Parallel Simulated Annealing and Genetic Algorithms

COMS4040 - Project

Michael Beukman

July 3, 2021

1 Abstract

Optimisation algorithms are widely used today, and simulated annealing and genetic algorithms excel in solving discrete problems that do not provide gradient information. These algorithms are computationally demanding though, so we attempt to address this computational burden by implementing these two algorithms in parallel using MPI and CUDA, respectively. We analyse the advantages and disadvantages of each method in terms of performance and quality of solutions generated.

2 Introduction

Optimisation algorithms are ubiquitous in the modern day, and they are used for a variety of purposes, from optimising general functions (Corana *et al.*, 1987), to optimising neural networks with stochastic gradient descent (Bottou, 2010) to route planning (Waldock and Corne, 2011) and more.

While many problems provide gradient information in a continuous space, which leads to many different, effective algorithms that make use of this information (Scheithauer, 1999), there are those discrete problems that are unsuitable to this paradigm (Sergienko and Shylo, 2006). These include the travelling salesman, the knapsack problem, and a wide variety of graph-based problems.

Effective methods for these discrete problems have been devised, and we focus on two of these in this report, genetic algorithms and simulated annealing.

Genetic algorithms attempt to mimic the natural world's concept of evolution and mutation to solve general problems (Goldberg, 1989).
Simulated annealing is an optimisation algorithm that is analogous to the annealing process in metals, where they are slowly cooled to remove imperfections and get to a lower energy state.
While these methods are powerful, they can be computationally intensive, but conducive to parallelism. In this report, we implement the above methods in parallel using CUDA and MPI and apply them to a few well-known problems.

3 Background

3.1 MPI

MPI is a standard for programming model distributed memory systems, using communication through a network (Clarke *et al.*, 1994). Each process is referred to as a rank, and it has its own, independent address space. These different ranks can communicate with each other using one of the many MPI communication methods.

3.2 CUDA

CUDA is a programming paradigm that allows one to program NVIDIA graphics cards to solve general-purpose tasks (Sanders and Kandrot, 2010). Computation is performed on a per-thread basis, and these threads are organised into thread blocks, which share a common cache.

3.3 Genetic Algorithms

"Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics" Goldberg (1989). The main idea behind genetic algorithms is that there is a population of individuals, where each individual is represented by a genotype, which can be anything from bit strings, integer vectors, to the weights of a neural network. Each individual represents a solution to the problem, and the main goal of these methods is to find the best solution according to some criteria.

Every individual is then assigned a *fitness* score, which measures how close the individual is to the desired solution. This is important because it then allows the highest fitness individuals to be chosen to reproduce and pass on their genetic information.

This reproduction method is called crossover, where the genotypes of two parents are combined to form the genotype of the child. Mutation is then performed on the child, which randomly alters its genotype to maintain genetic diversity and facilitate exploration. Ultimately by adding individuals to the reproduction pool in proportion to their fitness, the highest fitness individuals reproduce more often. This results in good traits that perform well being maintained and ill-performing traits being discarded.

This procedure is then repeated many times or until a sufficiently good solution is found.

3.4 Simulated Annealing

Simulated annealing (SA) is a heuristic optimisation algorithm that is based on the physical process of annealing, in which metals are heated up and cooled slowly, to remove internal stresses and make the metal stronger (Rayward-Smith *et al.*, 1996). This process can be applied to a range of optimisation problems, where the current solution (which has energy E_1) is perturbed slightly, and the energy level E_2 is determined. The energy can be simply any metric that we want to minimise, like distance in the travelling salesman problem (Rayward-Smith *et al.*, 1996; Kirkpatrick *et al.*, 1983). If $E_2 < E_1$, we simply accept the perturbed solution. Otherwise, we accept the perturbed solution with a probability $\exp(-\frac{E_2-E_1}{T})$, where T is the current temperature, which starts off large and gradually decreases. This allows the algorithm to explore initially and get out of local optima, but as time goes on it explores less and focuses on the current best solution.

4 Problems

For this report, we focus on two main problems. Firstly, the travelling salesman problem (TSP), where the task is to provide an order in which to visit a set amount of cities with the goal of minimising the total distance travelled. This is a common problem and is used to benchmark many different optimisation algorithms (Larranaga *et al.*, 1999; Dorigo and Gambardella, 1997). For the travelling salesman problem, we validated each final solution by ensuring that each city appears only once. Thus, all parallel results reported actually produced valid solutions.

