debondt system has 24 threads (virtual cores) from 12 cores. As with the Lenovo, the number of threads is due to hyperthreading. Hyperthreading takes advantage of the architecture of the CPU. Since CPUs typically have different components for floating point calculations, integer calculations, and I/O, different threads can theoretically be using the different sections of the CPU at the same time. This doesn't create extra cores physical cores, but it typically speeds up performance by 10-30%. Although hyperthreading is typically good for everyday computing, it may cause issues in certain parallel and high performance computing applications due to increased memory contention, cache misses, and other performance bottlenecks. Since hyperthreading does not create extra cores, you really only have full support for 12 threads, but speedup over 12 threads can be, and is, achieved [4].

The severe drop-off at 24 threads is likely due to other processes on the system combining poorly with the hyperthread scheduling. When there are fewer than 20 or so threads, there is enough processing power for both the program and the other system processes. As the number of processes approaches 24, it is more likely that the thread will be started on a processor which is already running something else. OpenMP attempts to migrate these threads to optimize runtime through their thread migration and CPU affinity settings. If OpenMP migrates the thread to a different physical processor, there could be cache coherency issues. Alternatively, the thread could be locked to a virtual core. Since this core is only virtual, if a single thread is pushing the physical core to 100% usage then this may be consuming both virtual cores. As a result, the thread could be left waiting for resources. As is shown in Figure 6, which shows the speedup over different tour lengths, longer tours do not suffer as much of a performance hit. This points to thread migration as the likely culprit. Longer tours are more likely to remain on a core through the duration of their work. It would only be during the iteration, when the thread is doing comparatively less work, that it will be migrated. Therefore, shorter tours, which iterate more quickly, will tend to be migrated more often. Alternatively, when the number of threads exceeds the number of cores, the speedup essentially levels out. In this case, thread migration is more likely to occur. This would point to thread binding as the more likely culprit, which is counter to what the earlier data showed. It may be that there is a combination of the two at play. When the number of threads is close, or equal, to the number of virtual cores, thread migration occurs, but is less likely. The result is that thread migration will move the threads to other cores, but the move is suboptimal since almost all of the cores are in use. The result is more suboptimal thread migration. When threads are migrated when all of the virtual cores are in use, each move is essentially equivalent since all the processors will have essentially the same amount of work on them. Although the threads migrate, the migrations are less likely to cause issues. Further testing would involve turning off hyperthreading in addition to changing more of the environment variables, such as `OMP_PROC_BIND` and `GOMP_CPU_Affinity`. This testing was not conducted.

More testing instances can be found on Github under the `Results` subfolder. In (almost) all of the files, the columns are as follows: number of threads, population size, number of iterations, fitness, speedup, (runtime). Though the vast majority of the files follow this format, several of the earlier tests could have slightly different configurations. Each of these instances were averaged over at least five runs, though most of them are averages over ten runs. In addition to the testing instances, the matlab programs for graphing can be found there as well as the initial attempt at a distributed genetic algorithm framework. The plan for the distributed framework was to run it similar to the parallel program, but to have each rank take on a different series of evolutionary operators. From this, I expected to have divergent behaviour on each host, a more varied set of populations, and therefore higher chances of finding an optimal solution.

6. Conclusions

Unfortunately, I was unable to implement everything I had planned for this project. The testing phase took up a considerable portion of the project time. Each test took anywhere between thirty minutes and several hours to complete. Although running each test didn't take too much effort, it was still a bottleneck in the development process. To assuage this, I parallelized my workflow by offloading the testing to the Debondt server. Locally, I would develop new code and test the results on small instances. When a suitable update was created, I would offload it to the server. This way I could develop a new iteration locally while running full tests on the old version remotely. This isn't ideal because I wouldn't fully know the characteristics of the new version until much later. It did allow for quicker release and versioning than testing and developing on the same machine. The main issue with testing in this manner was that I didn't know who else was using the server. Although I believe that I had most of the Debondt's resources, there is a chance that other programs were running during certain tests. This would certainly skew the results.

In hindsight, it would have been good to develop a more formal testing framework. As it was, I simply would update the parameters, run the system, then paste the output into a `.dat` file. Although this worked, it wasn't the most convenient. If I had pushed the output directly to files this would have been quicker. I did manage to formalize the testing slightly by running the majority of the simulations with the same parameter values and data sets. Changing these values for testing wasn't the most intuitive in my system though. With a formalized testing framework I would have been able to vary my tests with greater ease. Developing such a system probably would have taken more time than it was worth though.