The second task is the problem of finding the minimum of a function, specifically the n-dimensional *Rosenbrock function*, which can be written as (Shang and Qiu, 2006):

$$f(\bar{x}) = \sum_{i=0}^{N-1} 100(x_i^2 - x_{i+1}^2)^2 + (x_i - 1)^2$$

with constraint:

$$-30 \leq x_i \leq 30 \qquad\qquad 0 \leq i \leq n \tag{1}$$

This function is also a common benchmark for numerical optimisation algorithms. We ignore the constraint for simplicity though.

5 Methodology

The main focus of this report is to implement a general framework that uses simulated annealing and a genetic algorithm for the above computation methods, and we test this by implementing a few concrete problems, as described above.

The main computationally intensive parts of a genetic algorithm are (1) evaluation, (2) crossover and (3) mutation, so these are the parts we focus on parallelising.

The most computationally intensive part of simulated annealing is to evaluate the energy of a candidate solution, but multiple perturbations can be done and considered in parallel.

5.1 Representation

For both simulated annealing and genetic algorithms, we use a simplifying assumption that a solution to a problem can be represented using an array of integers or floating point numbers. For example, a solution to the travelling salesman problem can be represented as an array of integers that indicate the order in which the cities will be visited in (Larranaga *et al.*, 1999). A minimum solution to a function can be represented as a single or multiple floating point numbers. As Goldberg (1989) noted, one could even represent each individual as a bit string, which could, in theory, represent anything.

We make this assumption as it simplifies working with memory across the different programming paradigms.

We allocate two buffer of memory initially, so that one stores the current population or solution and the other buffer stores the next one. This reduces the memory allocations, and at the end of each iteration, we can simply swap the pointers to these buffers.

5.2 General Implementation

Here are some general implementation details that are valid across both compute frameworks.

5.2.1 Genetic Algorithm

For the genetic algorithm, we partition the algorithm into a few distinct sections, which we show in pseudocode here:

Evaluation Evaluate the individuals and write their fitnesses to an array. This fitness is $\frac{1}{d}$ for the travelling salesman and $\frac{1}{f(x)}$ for the Rosebrock function, as we want to minimise d and $f(x)$.

Normalise Calculate the total fitness and normalise each entry in the array.

Breed Choose two parents from the current population, perform crossover on them to generate two children. Then proceed to mutate these children.

Swap Finally swap the current and next population pointers and repeat.

We make some additional choices that we found performed better. First of all, at each breeding step, the best two individuals are located and stored, unmodified, at the beginning of the next population. This is to ensure that mutations do not cause regressions (Katoch *et al.*, 2021). Secondly, we found that utilising elitism for selection, in which only the top k individuals are considered for breeding performs substantially better (in terms of time and final score) than choosing from all the individuals with probability proportional to their fitness value. We uniformly choose from the top 2 individuals in our case. We mutate the children with probability 1. We found this performed well in conjunction with only using the top two individuals as parents.

Mutation and crossover for the travelling salesman problem are similar to those described in Larranaga *et al.* (1999), specifically the simple inversion mutation and a modified two-point crossover. Mutation simply inverted a random range and crossover took a range from a parent and copied it to the child, and the other parent's cities were just added in order, without duplicates. For the Rosenbrock function, mutation was simply perturbing some values with a random amount. Crossover was as described in Carr (2014) where we set $c_1[i] = \beta p_1[i] + (1-\beta)p_2$, where $\beta \in [0,1]$ and i were random numbers and the rest of the genome was copied over from the parents.

5.2.2 Simulated Annealing

Simulated annealing consisted of simply perturbing the current solution, determining its energy and taking that as our current solution if the energy was lower, or with probability described above. The perturbing operations were similar to the above mutation operations for the genetic algorithm. We reduced the temperature using a linear scale, i.e. at every iteration we update $T = 0.995T$, with an initial T of $100 \times C \times \sqrt{2}R$ for the TSP, where C is the number of cities, and R is the size of range that city positions can take (i.e. max - min). For the Rosenbrock function, we used an initial $T = 1000 \times N^2$ where N is the dimension of the Rosenbrock function. These initial choices were empirically determined to perform well, and it makes sense to make the initial temperature dependent on the problem size.

5.3 MPI Implementation

5.3.1 Genetic Algorithm

Suppose we run a genetic algorithm of population size P on N ranks. We assume that $K = P/N$ is even.

Since communication in MPI is often the limiting/bottlenecking factor, we attempt to minimise this as much as possible. Each rank evaluates its whole population, as well as breeding the whole population to form the next generation. Every I generations (usually 10), the ranks communicate and share their current populations, which is the starting point for individual evolution in the other 9 generations. We simply perform an *AllGather* operation, where each rank sends its two best individuals, as well as $K-2$ others. Thus we get a measure of genetic diversity, and the method is not simply performing N independent executions of a genetic algorithm.

In general, there are two main ways to implement a parallel genetic algorithm (Rajan and Nguyen, 2004). One is a master-slave model, where the root process gives out work to the compute processes. The other is an island migration model, where each processor has an entire population, which gets independently evolved and occasionally migrated between processors. We implemented both methods, and they have their drawbacks. The master-slave model, while producing similar results to the serial version, could not scale past a certain factor (2x in our case faster than a serial version when using 14 processors). This is potentially due to the large amount of communication, or that the master process can be the limiting factor. The island population model could scale better, but when performing a smaller number of generations when the number of processors was increased reduced the quality of the results, due to the infrequent communication. We focus on and report results only for the island method.

5.3.2 Simulated annealing

To implement simulated annealing using MPI is quite simple. Suppose each rank starts with a random solution. Then, each rank independently performs simulated annealing on that individual for 100 iterations. After this, we simply determine which rank had the best solution (using AllReduce), and copy that solution over to all the other ranks (using Broadcast). This solution is the starting point for the next 100 iterations. This communication interval can be adjusted, and increasing it would result in less overall communication, which improves scalability, but potentially at the cost of worse results.

This is very light on communication, as we only broadcast one individual and a few floating point numbers every 100 steps.

5.4 CUDA

5.4.1 Genetic Algorithm

For the CUDA implementation, we launch a few kernels sequentially. We first spawn enough threads such that each thread evaluates one individual. Then we run a new kernel with half as many threads, where each thread takes in two parents and breeds them to generate two children. The evaluation one simply evaluates all the individuals and stores their fitness. Since we only use the top two individuals, we do not normalise the fitnesses, which results in a substantial speedup. The two best agents are calculated in block 0. We calculated the second-best individual somewhat fuzzily, and it's not guaranteed that this is the best individual, but it is one of the best. This was done since it was much faster and performed similarly to calculating the second-best exactly. Finally, each thread breeds two parents to produce two children. Since we only use two parents to breed, we store them into shared memory to facilitate fast memory accesses. For the TSP, we created a shared memory array of booleans by using the individual bits inside an array of *int32*'s. This was used to change the crossover algorithm from $O(n^2)$ to $O(n)$, where n is the genome size, and it resulted in a significant speedup.

5.4.2 Simulated Annealing

We use a multi-tiered system in that each thread in a block starts with the same solution and then performs S iterations independently. Then we determine which solution was the best per block and save that and the score to global memory. Finally, we perform a reduction across blocks and save the overall best solution into global memory and copy it to constant memory (for fast broadcasts). Then we start again using that solution. Empirically, we found that using $S = 1$ and exchanging information more frequently

was better than using fewer overall iterations and a larger S (which took about the same amount of time). Using only one iteration allowed us to perform some optimisation regarding replacing the current solution with the candidate solution, which we only had to do once per block instead of once per thread.
We store the current block's solutions and scores in shared memory and only write the final individual per block to global memory. When shared memory is too small for all this data, we store the solutions in global memory, but keep the per-block scores in shared memory. We also store the the city locations for the TSP in constant memory.

6 Experiments

Since the different algorithms were implemented differently for each framework, we do not directly compare the performance. Instead, we take a more holistic view by comparing the quality of the final solution (e.g. final distance in the TSP), as well as how long it took to obtain that solution. We also compare the different methods (simulated annealing vs genetic algorithms) per problem/programming model to determine which one performs better under which constraints. We contrast the above with the serial version to determine the speedup that these parallel methods can obtain.

We measure and compare the quality of solution when iterating for a comparable amount of time, the time it takes to complete one generation or iteration as well as the speedup when increasing the number of processors for a constant problem size (strong scaling), and when increasing both the problem size and processors proportionally (weak scaling).