Saving the output, then running analysis on the data later was certainly a good idea. Initially I planned on graphing the results immediately. It was only my limited knowledge of C++ that guided me towards matlab. Thankfully though, performing the analysis in this manner meant that I had more control over the graphing process. If the system had output a graph directly without saving the data, I would not have been able to evaluate as many or as complex comparisons. Only by running analysis in this manner was I able to provide the best data visualizations and the most comparisons through the fewest number of test iterations.

If I had more time I would do the following:

- A suitable crossover implementation and a more variable selection method. Currently the framework lacks a crossover algorithm. As such, it is not a true GA. Also, the parent selection is takes place between the initial individual and its mutated offspring, so it is hard to extend to different selection types. The trouble is in extending the population out of the single iterative loop. I have had trouble in manipulating the data while maintaining speedup when trying to create a new population within the current iterative loop. The trouble mainly lies in the vector data type. Since this was my first C++ program, I didn't have enough knowledge of proper construction and data management. I thought that vectors would be a good way to store the data, but now I am unsure. Since the population and individual sizes were known beforehand, it might have been easier to simply use an integer array.

- A visualization for the TSP solution. Initially I had planned on creating a sort of bitmap or heatmap to visually display the individual for intra and inter population comparisons. The plan was to have each index of a matrix be a city in the tour. This index-city pairing would be immutable for the specific tour being run. The value at the index could then be one of several things including the distance from the city at that index to the next city in the tour; the location of the city in the tour; or the distance of that city to a fixed point. There are a great number of ways to generate this visualization, as long as the cities at each index in the matrix are the same across individuals, the visualization should provide a good comparison.

- Make the framework more extensible. Currently the system is only tailored towards the travelling salesperson algorithm, and even as far as that goes it is rather constrained. Ideally, the bulk of the framework would be general evolutionary operators and the bulk of the runtime logic for the evolutionary process. The individual's construction and fitness evaluation could be moved to a separate file. Since only really the individual and the fitness evaluation are implementation specific, the user would only have to alter this file to change the algorithm being evolved.

- Better organization of the code base. Currently, the framework exists as a single God file. Moving the evolutionary operators to a separate file would greatly increase readability.

- Further testing with thread migration, processor binding, and hyperthreading as discussed in Section 5.

In summary, there was a lot more to be done on this project. The initial proposal was too large a task for one single person to take within the timeframe of this project. Perhaps if I had several people, then I could have realized my initial plans more fully. In addition, I spent far more time testing and characterizing the initial versions than I had initially planned. Initially, this project was geared towards creating a parallelized framework to view divergent evolutionary behaviour. As development progressed however, the focus shifted more towards optimizing speedup. Although the current version is lacking in features, it is likely to have far greater speedup than the original plans would have allowed for. Perhaps the remainder of this framework can be a future project of mine.

7. Running the System

The file `parallel_GA.cpp` can be compiled via `g++ -fopenmp parallel_GA.cpp`. It the program fails to compile see `NOTE1` in the code, implement the changes, and recompile. It can then be run by via the command line with `./a.out`. If you would like to try out a different version they are tagged on github. These versions, and all the other files, can be found on `https://github.com/GeoffreyLong/Parallelized_GA`.

References

[1] Natural selection. `http://evolution.berkeley.edu/evolibrary/article/evo_25`. [online; accessed 03-Dec-2015].

[2] Applications of the tsp. `http://www.math.uwaterloo.ca/tsp/apps/`, 2007. [online; accessed 11-Dec-2015].

[3] B. Hemingway. Evolutionary computation. `http://courses.cs.washington.edu/courses/cse466/05sp/pdfs/lectures/10-EvolutionaryComputation.pdf`, 2005. [online; accessed 02-Dec-2015].

[4] T. Leng, R. Ali, J. Hsieh, and C. Stanton. A study of hyper-threading in high-performance computing clusters. *Dell*, 6(1):33–36, 2002.

[5] P. Lindberg. Performance obstacles for threading: How do they affect openmp code? `https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code`, 2009. [online; accessed 12-Dec-2015].

[6] R. Matai. *Traveling salesman problem: An overview of applications, formulations, and solution approaches*.

[7] P. McClean. Darwin's theory of evolution by natural selection. `https://www.ndsu.edu/pubweb/~mcclean/plsc431/popgen/popgen5.htm`, 1997. [online; accessed 03-Dec-2015].

[8] G. Reinelt. Tsplib. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`. [online; accessed 30-Nov-2015].

[9] M. Süß and C. Leopold. Common mistakes in openmp and how to avoid them. In *OpenMP Shared Memory Parallel Programming*, pages 312–323. Springer, 2008.