The experiments were run on nodes with 14-core *Intel(R) Core(TM) i9-10940X CPU @ 3.30GHz* processors and NVIDIA GeForce RTX 3090 GPUs. We used gcc 7.5.0, OpenMPI 4 and CUDA 10.
The compile flags used were *-std=c++14 -O3 -march-native* and *-use_fast_math* for CUDA. The communication framework was 1Gb ethernet.

We average over 3 different problem initialisations, and for each of these, we perform 10 runs, discard the first one as a warmup and average the remaining 9. We report the overall average in the following plots. All time shown in our results is wall-time, in milliseconds, measured using the *chrono* library.

6.1 Random Number Generation

Since this project relies on random numbers to perform the mutation, crossover and other operations, the quality of these random numbers are important. One might think that simply using a normal generator will work fine, but when generating these numbers in parallel, the numbers on different processing elements can be highly correlated, and different to the sequence of numbers generated by the sequential version (Tomassini *et al.*, 1999; L'Ecuyer *et al.*, 2017; Salmon *et al.*, 2011). There has been much work on implementing robust parallel random number generators, and we simply use one of these, *Threefry* (Salmon *et al.*, 2011). We specifically use an open-source C++11 implementation created by Thijs van den Berg[1]. For CUDA, we simply used the random number generators provided by NVIDIA.

6.2 Evaluation

Since the methods are inherently stochastic, and the parallel implementations are different in character to the serial one, validation was difficult. We thus mainly determined if the parallel versions converge to the same (or better) value as the serial implementation. We also ensure the generated result is valid, for example in the TSP, each city occurs only once per solution.

7 Results

Our main analysis is regarding the travelling salesman problem, and we discuss simulated annealing in Section 7.1 and genetic algorithms in Section 7.2. In Section 7.3 we briefly discuss the results for the Rosenbrock problem and in Section 7.4 we investigate the scaling properties of our MPI code. This could not be done for CUDA, as we didn't implement Multi-GPU support.

We did indeed run our MPI code across multiple nodes, as our results in Figure 1c, Figure 2c and Figure 2d show. Each multiple of 14 ranks represents another node, and we ran on 1, 2, 4, 8 and 16 nodes, which is shown as the red ticks in Figure 2c. We only oversubscribe when explicitly stating so.

7.1 Simulated Annealing

In Figure 1a we see that for a large problem size, 14 ranks and the CUDA implementation converged to the same value, but the best CUDA method reached it slightly faster.

[1] `https://github.com/stdfin/random`

The quality of the solution, as well as the speed, is better when using 14 ranks compared to serial. The purple line, which is CUDA using a block size of 4, is faster than the normal CUDA with a block size of 32 (with an equivalent number of total threads). This is simply because, with a smaller block size and a larger problem, the individuals could fit inside the shared memory, which is not the case for the larger block size. In general, it is suboptimal to only use a block size of 4 though, as threads are idle and it does not use the GPU to its full capacity.

The serial implementation performs similarly to using 1 MPI rank, as shown in Figure 1a. We thus only report the serial results from now on. In Figure 1c, we show that CUDA still performs more operations per second than MPI, but this tapers off as the problem size increases due to the aforementioned shared memory problem. The number of operations were estimated as processors × iterations × problem size, as it is somewhat difficult to obtain an exact value for the number of floating point operations. Using 224 ranks on 16 nodes performs badly due to slow communication.

7.2 Genetic Algorithm

In Figure 1b we can see that the genetic algorithm also converges to the best solution, but it does so much slower than the simulated annealing method. The genetic algorithm scaling, for a constant per rank problem size on one node is shown in Figure 2b.

7.3 Rosenbrock

in Figure 1d we can see that for larger problem sizes, simulated annealing does not reach the global minimum of 0, but it improves when increasing the number of processors. In general, simulated annealing for continuous problems is generally done with some additional features, like discretising the state space (Ingber, 1993) or using a step vector to guide new directions to search in (Corana *et al.*, 1987). We didn't use these enhancements, so that might explain why the results were sub-par.

When using CUDA, however, we always reached the global minimum, because we used many processors. Thus in each iteration of simulated annealing, we considered many different candidate solutions and chose the best. The CUDA runs generally took longer, however, so potentially running the MPI versions with more iterations, or a different temperature schedule could work.

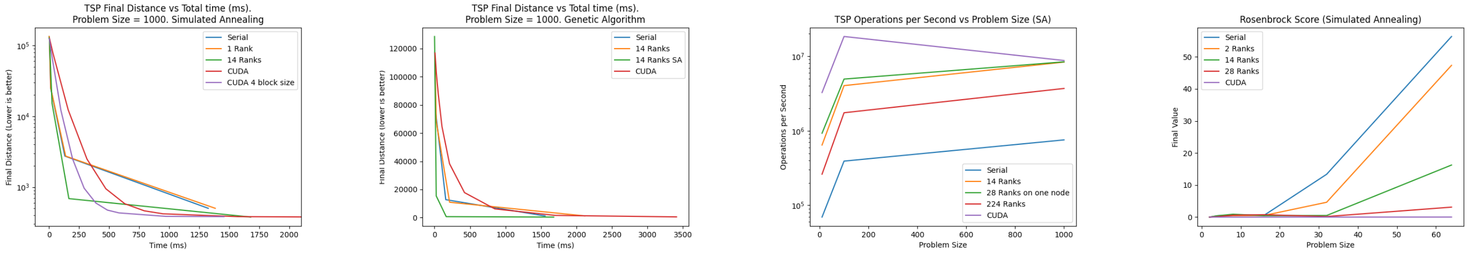
7.4 Scaling

In Figure 2d, when increasing the number of processors, but also proportionally increasing the total amount of work to be done, such that each processor performs a constant amount of work, we can see that the total time is roughly the same, with about a 20% increase from 1 rank to 14 ranks. When running on multiple nodes, the slow interconnect becomes a bottleneck, and the performance decreases. The final score does improve as the number of ranks increased, as we performed more iterations in total. In Figure 2c, when the total amount of iterations remain constant, and we increase the number of processors, we achieve near-perfect speedup for a single node. In this case, to achieve the exact same level of quality, we would need to decrease the number of iterations by roughly half the processor count to account for the communication that does not happen regularly (i.e. to get the same quality on 14 ranks as one rank running I iterations, you can use about $\frac{2I}{14}$ iterations instead of the perfect scaling $\frac{I}{14}$). This point is shown in Figure 2a, where the 'Corrected' line shows the time it takes to achieve comparable solution quality to the original 1 rank run. We can also see that oversubscribing (i.e. using 28 ranks for only 14 physical cores) does not provide a benefit in this case, which is also illustrated in Figure 1c.

Thus our code scales quite well, but communication becomes a bottleneck when running on multiple nodes, and the performance drops significantly. Having access to a faster interconnect will improve the scalability of the code across multiple nodes.

8 Limitations

There are some limitations in our methods, notably that the genetic algorithms performed sub-par when parallelising them in general, potentially due to higher communication costs, or even suboptimal crossover and mutation operations. The CUDA method had some downsides, namely slowdowns due to synchronisation and a limited amount of shared memory, which does limit the effective problem size that we can solve. Optimising some of the parameters of the methods, like communication frequency, population size and temperature scaling could also improve our results.



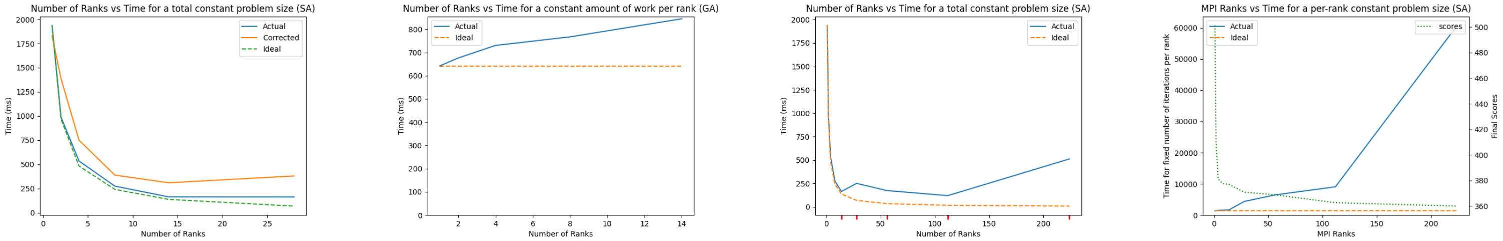
(a) How quickly each method gets to a result

(b) Genetic Algorithm Results

(c) Number of operations vs problem size

(d) Rosenbrock Final score vs problem size

Figure 1: Algorithm Results



(a) Scaling on one node (SA)

(b) Genetic Algorithm Scaling

(c) Strong Scaling, Multiple Nodes (SA)

(d) Weak Scaling, Multiple Nodes (SA)

Figure 2: Scaling Results

9 Discussion

In general, we can see that the simulated annealing approach was superior to the genetic algorithm in terms of sample efficiency and time taken to obtain valid results. The amount of communication for the genetic algorithm was also much higher since it needed to exchange the entire population every few iterations. Simulated annealing only needed to communicate the single best individual every 100 iterations, which is much less data.

The CUDA implementations were much better than the serial version, and comparable, or slightly better than the 14 rank MPI version. The MPI version has the potential to scale much better across multiple nodes with a fast interconnect though. We also saw that our methods generalise to different problems, like function minimisation with relatively little extra work.

10 Conclusion

In conclusion, this report investigated parallel simulated annealing and genetic algorithms to solve a few common problems. We parallelised these methods using CUDA and MPI respectively. The most general, efficient method proved to be simulated annealing using MPI, because, unlike CUDA, it has the potential to scale up to achieve much larger speedups than what we could see on one node. CUDA, despite its limitations, performed well, and this massively parallel model allowed us to perfectly solve harder and larger problems.

We thus developed parallel methods that generate solutions of comparable or better quality than the serial version, in much less time, so the project was an overall success. The full code is available at https://github.com/Michael-Beukman/ParGA.

11 Acknowledgements

Computations were performed using High Performance Computing infrastructure provided by the Mathematical Sciences Support unit at the University of the Witwatersrand.

References

[Bottou 2010] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[Carr 2014] Jenna Carr. An introduction to genetic algorithms. *Senior Project*, 1(40):7, 2014.

[Clarke *et al.* 1994] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In Karsten M. Decker and René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, Basel, 1994. Birkhäuser Basel.

[Corana *et al.* 1987] Angelo Corana, Michele Marchesi, Claudio Martini, and Sandro Ridella. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 13(3):262–280, 1987.

[Dorigo and Gambardella 1997] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the travelling salesman problem. *biosystems*, 43(2):73–81, 1997.

[Goldberg 1989] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*, chapter 1. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[Ingber 1993] Lester Ingber. Simulated annealing: Practice versus theory. *Mathematical and computer modelling*, 18(11):29–57, 1993.

[Katoch *et al.* 2021] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multim. Tools Appl.*, 80(5):8091–8126, 2021.

[Kirkpatrick *et al.* 1983] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[Larranaga *et al.* 1999] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[L'Ecuyer *et al.* 2017] Pierre L'Ecuyer, David Munger, Boris Oreshkin, and Richard Simard. Random numbers for parallel computers: Requirements and methods, with emphasis on gpus. *Mathematics and Computers in Simulation*, 135:3–17, 2017.

[Rajan and Nguyen 2004] SD Rajan and DT Nguyen. Design optimization of discrete structural systems using mpi-enabled genetic algorithm. *Structural and Multidisciplinary Optimization*, 28(5):340–348, 2004.

[Rayward-Smith *et al.* 1996] Vic Rayward-Smith, Ibrahim Osman, Colin Reeves, and G. Simth. *Modern Heuristic Search Methods*. 01 1996.

[Salmon *et al.* 2011] John K Salmon, Mark A Moraes, Ron O Dror, and David E Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[Sanders and Kandrot 2010] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[Scheithauer 1999] Guntram Scheithauer. *Jorge Nocedal and Stephen J. Wright: Numerical Optimization, Springer Series in Operations Research, 1999, ISBN 0-387-98793-2*, 1999.

[Sergienko and Shylo 2006] I. Sergienko and Vladimir Shylo. Problems of discrete optimization: Challenges and main approaches to solve them. *Cybernetics and Systems Analysis*, 42:465–482, 07 2006.

[Shang and Qiu 2006] Yun-Wei Shang and Yu-Huang Qiu. A note on the extended rosenbrock function. *Evolutionary Computation*, 14(1):119–126, 2006.

[Tomassini *et al.* 1999] Marco Tomassini, Moshe Sipper, Mose Zolla, and Mathieu Perrenoud. Generating high-quality random numbers in parallel by cellular automata. *Future Generation Computer Systems*, 16(2-3):291–305, 1999.

[Waldock and Corne 2011] Antony Waldock and David Corne. Multiple objective optimisation applied to route planning. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1827–1834, 2011.